USENIX Association

Proceedings of the 18th USENIX Symposium on Networked System Design and Implementation

April 12–14, 2021

ISBN 978-1-939133-21-2

Conference Organizers

External Reviewers

Message from the NSDI ’21 Program Co-Chairs

Welcome to NSDI ’21!

It is an understatement to say that the past year has been a stressful and difficult time for us all. The pandemic, the associated economic disruption, and the large-scale social movements for racial equality have given us reasons for both hope and grief during the last 12 months. Placed in this context, our academic work as computer scientists can feel small. Many of us struggled with our health (mental, physical, or otherwise) in the past year, as we grappled with the era in which we live. The fact that NSDI is still happening this year is a testament to the graciousness with which our community has treated each other during these hard times. As paper submitters, paper reviewers, PC members, and conference organizers, we helped each other with our kindness and our flexibility; by showing compassion in the face of last-minute schedule disruptions and requests for a little extra time, we made it possible for a huge undertaking like NSDI to move forward in uncertain circumstances.

NSDI ’21 received 369 papers in total: 114 in the spring deadline, and 255 in the fall deadline. 59 papers were accepted, for an acceptance rate of 16%. Papers were reviewed by a group of 70 experts from both academia and industry. We sincerely thank those reviewers, who provided thoughtful feedback during an enormously disruptive time. We also thank the paper authors; your submissions are what make NSDI such a great venue, and we hope that you will enjoy the conference program.

We’d like to thank all of the USENIX staff who helped us to organize this year’s conference amidst such extraordinarily challenging circumstances. At every step in the process, from configuring the HotCRP server to dealing with camera-ready production, the USENIX staff provided invaluable advice and implementation. So, we send a heartfelt thanks to Casey Henderson, Olivia Vernetti, Camille Mulligan, Arnold Gatilao, Jasmine Murcia, Jessica Kim, Julia Hendrickson, and the rest of the USENIX team. Our community is lucky to have support from such dedicated USENIX staff.

We are excited that NSDI ’21 will be held, even if the conference presentations will be virtual. We thank the efforts of everyone who made this possible, whether you submitted papers, reviewed papers, helped with the organization of the conference, or will just be attending the conference. We hope that you are safe and healthy, and that the next year will be easier and more joyful than the last.

James Mickens, *Harvard University*
Renata Teixeira, *Netflix*
NSDI ’21 Program Co-Chairs

18th USENIX Symposium on Networked Systems Design and Implementation (NSDI ‘21)

April 12–14, 2021

Accessing Cloud with Disaggregated Software-Defined Router

Hua Shao[1]*, Xiaoliang Wang[2,3]*, Yuanwei Lu[2], Yanbo Yu[2], Shengli Zheng[2], Youjian Zhao[1]
[1] *Tsinghua University,* [2] *Tencent,* [3] *Nanjing University*

Abstract

The last decade has witnessed a rapid growth of public clouds. More and more enterprises are deploying their applications on the cloud platform. As one of the largest public cloud providers, Tencent cloud serves tens of Tbps inbound-/outbound traffic via cloud gateways for customers with diverse cloud access requirements. Traditionally, cloud gateways were built with proprietary routers. From years of experience operating cloud network, we found that commodity router based cloud gateways are hard to scale, lack of extensibility and are difficult to inter-operate with the SDN-based cloud networks. To this end, we build our own Disaggregated Software-defined Router (DSR) to serve cloud access traffic. We architecturally split cloud router functionalities into several disjoint modules: 1) an access module built out of off-the-shelf commodity switches; 2) a software-based fast and scalable forwarding module; 3) a robust and scalable routing module built with commodity servers; 4) an SDN control module for traffic management and devices configuration. All the components can be independently scaled and maintained. DSR can deliver new network features at high velocity and has sustained the rapid growth of the cloud access traffic. In this paper, we present the design, implementation and our years of operational experiences of DSR.

1 Introduction

With customers increasingly deploying their computation and storage services in the cloud, public cloud services have achieved rapid growth in recent years [11, 16, 18, 35, 37]. As one of the largest cloud providers, Tencent cloud has provided a wide range of services including computing, storage and CDN to support a diverse set of customizable solutions across multiple industries such as e-commerce, online education and mobile games. To satisfy the cloud access requirements of worldwide customers and their end-users, it is critical for the cloud provider to set up fast, secure, stable and low-latency

*Joint first authors

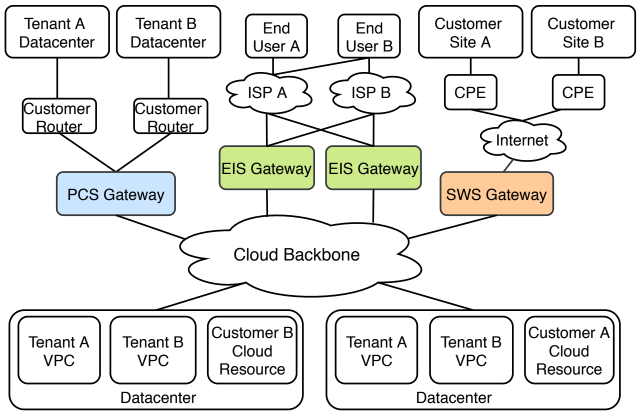


Figure 1: Various types of cloud access scenarios

connection among cloud customer's on-premise datacenters, distributed sites, Internet end-users and the cloud resources.

Tencent cloud supports various types of solutions to facilitate customers' cloud access requirements, namely Private Connect Service (PCS), Elastic Internet Service (EIS) and Software Defined WAN (*i.e.* SD-WAN) Service (SWS).

Private Connect Service. PCS establishes a direct connection between customers' on-premise datacenters and public cloud. It bypasses the Internet and offers higher reliability, faster speed and lower latency than public Internet connections. As demonstrated in Figure 1, customers access to their virtual private clouds (VPCs) hosted in the public cloud through PCS gateways. In our network, each PCS gateway supports $\sim$10K tenants, one of which has up to 20K routes. PCS provides guaranteed network quality, like consistent network performance on latency and bandwidth with low cost.

Elastic Internet Service. Many customers, *e.g.* content providers and online education institutes, run their computing and storage services in our cloud datacenters. EIS allows global end users to access these online services. As shown in Figure 1, EIS gateways are widely deployed at the Points of Presence (PoPs), interconnecting with multiple Autonomous Systems (AS). EIS gateway needs to fulfill several important tasks, *e.g.* BGP peering with different ISPs, management for inbound/outbound traffic across optimized network routes and forming demilitarized zone (DMZ). As the main entrance of a cloud region, EIS gateway needs to support a large forwarding table. In our settings, EIS gateway needs to support up to 10M longest prefix match (LPM) entries.

SD-WAN Service. Software-defined WAN Service (SWS) allows enterprise's branch offices to access their resources on the cloud through encrypted connections over any mixed transport services, such as LTE, Internet broadband, and MPLS, as illustrated in Figure 1. By leveraging IPSec VPN, WAN optimization (*e.g.* application-aware QoS, robust redundant packet transmission and data compression) and software-defined management/orchestration technologies, SWS simplifies IT infrastructure deployment and network management to connect users at distributed sites to applications on the cloud securely.

Cloud gateway is the main component in the access sites to meet these access requirements. Traditionally, the cloud gateway applies proprietary commodity routers to provide large-scale port extension, high speed traffic routing and forwarding, as well as access control against illegal connections or DDoS attack. However, with the rapid increase of customers, the traditional solution can not sustain anymore. First, we can not deploy millions of forwarding table entries in the commodity routers to meet our customers' requirements. Second, the lack of programmability of the proprietary devices slows down the feature roll-out velocity. We have to count on vendors' development plan which can be months or even years. Third, it is hard to inter-operate between the cloud SDN network and the edge BGP network. To meet these operational challenges, we provide ingenious solution for diverse types of customers to access the cloud (see § 2.2 for more details).

We set out to build a Disaggregated Software-defined Router (DSR) to address the above challenges. It enables cost-effective scaling to keep up with the fast growth of traffic volume. It leverages software feature velocity to provide fast response to user requirements. To be specific, DSR consists of a high performance data plane forwarding module, which is responsible for operations like routing table lookup, tunnel traffic encapsulation/decapsulation, IPSec encryption/decryption and packets forwarding. A standalone BGP routing module is built to exchange routing information between cloud network and external peers. SDN controllers are used to realize configuration, state management and orchestration of different components. Last but not the least, it performs routing decisions in a centralized fashion, thus yields better traffic steering. We leverage a disaggregated architecture, *i.e.* each component can be scaled independently and released on demand.

DSR serves tens of Tbps traffic, which has been deployed in Tencent cloud for over 3 years. In this paper, we introduce the design and implementation of the scalable and flexible system to access the cloud. We first use an example of the PCS connection to demonstrate the limitation of commodity routers based gateways (§ 2). Referring to the structure of commodity devices, we explain the architecture of DSR (§ 3). We present the system design details (§ 4, § 5, § 6) and share our operational experiences to the community (§ 8).

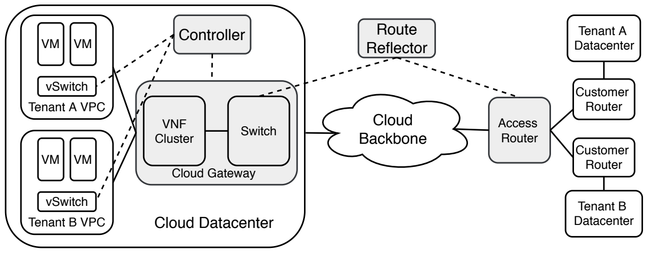


Figure 2: Commodity router based PCS

2 Background and Motivation

For ease of understanding, we first demonstrate the original solution of PCS using commodity devices. Then, we introduce the limitation of commodity router-based architecture which motivates the design of DSR.

2.1 Commodity Routers based PCS

Previously, PCS service is built by leveraging the MPLS VPN technique [7]. Figure 2 shows the architecture of the original version of PCS, which consists of the following components:

- **Access Router (AR):** It is a commodity router supporting various types of electrical or optical interfaces with different rates, *e.g.* 1Gbps/10Gbps/100Gbps. AR sets up BGP sessions for routing between customers' on-premise datacenters and public cloud. Tenants are isolated based on Virtual Routing and Forwarding (VRF) tables. AR works as an MPLS provider edge (PE) router of the cloud MPLS backbone network.

- **Cloud Gateway (CG):** CG is deployed in cloud datacenters as another MPLS PE node of the backbone network. It is built by leveraging a commodity switch and a server-based Virtual Network Function (VNF) cluster. The commodity switch is the gateway between datacenter network and MPLS backbone network. Tenants are isolated through the switch build-in VRF. Due to the limitation of routing entries in switch, the routing table from the on-premise datacenters to tenant Virtual Machines (VMs) are stored in the VNF cluster.

- **Route Reflector (RR):** RR is a commodity router. RR establishes MP-BGP sessions with AR and CG respectively. It conveys customer's on-premise IP routes to the cloud switch and vice versa.

- **Controller:** Controller is introduced to manage and configure network components. For example, controller installs a default route to CG at virtual switch (vSwitch) of each server for traffic destined for on-premise datacenters.

Now we explain how a packet is forwarded from a tenant virtual machine to its on-premise datacenter. For the outbound traffic, the vSwitch first encapsulates packets with GRE [13] header (The VPC id assigned to the tenant is encoded in the GRE header) and forwards the packet to VNF deployed in a server cluster. The VNF decapsulates the GRE tunnel and encapsulates the packet with a VLAN header which embeds a certain VLAN id according to the VPC id. Then the packet is forwarded to the cloud switch, which forwards the packet to the MPLS backbone based on the forwarding table. Whenever AR receives the packet, it forwards the packet to the destination.

For the inbound traffic, the main difference is at the cloud gateway. The cloud switch is configured with a default route to forward all traffic to the VNF cluster, which stores all the routing information for VMs in the cloud datacenter. The gateway supports the access control through firewall and safeguards applications through Distributed Denial of Service (DDoS) protection service running on dedicated servers. Finally, the VNF cluster encapsulates the packet with GRE tunnels and forwards it to certain physical machines according to its routing decision.

2.2 Motivation

Based on production experiences of operating the network, we found many limitations of the commodity router to meet the rapidly growing customers and their rising demands.

For AR, though the commodity devices can forward a large volume of traffic, *e.g.* Cisco ASR 9000 series routers [5] support tens of Tbps throughput, it cannot scale to support a large number of tenants, *i.e.*, the number of VRFs and LPM table entries are rather limited. The commodity access router only supports less than 1K of VRF elements and 1M FIB entries which cannot satisfy the requirements of PCS which needs to support $\sim$10K tenants and EIS which requires $\sim$10M forwarding tables. We have to expand network by using additional routers. It is not cost-effective and substantially increases the management and control complexity.

We usually need to roll out new network features in few weeks to meet our customers' requirements, *e.g.*, supporting jumbo frames or 4B-length AS-path attribute. In addition, the cloud providers usually deploy new functions to improve operation and maintenance capabilities, *e.g.* measurement of top-N largest flows. However, due to the lack of programmability and limited device management API of proprietary commodity routers, it can take several months or years if we need vendors to make change to device software or hardware [35, 41]. The slow feature velocity significantly affects the user experiences and the reputation of the cloud provider.

BGP protocol is widely used to interconnect autonomous networks in the Internet [35, 41]. For instance, MPBGP protocol is used to exchange routing information between access router and cloud switch. On the other hand, the cloud networks have increasingly adopted the SDN technique for efficient traffic steering and fast network failure convergence. We found many problems for inter-operation between the cloud SDN network and the external BGP network. For example, in commodity router based PCS, the controller should update the state of the direct connection in real time, such that if the direct connection fails, vSwitch can carry out corresponding routing updates and quickly switch to the backup path. To achieve that, the controller has to periodically pull the routing tables from CG. However, due to the slow NETCONF messages processing, the long pulling interval leads to slow routing convergence.

3 Design

We target at building a general platform, *i.e.* a disaggregated software-defined router (DSR), to meet various types of cloud access requirements.

3.1 Design Rationale

DSR should meet the following requirements:

- **Simplicity**. The commodity router needs to support all kinds of network standards and a variety of customized protocols for cloud network, enterprise network or campus network, etc. In cloud access scenarios, many features integrated in the commodity routers are not required. We can simplify the design by supporting a minimum set of functionalities, *e.g.*, BGP and static routing for inter-operation with external network, Bidirectional Forwarding Detection (BFD) [23], Internet Protocol Service Level Agreement (IPSLA) [10] for fast convergence, and VXLAN [27] and GRE [19] for tunneling.

- **Scalability**. To meet various cloud access requirements of Internet users, customer on-premise datacenters and enterprise branch-sites, the key components of the cloud gateway, *i.e.* data plane, routing and control plane should be able to scale independently without affecting each other.

- **Reliability**. All the key components are built in the cluster with redundancy such that no single point failure can cause performance degradation. Specifically, the key components can be deployed cross available zones for remote disaster recovery.

- **Elasticity**. The system should support high feature velocity and realize efficient inter-operation between the external

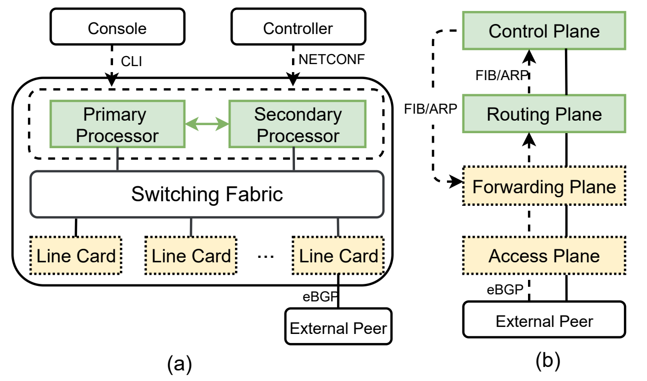


Figure 3: Sketch of (a) a typical commodity router and (b) the disaggregated software-defined router architecture

BGP network and the internal software defined cloud network in real time.

3.2 System Overview

3.2.1 The disaggregated design

We review the architecture of a typical commodity router. As shown in Figure 3 (a), it consists of three major components:

- **Processor**: Processor acts as the control plane, which runs network protocols like BGP, OSPF and BFD. Usually, there is a secondary processor for high availability.
- **Switching Fabric**: Switching fabric is responsible for high speed internal inter-connection.
- **Line Card**: Line card is intended to connect many users with different types of interfaces. It forwards and filters packets based on routing tables and access control lists (ACLs) respectively.

The three components are tightly coupled in the commodity router. The processor capability, routing table sizes and bandwidth are configured with a fixed ratio. We can not independently scale any component on demand.

Consequently, we turn the router into a disaggregated architecture, as illustrated in Figure 3 (b):

(i) The functions of line card are divided into two components: the access plane and the forwarding plane. The access plane provides various types of interfaces and supports layer 2 forwarding. Based on the operational experiences, the access plane is stable, *i.e.* we do not need to frequently update the access plane. As a result, we build the access system with a group of small-scale commodity switches. The forwarding plane deals with layer 3 packets processing with large scale forwarding tables (FIB, IPSec, QoS, etc). We introduce the software-defined forwarding module to achieve high feature velocity and scalability.

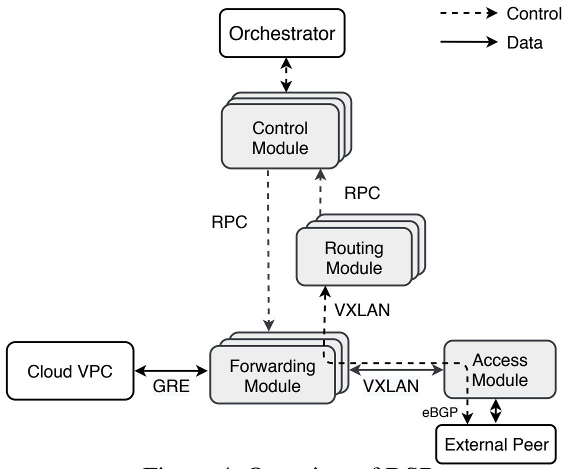


Figure 4: Overview of DSR

(ii) The functions of the processors are divided into two components: the routing plane and the control plane. They are implemented with software over common servers. The routing plane takes charge of the protocol and routing management, like BGP, BFD, IP-SLA and static routing protocols. The routing information is delivered to the control plane through RPC messaging. The control plane stores the routes, generates the forwarding tables, and then installs the FIB/ARP tables to the forwarding plane. This design is based on the observation that the routing plane and control plane require different resource settings. For example, the routing plane requires high computational capacity for complex protocol processing. Meanwhile, the control plane deals with performant management and ensures consistent forwarding processing during upgrade. We use different software suites (DPDK, ONOS&ODL) there. As a result, the development and release cycles are different for these two planes.

(iii) We use the standard VXLAN protocol for the interconnection of different planes. Tenants are isolated based on VXLAN VNI. All the components are decoupled and deployed in different fault domains which allows independent scaling. Generally, the access plane is deployed at the access sites, *e.g.* PoPs or remote user sites. The others are deployed in the cloud datacenter.

(iv) The forwarding, routing and control planes are built using multiple servers in a cluster. No single server failure can cause the break down of the entire system.

3.2.2 System architecture

The architecture of DSR is shown in Figure 4. It includes the following components.

Access module. We apply commodity off-the-shelf switches for the inter-connection between the internal cloud network and the external peers. It supports protocols of BG-

P/OSPF/BFD/LACP (Link Aggregation Control Protocol) at underlay network. It provides various types of physical ports with different rates, *e.g.* 1GE/10GE/100GE. For the overlay network, it realizes layer 2 isolation for tenants using the switch built-in VLAN functionality. VXLAN tunnels are established for communication between the access module and the forwarding module.

Forwarding module. The forwarding module is responsible for high performance packet processing and routing with large forwarding tables. It sends BGP/BFD messages to the routing module via layer 2 VXLAN tunnel and forwards data path traffic VPCs via self-defined GRE tunnel in the cloud datacenter. We build the forwarding module using a group of servers. With the server built-in large memory, it can support a large number of VRFs and large LPM tables for tenant isolation and packet forwarding. Forwarding module also supports IPSec processing and WAN optimization functionalities.

Routing module. The key purposes of the routing module are: (i) inter-connecting with the external peers (commodity devices) through dynamic routing using BGP; (ii) supporting a large amount of customers and maintaining the neighbors information; (iii) realizing fast convergence and failover when routing information updates and network failure happens. The core of the routing module is a high performance home-made BGP speaker. We optimize the BGP speaker performance and customize it for different scenarios. For example, in PCS, we embed the bandwidth allocation ratio for customers' traffic in the BGP header. While, in ISP peering scenario, we select the routing path based on the network state through SDN controller. It supports Non-Stop Routing & Forwarding (NSR& NSF) for high availability. We optimize its ability to handle a large number of BGP updates. For outbound traffic, the control module feeds VPC routes into the routing module, which in turn conveys the routes to the external peer. As the BGP messages are carried over VXLAN tunnels, they are able to establish BGP sessions directly between the external peer and routing module.

Control module. Control module acts as a local controller. It stores the routing information and is responsible for optimal traffic steering computation. Apart from that, control module provides distributed message queues in order to efficiently synchronize dynamic forwarding rules among forwarding module clusters and routing module clusters.

Orchestrator. The orchestrator acts as a global controller. It is responsible for distributing operator's configuration requests to corresponding control modules. The Orchestrator collects particular traffic scheduling requests from external management system and synchronizes them to the control module. It is a centralized routing computation platform by taking into consideration of various kinds of metrics, *e.g.* network latency, bandwidth capacity and monetary costs, to achieve consistent performance while reduce costs for customers [35, 41].

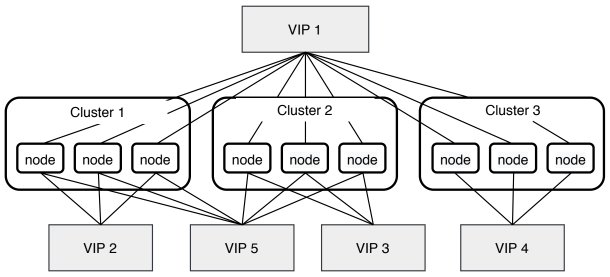


Figure 5: Scaling out forwarding plane through multiple VIPs

3.2.3 Challenges

Since DSR components are loosely coupled with each other, we can easily add or remove one instance in any component independently for scaling or upgrading purposes without affecting the whole system. For the design and implementation of each specific module, we meet the following challenges:

- We need to ensure high forwarding and routing capacity in software given a large number of tenants and highly frequent routing updates. To be specific, DSR serves 10Ks of tenants and maintains 10M entries routing tables at line rate. We have optimized the forwarding module and the BGP speaker of the routing module to address this challenge. Meanwhile, we provide low latency packet forwarding by minimizing the impact of system call, CPU scheduling and packet losses.
- Originally, the commodity router is a single device to serve the arrival traffic. However, the DSR is a distributed architecture consisting of multiple components. The instances of each components are deployed in the cluster, which are connected through multiple-path Clos network. The disaggregated design makes the management more complex than managing an individual commodity router. We need to carefully schedule and route the arrival traffic and control messaging inside DSR, which can easily affect the performance and stability of the whole system.

4 Scalability

We explain the optimization on forwarding module and the routing module to address the scalability issues.

4.1 Scalable Forwarding Plane

In order to provide a scalable forwarding module to meet the packet processing requirements, our efforts go along two directions: (i) Scaling out the packet processing components by adding new cluster of servers; (ii) Optimizing the program of packet processing and LPM lookup to ensure fast forwarding at large scale.

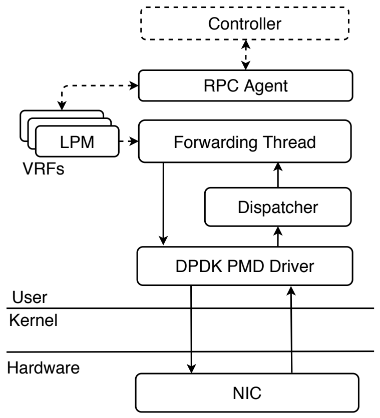


Figure 6: Architecture of the dataplane forwarding module

4.1.1 Scale out with multi-VIP

For building a scalable forwarding module based on common servers, a straightforward approach is to expose a single virtual IP (VIP) for all servers in the cluster. We announce the VIP to all tenants. All servers are then configured with the same forwarding tables. By doing so, arrival traffic are balanced to servers using ECMP hashing at a per flow basis. With this stateless load balancing, the system's processing ability can be easily scaled out/in by adding/removing servers. However, with more and more VRFs are configured, the forwarding capability of the cluster is limited since each server has to support the entire routing tables of VPCs in the region.

In fact, in our production network, the tenants' requirements on bandwidth are diverse. Most tenants require traffic bandwidth less than 10Gbps. Only a few tenants have traffic around 100Gbps. We seldom see tenants have traffic more than 500Gbps or over the forwarding capability of a single cluster. To this end, we introduce a flexible multi-VIP structure. As shown in Figure 5, each cluster is configured with multiple VIPs shared by tenants. For example, VIP 2 belongs to cluster 1, which serves multiple tenants with little traffic. Meanwhile, VIP 5 is applied to cluster 1 and cluster 2, which are assigned to users with large amount of traffic, *i.e.*, these users can deploy forwarding tables in both cluster 1 and cluster 2. Similarly, a tenant with an extremely large amount of traffic can be fulfilled by using VIP 1. Consequently, there is no needs for a single server to store routing tables of all tenants. We can improve the scalability of the system while retaining the benefits of the stateless load balance.

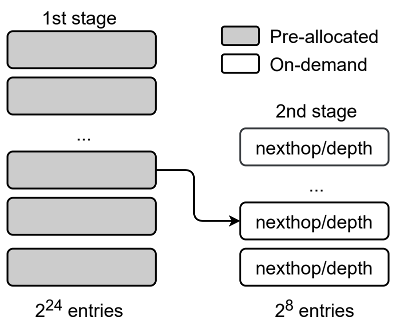
4.1.2 Fast datapath route lookup

We have leveraged the DPDK suites [12] to develop the high performance dataplane forwarding module. As illustrated in Figure 6, the arrival packets are forwarded directly from the NIC to user space bypassing kernel overhead. To efficiently utilize the modern multi-core CPU architecture, traffic need to be balanced to multiple cores, we apply the NIC built-in RSS functionality to uniformly distribute the traffic to different dispatchers which run on dedicated CPU cores. To avoid packet out-of-order, packets belong to the same overlay flow should be processed on the same core. To achieve that, the dispatchers decapsulate packets' outer tunnels and distribute packets to different forwarding threads based on the overlay 5-tuple. The forwarding threads encapsulate packets with another tunnel according to packets' destination. Packets are then forwarded to external WAN network. We plan to accelerate this procedure with advanced NICs which support overlay packet header hashing.

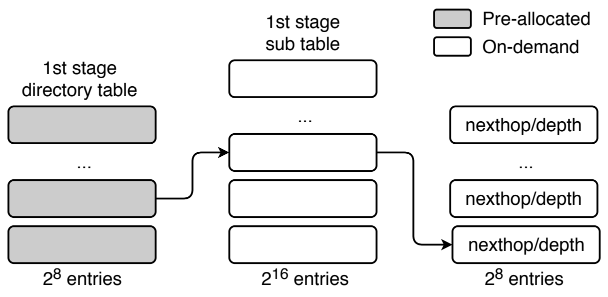
Short packet processing pipeline. In high performance packet processing, a long pipeline will likely lead to more cache misses and causes performance degradation. The packet processing cost mainly stems from the LPM lookups. Generally, a packet would require two LPM lookups, *i.e.* the packet first does LPM lookup for underlay encapsulation header based on the overlay IP, then the encapsulated packet does LPM look up for the output physical port based on the underlay IP. We have shortened this pipeline by saving the second LPM lookup. To achieve that, we combine the second LPM lookup results *i.e.* the output physical port and the the first LPM lookup results *i.e.* the encapsulation header into one unified action. The action is pre-programmed into the action field of the first LPM lookup entry. After the first LPM lookup, the packet is encapsulated and forwarded to the corresponding physical port following the pre-programmed rules. With the shorter pipeline, the packet processing performance is largely improved.

Fast route lookup at large scale. When dealing with 10M routing entries, we need to take care of the storage and the lookup speed. At first, we leverage the LPM library in DPDK suites [1] to implement the forwarding function. Unfortunately, the original DPDK LPM library is not efficient to store routing entries. We encountered performance issues with the increasing amount of tenants.

To illustrate the problem, we first briefly introduce how the DPDK LPM library works. The library uses a classic trie-tree structure to store the routing entries. As shown in Figure 7(a), for IPv4 lookup, it uses two stages. The first stage is an array with 2^{24} entries. The higher 24-bit of an IPv4 address is used as the array index. Each entry stores the base address of a secondary stage array. The lower 8-bit of an IPv4 address acts as the index into the secondary stage array. Based on the address in the first array, a routing result can be identified. Each IPv4 lookup requires at most two memory lookups in this design. However, to achieve this, DPDK LPM library needs to pre-allocate a large memory to store the entire 2^{24} entries no matter whether there exists an IP route or not. This leads to severe memory waste and limits the ability to support a large number of VRFs. For example, to store 64K IPv4 routes, the first stage would require $2^{24} \cdot 4B = 64MB$ memory for each VRF. In the worst case, to store 64K routes in the second stage, it requires $64K \cdot 2^8 \cdot 4B = 64MB$, *i.e.* 128MB



(a) DPDK LPM table design



(b) Optimized LPM table design

Figure 7: LPM table data structure

memory in total. To support 10K VRFs, 1.28TB memory is required. Moreover, the first stage requires a large continuous memory, the memory fragmentation makes it hard to fulfill the need especially when the system has been up for a long time.

The root cause of this problem is the static memory allocation mechanism. In fact, only few tenants would require a large routing table. Most of our customers has a small routing table. Based on this observation, we design a dynamic memory allocation mechanism for LPM tables (Figure 7(b)).

First, we turn the DPDK LPM first stage array into one directory table (2^8 entries) and many sub-tables (each with 2^{16} entries). Only the directory table is pre-allocated. Thus the minimum memory required for a VRF is only $2^8 \cdot 8B = 2KB$. With the same amount of memory, the number of VRFs we can support increases by 3 orders of magnitude. Furthermore, with the same amount of memory, the maximum table size that can be used by a VRF is also enlarged.

Second, in the original DPDK LPM design, the memory of both the first and the second stages are pre-allocated. In contrast, in our design, the first stage sub-tables and the seconds stage tables are both allocated on demand. This significantly improves the memory efficiency. However, this design leads to problem when considering the route modification and deletion. When a route is deleted, we can not free the memory taken by this route because there may be arrival packets at the same time. A lock mechanism can help solve the problem but with a large overhead. Instead, we have designed an user-space RCU (Read-Copy Update) mechanism to solve this problem without using any lock. To be specific, when updating a forwarding entry, we first do a copy of the original data and make changes on the new copy. Then we modify the corresponding pointer to point at the new copy. The old entry is deleted only when there is no other threads accessing it. We implemented the RCU as a Quiescent-State-Based Reclamation (QSBR) flavor one [20]. Our evaluation shows that we can provide high performance packet processing and high route updates speed at 64K entries/s simultaneously.

Hardware acceleration With the rapid growth of traffic and slowing down of Moore's law, the CPU becomes the bottleneck for packet processing in software-based routing modules. To address this problem, prior studies have applied the cost-efficient hardware acceleration technique [16,21,28,29]. With regard to the limited forwarding table, *e.g.* Barefoot Tofino chips have ∼60MB on-chip memory [6, 16, 28, 32], which can only support 100K of LPM entries according to our evaluation, we have designed a large flow offloading scheme for the forwarding module. Based on the historical information of the cloud traffic, we found that the top 1% largest flows contribute 70% of overall traffic. We maintain a small group of prefixes in switch hardware for those large flows and forwards them with the hardware. By pushing the flexible and complex logic to software while leveraging the stable hardware offloading features, this approach saves more than 50% CPUs and greatly reduces the cost[1].

4.2 Highly Scalable Routing Module

Traditionally, in commodity router based PCS, standard MPLS-VPN technique [42] is used (§ 2). To differentiate BGP routes of different tenants, BGP messages from each tenant is configured to export m distinct BGP Routing Target (RT) attributes. At the cloud side, different tenants are separated via VRFs, say n VRFs. A tenant VRF only learns BGP routes from the corresponding remote on-premise datacenter, which is achieved by importing certain RTs. The process of route insertion is rather slow for traditional routers. Once receiving a BGP route update, the corresponding VRF is located by comparing each RT attribute with all VRF's RT values. m RTs (m is around 10) and n VRFs (n is around 10K) result in an insertion complexity of $O(mn)$. The performance can not sustain highly frequent BGP routes insertion.

In DSR, with the help of the connect module switch, BGP import/export RT features are no longer required. Specifically, DSR leverages the connect module switch to tag routes from different tenants based on VXLAN tunnels. The VNI in the VXLAN tunnel is pre-determined for each tenant. Then the BGP module can insert the route to the corresponding VRF using the VNI number. This design reduces the route insertion complexity to $O(n)$.

Some PCS customers have multiple datacenters. The newly built datacenters use the latest DSR based PCS while others

[1]The detail of the implementation is beyond the scope of this paper.

may still use commodity router based PCS. For availability purpose, the routing module of DSR needs to learn the routes to certain datacenters from commodity routers. This induces a requirement of inserting BGP routes into VRFs based on the RT attributes. As aforementioned, this would take $O(mn)$ time complexity for m RT attributes and n VRFs, leading to scalability issue.

To optimize the BGP insertion performance in this scenario, we use a hash function based solution to speed up the process of VRFs lookup. The key idea is that we store the RT-to-VRF information in a hash table where the RT value serves as the key. The value of each item is organized as a linked list of VRFs based on the RT. With this optimization, the capability of route insertion is significantly improved.

The routing plane needs to exchange keep-alive messages with a huge number of BGP neighbors, *e.g.* 10K. Traditionally, the keep-alive function is implemented in the same thread with the routing message processing function. When the thread is heavily loaded with routing updates, the keep-alive messages can not be processed in real time, leading to BGP timeout at remote side. We separate the keep-alive message handling and BGP message processing into different threads to solve the problem. As a result, we can maintain stable BGP neighbor relations even under the most heavy BGP message processing workload.

In summary, with all the above optimizations, DSR can improve the routing module BGP performance by ~20 times. The BGP module can perform 500K routes insertions per second and support 10Ks of VRFs (see § 7.2 for more details).

5 Reliability

We have deliberately designed several techniques to enhance the system reliability.

5.1 Forwarding Path Failure Detection

Conventionally, when peering with commodity routers, we enable BFD protocol [23] to provide fast and consistent forwarding path failure detection. However, with the disaggregated design, components of DSR are deployed in different availability zones. There are multiple paths available between the peering devices and the routing module. The BFD messages will transmit along only one single path. If one link of the path failed, the BFD message will be lost. Then the BGP module would regard that the remote peer is not reachable and delete the corresponding routes for fast convergence. In fact, because there are many paths available from the access module to the routing module, the data path is not disconnected.

To address this problem, we should distribute the BFD messages into multiple paths. We have modified the hash function of ECMP in the access module. In detail, for an arriving BFD packet, instead of using the fixed five-tuple of the message to generate the source UDP port value in the VXLAN header, we apply the IP ID field to compute the ECMP hashing. The hash key is then encoded into the UDP source port of the VXLAN header. As a result, BFD packets will traverse different paths along the network. We apply the similar idea at the forwarding module and routing module for BFD ECHO packets in the reverse route. When a link fails, there are still enough BFD packets available to report the healthy state of the remote peer.

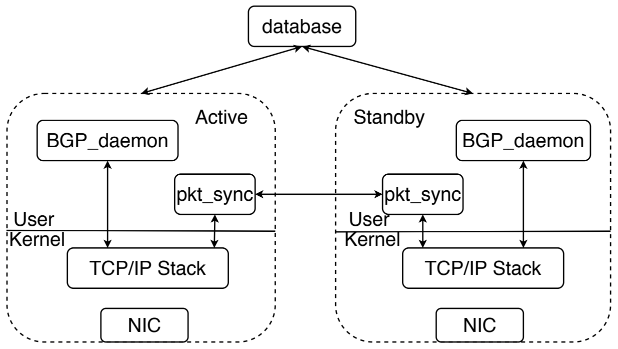


Figure 8: The NSR mechanism of the BGP module

5.2 BGP Non-Stop Routing

We developed a Non-Stop Routing (NSR) [42] mechanism against single BGP module failure. NSR mechanism requires that the failure of the BGP speaker to be transparent to the peering devices. This allows the failed BGP speaker immediately switch-over to a backup BGP speaker, and the backup device have all the information required to take over.

Traditionally, the commodity routers use a backup processor to implement the NSR functionality, as shown in Figure 3. They customize the protocol processors in the Linux kernel stack to enable this feature. However, in our case, the cloud resource manager requires that the kernel software should be homogeneous for all cloud servers to avoid operational issues.

We come up with a solution without kernel modification. As shown in Figure 8, the key idea behind our NSR design is to make the active BGP module synchronise its TCP states, *e.g.* sequence number and window size, with the standby BGP module before the active BGP speaker sends acknowledgements to the remote peer. We introduce a reliable database in the control plane to synchronize the neighbors information. Once the backup BGP speaker receives the TCP states, it initiates a socket system-call with TCP repair option, which keeps the TCP states in correct accordance without sending or receiving any TCP packet. Thus the standby module can have the same TCP states with its active counterpart.

Moreover, during BGP software upgrade, we need to make sure the data path traffic is not affected. This is accomplished by employing the NSR mechanism to proactively stop the active one and hand over BGP processing to the standby module. When the software upgrade is done, the active module is restored. It is notable that this process allows us to scale out/in the routing module easily.

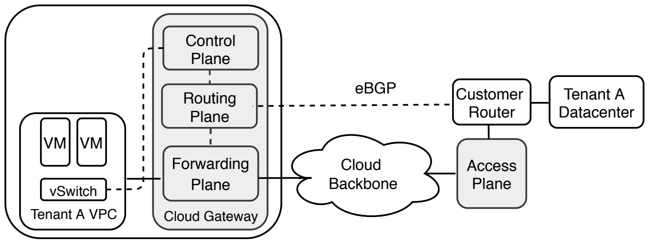


Figure 9: DSR based private line service

6 Fast Development of New Features

Public cloud users mainly rely on their providers to provide an elastic, reliable, scalable and safe networking environment. Customers have diverse requirements in terms of latency, bandwidth, self-defined protocol, etc. DSR is proposed to achieve fast feature velocity to satisfy customers' requirements. Through a few examples we demonstrate the benefits of using DSR for the operation of cloud network.

DSR based PCS gateway. We revisit the example introduced in § 2.1 and show that a more scalable and elastic PCS can be built with DSR compared with previous commodity router based design. As shown in Figure 9, once the tenant network is connected to the cloud network via PCS, the customer router can directly set up BGP sessions with the routing module of the cloud gateway. The routing module announces the routes of tenant on-premise datacenters to its VPC network through the control module. Similarly through the BGP sessions, the tenant on-premise datacenter can learn the VPC routes and VM routes from the routing module. We are able to provide real-time routing updates between the cloud SDN network and the external BGP network.

Customizing egress routing. A PCS customer may set up multiple PCS channels with the cloud through multiple DSR instances deployed at different sites. We can provide network quality aware traffic steering for our customers in this scenario. In detail, beside propagating the peering routes to the control module, DSR measures and informs the control module the quality of each path, including bandwidth, delay and packet loss rate. With both the routing information and network quality information, the control module can choose the optimal egress route for the traffic. Moreover, customers can customize their own routing policy to choose a specific path for certain traffic.

Fast failure recovery. Beside a PCS channel, the on-premise datacenter can configure another IPSec VPN channel over Internet as a backup. By default, traffic are forwarded through the PCS channel. When network failure is detected, *e.g.* a broken cable, the BGP module quickly reports the failure to the control module and withdraws the route. The control module then modifies the next hop of the routing on the forwarding module to the IPSec channel. The whole procedure can be done within 10ms, which is two orders of magnitude lower than traditional solution with commodity routers which requires periodically checking the routing updates from the commodity routers through the slow NETCONF interface.

On-premise to on-premise datacenters traffic acceleration. Through PCS, customers can transfer data among multiple on-premise datacenters via a hub-spoke model, where the cloud gateway acts as the hub. Previously, this is enabled by the route reflector, which reflects BGP routes among multiple cloud access routers. The scalability to support a large amount of customers are limited by the route reflector's BGP processing ability. With the home-made BGP module, the constraint is relaxed as the routing module can be horizontally scaled.

Protection against DDoS attack. DDoS traffic are short-bursts with high volume. When DDoS traffic are identified by the DDoS detection engine, those traffic need to be redirected to a DDoS traffic cleaning center. Notice that DSR is built on commodity servers, it can be potentially affected by DDoS traffic. We aim to redirect the suspicious DDoS traffic at the access module before they come to the software-based forwarding module. Unfortunately, the attack traffic are identified based on layer 3 routing information, but the current access switch works as a layer 2 switch for simplicity. To enable layer 3 packets forwarding for the DDoS traffic only, we introduce an ARP-spoofing scheme. To be specific, we first install the suspicious DDoS traffic routes as advertised by the DDoS detection engine in the access switch. And whenever an ARP request message of the DDoS traffic arrives, DSR can send an ARP reply with the MAC of the access module instead of the MAC of the DSR routing module. By so doing, all the inbound DDoS traffic will be encoded with the MAC of the access module as the destination MAC. Then the access switch can work as a layer 3 switch, and forwards suspicious DDoS traffic based on layer 3 routing table. Notice that as we only do ARP spoofing for DDoS traffic, the normal inbound traffic and outbound traffic are still directly forwarded to the forwarding module of DSR.

7 Evaluation

We demonstrate the capability of packet processing and BGP convergence time of DSR. The servers we used for evaluation are Dell PowerEdge R740 with 192GB memory and 80 cores (Intel(R) Xeon(R) Gold 6133 CPU@2.50GHz with hyper threading enabled). Each server is equipped with a dual 40Gbps port NIC.

7.1 Forwarding Module Evaluation

The host-based DSR forwarding module is able to process packets at line rate with low latency. In the experiment, we generate VXLAN packets and the forwarding module decapsulates a total of 68B from the header of the arrival packet.

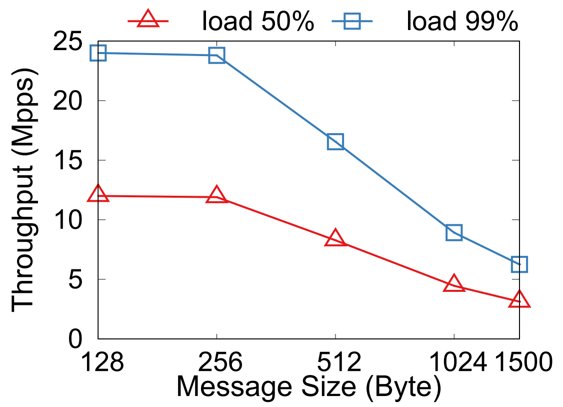


Figure 10: Dataplane throughput under different loads

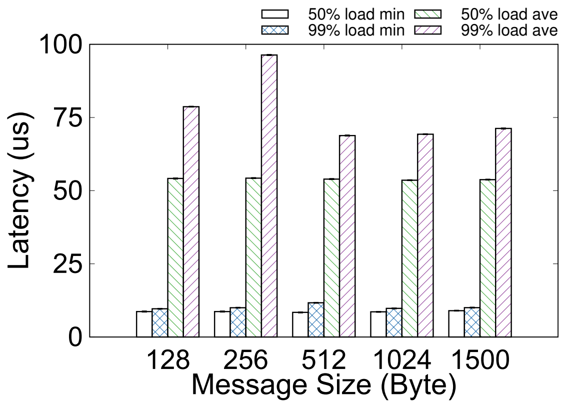


Figure 11: Dataplane latency under different loads

To maintain low latency and high throughput, we have introduced many software optimization techniques in the forwarding plane, *e.g.* kernel bypassing, minimizing cache miss and short pipeline. We measure the CPU consumption of the forwarding plane under moderate 50% and high 99% workloads with various packet sizes ranging from 128B to 1500B. For the load of 50% workload, the generated traffic consume 50% of the CPU while for 99% workload, the traffic consume 99% of the CPU. We can effectively utilize the CPU resource to provide high rate packet processing.

Dataplane throughput under different workloads. As shown in Figure 10, at 99% load, the datapath can process traffic at high rate under different packet sizes ranging from 128B to 1500B. It can achieve 24Mpps for packets of 128B. The throughput decreases proportionally with the workloads which shows graceful scaling property. The good performance stems from our optimizations, *i.e.* short pipeline design and FIB lookup optimization, as introduced in § 3.

Dataplane latency under different workloads. We measure the per-packet processing latency of DSR. As shown in Figure 11, the data plane can provide low and stable latency of 50-70μs for both 50% and 99% workloads. The minimum latency identifies the packet processing delay without ring-buffer queueing delay. It is 10μs. For average latency, it is stable across different packet sizes. With the increase of workloads, the software ring-buffer queue size also slightly increases which leads to a moderate increase in latency.

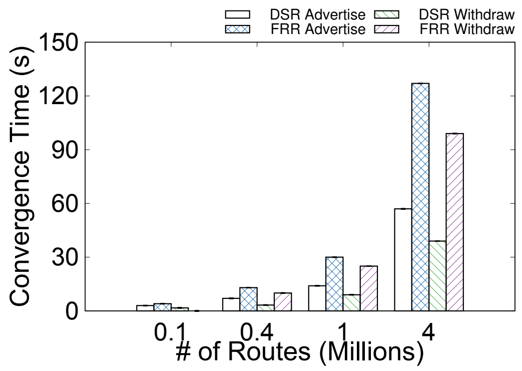


Figure 12: DSR BGP performance compared with FRR

Dataplane throughput during FIB update. To demonstrate the effectiveness of the datapath lockless design, we compare datapath throughput with and without FIB updates. The FIB update rate is 64Kbps. We generate VXLAN traffic for 30s and records the packet processing rate. The throughput can achieve 20Mpps with or without FIB updates, *i.e.* supporting lin-rate packet processing. This demonstrates that the lock-free design enables fast routing updates without impacting the datapath forwarding performance. We omit the figure due to space limit.

7.2 Routing Module Evaluation

We compare the routing module with the open source FRR [3] solution under 300 RTs, and demonstrate the performance of our home-made BGP speaker under a large number of VRFs and neighbors.

BGP advertising and withdrawal convergence time is the key metric to be reported. As shown in Figure 12, the DSR routing module outperforms the FRR under different number of routes, *e.g.* 0.1M, 0.4M, 1M and 4M. For BGP advertisement, when there are 0.1M routes, the convergence time of DSR is 33.3% lower than FRR. When the number of routes is 4M, DSR converges ~2.2× faster than FRR. For BGP withdraw, DSR has 1.5-2× lower convergence time than FRR. The performance gain is achieved based on the optimization introduced in § 4.2 which reduces the complexity for route insertion from $O(mn)$ to $O(n)$.

We then evaluate the performance of DSR routing module when there are 1K VRFs and each VRF has 8 neighbors. It is notable that the traditional commodity router deployed in our system can not support 1K VRFs. We record the time to advertise, withdraw and report different amounts of routes to the controller. As shown in Figure 13, the time increases moderately when the amount of routes increases, which demonstrates excellent scalability of the home-made BGP speaker. This performance gain mainly stems from the BGP optimization techniques introduced in § 4.2.

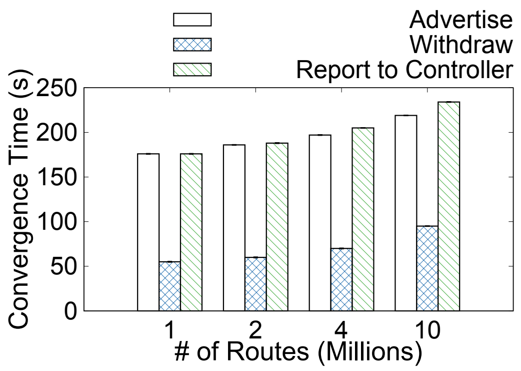
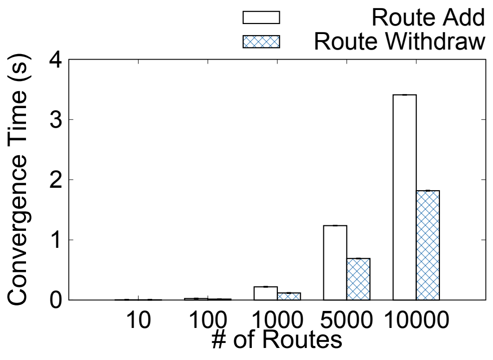
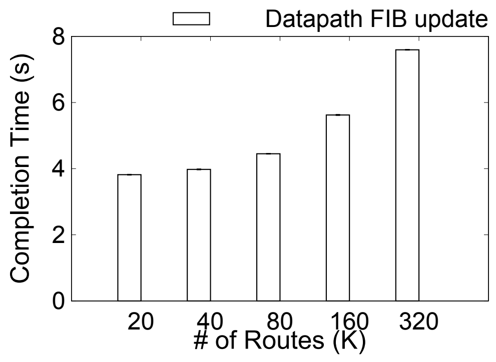


Figure 13: BGP performanec under many VRFs



(a) Performance for handling reported BGP routes



(b) Performance for routing configuration to data path

Figure 14: SDN controller performance for BGP processing

7.3 Control Module Evaluation

The control module is responsible for computing FIB based on the routes collected from the routing module and installs the FIB entries to the datapath forwarding plane. We evaluate the effectiveness of control module for BGP routing configuration. As shown in Figure 14(a), it takes less than 200ms to add or withdraw 1K routes. For adding and withdrawing 10K routes, it takes about 3.4 seconds and 1.8 seconds respectively. More VRFs will add a little more overhead during the VRF insertion process. We then evaluate the control module's performance for updating the FIB entries of the underlying forwarding module. As shown in Figure 14(b), it takes ∼7.5 seconds to configure 320K routes, which meets our needs in most cases.

8 Operational Experiences

The DSR system has been in production for over three years and deployed at over 30 sites replacing the commodity routers there. We has delivered tens of new features with a release cycles of 2∼4 weeks. In contrast, the typical commodity router based solution releases new feature in several months or years. As a result, DSR can save an order of magnitude of monetary costs compared with the traditional solution. Now we introduce the efforts we have taken to operate DSR.

8.1 vNetVerifier System

DSR consists of multiple components built on x86 servers. During the operation, we found that x86 servers are not as reliable as commodity router. Despite the techniques we have used to improve reliability, we need a health monitoring system to discover and locate the failure. We have proposed a virtual network verification system, *i.e.* vNetVerifier, to solve the problem. We discuss some useful features of the system.

Transparent monitoring plane. We aim to achieve active monitoring without interfering with the customer traffic. For this purpose, we have created multiple virtual testing tenants and added monitoring rules in corresponding components. They act like real tenants using the same control and data paths as normal cloud tenants. We generate probing traffic continuously to monitor system faults.

Fine-grained probing. We carry out fine-grained datapath probing at single server granularity and single processing core granularity. As the datapath servers of the same forwarding fleets share the same Virtual IP using ECMP hashing, to monitor a targeted server, we craft probing packets with the underlay IP of the targeted server as the destination IP. For single core monitoring, we encode the processing core ID into the probing packets. Then the dispatchers on the datapath server examine the core ID and forward the packet to the targeted processing core. In this way, we achieved fine grained probing at server-level and core-level. As the decoupled components of DSR are connected via multiple paths. A single-path failure may lead to a wrong reaction as discussed in § 5.1. As a result, we intentionally eliminate the effect of single-path failure through multi-path probing similar to the operation introduced in § 5.1.

Online testing. Conventionally, before we update network components, *e.g.* software upgrade or hardware replacement, we test the components intensively in the small-scale testing environment. However, it is hard to mimic the realistic environment to find all the problems. To this end, we carry out online testing by leveraging the testing tenant in our production environment. To be specific, we convert different testing cases into corresponding network configurations and customized probing packets. Then we carry out online testing before and after the network updates. When abnormal traffic are detected, the ongoing network upgrading will be stopped and reverted. We are also investigating at high fidelity network emulation approaches similar as CrystalNet [26].

8.2 Fault Isolation and Localization

Fault domain isolation. An important principle during our operation is to guarantee that the datapath forwards packets correctly and continuously even when the control module and routing module fails. To achieve that, we need to do fault domain isolation for different components. A fault domain is defined as the "blast range" of a certain system failure, *e.g.* the failure of one BGP module node may affect the routing decision of a datapath forwarding node which may drop user traffic. The disaggregated design and deployment of control module, routing module and forwarding module makes it possible to isolate different fault domains. Since the control module stores all the routing information in a persistent database, if the BGP modules fail, the datapath can forwards traffic using the control module's routing information.

BGP events and traffic paths analysis. When fault happens, we need sufficient information to identify the root cause. The BGP events and traffic paths are two most important types of information. We have recorded all the BGP events in the database for post-mortem analysis. Since traffic paths can be dynamically changed, we use traceroute like tools to identify the path.

9 Related Work

To overcome the limitation of the traditional BGP protocol, Facebook [35] and Google [41] have developed SDN-based systems to control egress traffic routing for peering edges. Facebook EDGE FABRIC relies on vendor routers, which overrides the routers' normal BGP selection in each individual Points of Presence (PoP). Google Espresso removes the need for commodity BGP routers by employing centralized traffic engineering, which selects egress at distant PoPs. It is notable that DSR faces more complicated scenarios, including not only the peering routing for Internet users, which is the focus of EDGE FABRIC and Espresso, but also inter-connection between public cloud and customers' on-premise datacenters as well as customers' branch offices. Though we share the same idea with Espresso that separating the logic of the control plane from the data plane, DSR targets a different set of challenges on scalability, flexibility and reliability. In order to satisfy the requirements of diverse cloud access scenarios, the proposed disaggregated software router architecture is a general platform that is capable of customizing and scaling each module independently. DSR meets more stringent scalability requirements to support 10Ks VRF tables for PCS and 10M FIB entries for EIS. Therefore, we demonstrated the optimization for scaling the forwarding and routing plane, which is not addressed in Espresso. For flexibility, Espresso introduce the application-aware routing to serve Internet users. In contrast, DSR focuses on fast feature delivery to satisfy the demands of cloud customers, like self-defined routing policy and GRE encapsulation, etc. Furthermore, we solved multiple reliability issues when introducing the disaggregated software-defined router whose components are implemented in a distributed environment, which is not addressed in Espresso either.

NFV technologies are widely used to replace traditional proprietary middle boxes and switching devices. NFV technique reduces cost and provides high feature velocity [17, 24, 25, 30–34, 36, 40]. Previous works focus on several different aspects of the NFV, *e.g.* elastic scaling [22, 34, 40], NFV management [30, 39], performance optimization and implementation [9, 30], etc. DSR is an integration of multiple NFV techniques at cloud gateway. DSR applies DPDK [12] and VPP [8, 14] for the high performance data module. For the BGP speaker, we have evaluated the open source candidates like Quagga [2], GoBGP [4], BIRD [15] and FRR [3]. We developed our high performance BGP speaker based on FRR, which supports multiple functions like IPv6, BFD, etc. For IPSec gateway in the SWS scenario, both the control and data path of the IPSec protocol are coupled on the forwarding fleet. We plan to investigate similar approach as in [38] to separate the control and data plane of the IPSec protocol in order to achieve high scalability.

10 Conclusion

Cloud gateways play a significant role for enterprise customers and Internet users to access the resources in the cloud. With the rapid growth of users, we introduce DSR to replace the commodity routers based cloud gateways in Tencent Cloud. It employs a disaggregated software-defined architecture which provides high scalability and accelerates the features rollout velocity. Meanwhile, integration with the SDN technique achieves smart traffic steering and fast failure convergence. DSR has been deployed in production for over 3 years and has served tens of Tbps traffic for our customers.

Acknowledgments

The authors thank the anonymous reviewers and our shepherd, Nandita Dukkipati, for providing valuable feedback. We thank the teams at Tencent for their contributions to the work, including but not limited to, Pan Feng, Quan He, Haiyang Tan, Qingyan Yu, Bo Wei, Changling Tang, Da Hu, Jianchao Lv, Jiangbo Wang, Jiexuan Cui, Kejin Zuo, Lingyu Zhang, Mengfan Dai, Ruiqiang Dai, Xiaolong Si, Yanxiong Chen, Zhengbin Wang, Zhengzhong Yu, Zhi Tan, Zhichao Hou, Zhifang Wu, Zhiyi Qiu, Zicheng Yuan and Yachen Wang.

References

[1] DPDK documentation, LPM library. https://doc.dpdk.org/guides/prog_guide/lpm_lib.html.

[2] GNU Quagga Project. http://www.nongnu.org/quagga/, 2010.

[3] FRRouting (FRR). https://frrouting.org, 2017.

[4] GoBGP: BGP implementation in Go. https://github.com/osrg/gobgp, 2019.

[5] Cisco ASR 9000 series aggregation services routers, https://www.cisco.com/c/en/us/products/routers/asr-9000-series-aggregation-services-routers/index.html, Mar 2020.

[6] Trident4 / BCM56880 series, high-capacity strataxgs trident 4 Ethernet switch series, 2020.

[7] Loa Andersson and George Swallow. RFC3468: The multiprotocol label switching (MPLS) working group decision on MPLS signaling protocols, 2003.

[8] David Richard Barach and Eliot Dresselhaus. Vectorized software packet forwarding, June 14 2011. US Patent 7,961,636.

[9] Anat Bremler-Barr, Yotam Harchol, and David Hay. OpenBox: a software-defined framework for developing, deploying, and managing network functions. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 511–524, 2016.

[10] CISCO. IPSLA. http://www.cisco.com/en/US/products/ps6602/products_ios_protocol_group_home.html.

[11] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, et al. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 373–387, 2018.

[12] Intel DPDK. Data plane development kit. https://www.dpdk.org/, 2014.

[13] Dino Farinacci, Tony Li, Stan Hanks, David Meyer, and Paul Traina. Generic routing encapsulation (GRE). *RFC*, 2784, 2000.

[14] FD.io. VPP. https://wiki.fd.io/view/VPP.

[15] Ondrej Filip, L Forst, P Machek, M Mares, and O Zajicek. BIRD internet routing daemon. *NANOG-48, Austin, TX*, 2010.

[16] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure accelerated networking: Smartnics in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 51–66, 2018.

[17] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. OpenNF: Enabling innovation in network function control. *ACM SIGCOMM Computer Communication Review*, 44(4):163–174, 2014.

[18] 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 *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, SIGCOMM '09, page 51–62, New York, NY, USA, 2009. Association for Computing Machinery.

[19] Stan Hanks, David Meyer, Dino Farinacci, and Paul Traina. Generic routing encapsulation (gre). 2000.

[20] Thomas E Hart, Paul E McKenney, Angela Demke Brown, and Jonathan Walpole. Performance of memory reclamation for lockless synchronization. *Journal of Parallel and Distributed Computing*, 67(12):1270–1285, 2007.

[21] Stephen Ibanez, Gordon Brebner, Nick McKeown, and Noa Zilberman. The P4->NetFPGA workflow for line-rate packet processing. In *Proceedings of ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 1–9, 2019.

[22] Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. Stateless network functions: Breaking the tight coupling of state and processing. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 97–112, 2017.

[23] Dave Katz, Dave Ward, et al. Bidirectional forwarding detection (BFD). RFC 5880, June, 2010.

[24] Sameer G Kulkarni, Wei Zhang, Jinho Hwang, Shriram Rajagopalan, KK Ramakrishnan, Timothy Wood, Mayutan Arumaithurai, and Xiaoming Fu. Nfvnice: Dynamic backpressure and scheduling for nfv service chains. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 71–84, 2017.

[25] Bojie Li, Kun Tan, Layong Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. ClickNP: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 1–14, 2016.

[26] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. Crystalnet: Faithfully emulating large production networks. In *Proceedings of Symposium on Operating Systems Principles (SOSP)*, pages 599–613, 2017.

[27] Mallik Mahalingam, Dinesh G Dutt, Kenneth Duda, Puneet Agarwal, Lawrence Kreeger, T Sridhar, Mike Bursell, and Chris Wright. Virtual extensible local area network (VXLAN): A framework for overlaying virtualized layer 2 networks over layer 3 networks. *RFC*, 7348:1–22, 2014.

[28] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching ASICs. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 15–28, 2017.

[29] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. Stateless datacenter load-balancing with beamer. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 125–139, 2018.

[30] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: a framework for nfv applications. In *Proceedings of Symposium on Operating Systems Principles (SOSP)*, pages 121–136, 2015.

[31] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. Netbricks: Taking the v out of NFV. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 203–216, 2016.

[32] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The design and implementation of Open vSwitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 117–130, 2015.

[33] Shriram Rajagopalan, Dan Williams, and Hani Jamjoom. Pico replication: A high availability framework for middleboxes. In *Proceedings of Annual Symposium on Cloud Computing*, pages 1–15, 2013.

[34] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. Split/merge: System support for elastic execution in virtual middleboxes. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 227–240, 2013.

[35] Brandon Schlinker, Hyojeong Kim, Timothy Cui, Ethan Katz-Bassett, Harsha V Madhyastha, Italo Cunha, James Quinn, Saif Hasan, Petr Lapukhov, and Hongyi Zeng. Engineering egress with edge fabric: Steering oceans of content to the world. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 418–431. ACM, 2017.

[36] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K Reiter, and Guangyu Shi. Design and implementation of a consolidated middlebox architecture. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 323–336, 2012.

[37] Upendra Bhalchandra Shevade, Kyle Benjamin Schultheiss, and Gregory Rustin Rogers. Scalable routing service, June 4 2019. US Patent App. 14/274,534.

[38] Jeongseok Son, Yongqiang Xiong, Kun Tan, Paul Wang, Ze Gan, and Sue Moon. Protego: Cloud-scale multitenant ipsec gateway. In *USENIX Annual Technical Conference (ATC)*, pages 473–485, 2017.

[39] Chen Sun, Jun Bi, Zhilong Zheng, Heng Yu, and Hongxin Hu. Nfp: Enabling network function parallelism in nfv. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 43–56, 2017.

[40] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. Elastic scaling of stateful network functions. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 299–312, 2018.

[41] Kok-Kiong Yap, Murtaza Motiwala, Jeremy Rahe, Steve Padgett, Matthew Holliman, Gary Baldus, Marcus Hines, Taeeun Kim, Ashok Narayanan, Ankur Jain, et al. Taking the edge off with espresso: Scale, reliability and programmability for global internet peering. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 432–445. ACM, 2017.

[42] Randy Zhang and Micah Bartell. *BGP design and implementation*. Cisco Press, 2003.

CodedBulk: Inter-Datacenter Bulk Transfers using Network Coding

Shih-Hao Tseng
Cornell University

Saksham Agarwal
Cornell University

Rachit Agarwal
Cornell University

Hitesh Ballani
Microsoft Research

Ao Tang
Cornell University

Abstract

This paper presents CodedBulk, a system for high-throughput inter-datacenter bulk transfers. At its core, CodedBulk uses network coding, a technique from the coding theory community, that guarantees optimal throughput for individual bulk transfers. Prior attempts to using network coding in wired networks have faced several pragmatic and fundamental barriers. CodedBulk resolves these barriers by exploiting the unique properties of inter-datacenter networks, and by using a custom-designed hop-by-hop flow control mechanism that enables efficient realization of network coding atop existing transport protocols. An end-to-end CodedBulk implementation running on a geo-distributed inter-datacenter network improves bulk transfer throughput by $1.2-2.5\times$ compared to state-of-the-art mechanisms that do not use network coding.

1 Introduction

Inter-datacenter wide-area network (WAN) traffic is estimated to quadruple over the next five years, growing $1.3\times$ faster than global WAN traffic [2]. In the past, such an increase in demand has been matched by physical-layer technological advancements that allowed more data to be pumped on top of WAN fibers, thus increasing the capacity of existing inter-datacenter links [14]. However, we are now approaching the fundamental non-linear Shannon limit on the number of bits/Hz that can be practically carried across fibers [14]. This leaves datacenter network providers with the expensive and painfully slow proposition of deploying new fibers in order to keep up with increasing demands.

Several studies have reported that inter-datacenter traffic is dominated by geo-replication of large files (*e.g.*, videos, databases, etc.) for fault tolerance, availability, and improved latency to the user [3, 6, 8, 23, 25, 26, 31, 39, 40, 47, 49]. We thus revisit the classical multicast question: given a fixed network topology, what is the most throughput-efficient mechanism for transferring data from a source to multiple destinations? The answer to this question is rooted in network coding, a technique from the coding theory community that generalizes the classical max-flow min-cut theorem to the case of (multicast) bulk transfers [9, 30, 33]. Network coding guarantees optimal throughput for individual bulk transfers by using in-network computations [9, 24, 30, 33]; in contrast, achieving optimal throughput using mechanisms that do not perform in-network computations is a long-standing open problem [34]. We provide a primer on network coding in §2.2. While network coding has been successful applied in wireless networks [28, 29], its applications to wired networks have faced several pragmatic and fundamental barriers.

On the practical front, there are three challenges. First, network coding requires network routers to buffer and to perform computations on data, which requires storage and computation resources. Second, computing "network codes" that define computations at routers not only requires a priori knowledge about the network topology and individual transfers, but also does not scale to networks with millions of routers and/or links. Finally, network coding requires a single entity controlling the end-hosts as well as the network.

In traditional ISP networks, these challenges proved to be insurmountable but the equation is quite different for inter-datacenter WANs. The structure of typical inter-datacenter WAN topologies means that, instead of coding at all routers in the network, coding can be done only at resource-rich datacenters—either at border routers or on servers inside the datacenter—without any reduction in coding gains (§2.1). Inter-datacenter WAN operators already deploy custom border routers, so increase in computation and storage resources at these routers to achieve higher throughput using the available WAN bandwidth (that is expensive and increasingly hard-to-scale) is a good trade-off to make. The second and third challenges are also alleviated by unique characteristics [23, 25] of inter-datacenter WANs: (1) network sizes limited to hundreds of routers and links enables efficient computation of network codes and implementation of network coding; (2) SDN-enabled routers combined with the fact that transfers are known a-priori [39, 40] allow for centralized code computations which can, in turn, be programmed into routers; and (3) a single entity controlling end-hosts as well as the network.

While inter-datacenter WAN features lower the pragmatic barriers, one fundamental challenge still needs to be resolved. Traditional network coding assumes that there is no other traffic in the network, either foreground (*e.g.*, interactive traffic) or background (*e.g.*, other bulk transfers). More generally, network coding assumes that all links have the same latency and bandwidth, and that link latencies and bandwidths remain static over time. This assumption does not hold in practice, *e.g.*, due to sporadic high-priority interactive traffic, and due to multiple concurrent bulk transfers atop inter-datacenter WANs. We refer to this as the *asymmetric link* problem.

CodedBulk is an end-to-end system for high-throughput inter-datacenter bulk transfers that resolves the asymmetric link problem using a simple Hop-by-hop Flow Control (HFC) mechanism. The core idea in HFC mechanisms is to partition available buffer space at routers among active flows, so as to avoid buffer overflow [41, 45]. HFC mechanisms have been explored for traditional non-coded traffic [37, 41, 42, 51]; however, using HFC mechanisms for network coding imposes an additional constraint: all flows that need to be coded at any router must converge to the *same* rate. Simultaneously, routers need to work with limited storage and compute resources in the data plane. We show that *any* buffer partitioning scheme that assigns non-zero buffers to each (coded) flow achieves the following desirable properties: (i) for each individual bulk transfer, all incoming flows that need to be coded at a router converge to the same rate; (ii) for all bulk transfers sharing a link, the rate for all their flows through the link converge to the max-min fair rate; and (iii) the network is deadlock-free.

The use of network coding, coupled with HFC, also means that flow control at a router is correlated across multiple flows. While this could be implemented via modifications in the network stack, we introduce a *virtual link* abstraction which enables CodedBulk without any modifications to existing flow control-enabled transport layer and multipath-enabled network layer implementations. For instance, CodedBulk currently runs on top of unmodified TCP and MPLS-enabled network layer mechanisms supported by existing inter-datacenter WANs. Our implementation requires no special traffic shaping mechanism, allows co-existence of high-priority interactive traffic, and handles failures transparently.

We envision two deployment scenarios for CodedBulk. First, an infrastructure provider can provide CodedBulk as a service to geo-distributed services (including its own). In the second scenario, a geo-distributed service renting compute, storage and inter-datacenter bandwidth resources from an infrastructure provider can use CodedBulk to improve bulk transfer throughput without any support from the provider. We have implemented CodedBulk for both scenarios — an overlay service, a software proxy and a hardware proxy. The first two implementations currently run on geo-distributed inter-datacenter WAN. All the three implementations, along with a CodedBulk simulator, are available at: `https://github.com/SynergyLab-Cornell/codedbulk`.

The benefits of network coding depend on the underlying network topology, the number of destinations in individual bulk transfers, the source and set of destinations in each bulk transfer, the number of concurrent transfers and interactive traffic load. To understand the envelope of settings where CodedBulk provides benefits, we evaluate CodedBulk over a testbed comprising 13 geo-distributed datacenters organized around the B4 [25] and Internet2 [5] inter-datacenter WAN topologies, and perform sensitivity analysis of CodedBulk performance against all of the above factors. Our evaluation demonstrates that CodedBulk achieves $1.2-2.5\times$ higher throughput for bulk transfers when compared to existing state-of-the-art mechanisms that do not perform network coding. All the results presented in this paper are for real implementations of CodedBulk; this paper uses no simulation results.

2 CodedBulk Overview

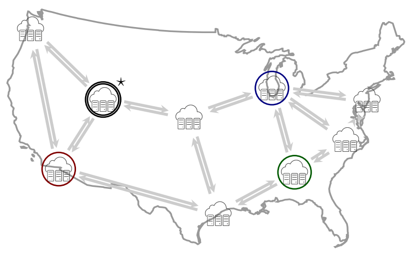
We begin by describing our model for inter-datacenter WANs (§2.1). We then provide a primer for network coding (§2.2). Next, we discuss several pragmatic challenges that have posed a barrier to adoption of network coding in wired networks and how unique characteristics of inter-datacenter WANs enable overcoming these barriers (§2.3). We close the section with a high-level description of the CodedBulk design (§2.4).

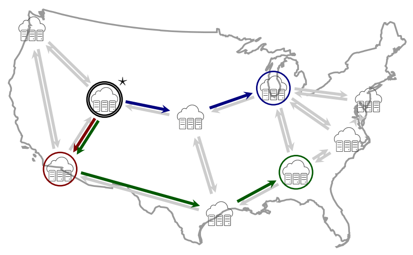
2.1 Preliminaries

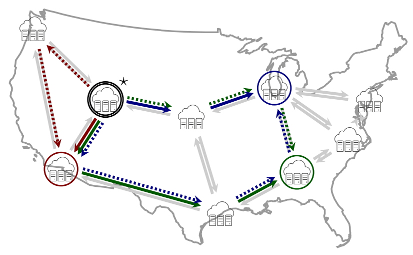
We follow the same terminology as in existing inter-datacenter WAN literature [26, 27, 31, 32, 39, 40, 49, 52]. Specifically, we model the inter-datacenter WAN as a directed graph $G = (V,E)$, where V is the set of nodes denoting datacenters and E is the set of links between pairs of datacenters. To account for the full-duplex nature of inter-datacenter WAN links, we create two links $u \rightarrow v$ and $v \rightarrow u$ for each pair of nodes $(u,v) \in V$ with a physical link between them. Each link has a capacity equal to its bandwidth available for bulk transfers.

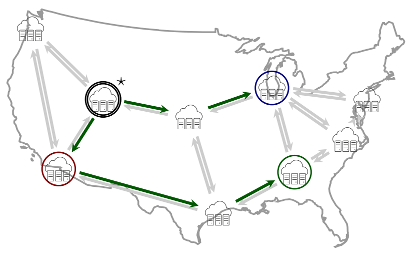
We discuss two important aspects of the network model. First, while links in wired networks are full-duplex, the graph in inter-datacenter literature is usually modeled as a directed graph since links in two directions can have different *available* bandwidths at different times, *e.g.*, due to high-priority interactive traffic using (a fraction of) the bandwidth in one direction. Second, in practice, geo-distributed datacenters are often connected via intermediate routers operating at layer-1 or layer-3; these routers either operate exactly like a relay (have degree two, with incoming bandwidth equal to outgoing bandwidth), or have their bandwidth statically partitioned across multiple outgoing links. Both these cases are equivalent to having a direct link with a specific capacity in each direction between each pair of datacenters with a physical link between them.

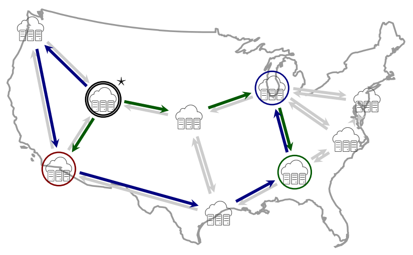
We define bulk transfers as in prior work [23, 25, 26, 27, 31, 32, 38, 39, 40, 49, 52]: transfers that are bandwidth-intensive. A bulk transfer is a source s sending a file to a subset of nodes $T \subseteq V - \{s\}$.

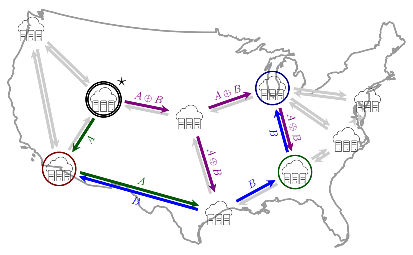

(a) Inter-datacenter bulk transfer example


(b) single-path solution (suboptimal)


(c) multi-path solution (suboptimal)


(d) Steiner-tree solution (suboptimal)


(e) Optimal non-coded solution (computed by hand; efficient algorithms to compute such an optimal solution are not known).



(f) CodedBulk solution (optimal).

Figure 1: Understanding benefits of network coding. (a) An instance of the inter-datacenter bulk transfer problem on the Internet2 topology [5], with one source (marked by a circle with a $\star$) and three destinations (marked by circles). The network model is as described in §2.1. (b, c, d) Existing solutions based on single-path, multi-path and Steiner arborescence packing can be suboptimal (detailed discussion in §2.2). (e) An optimal solution with Steiner arborescence packing (computed by hand); today, computing such a solution requires brute force search which is unlikely to scale to inter-datacenter deployment sizes (tens to hundreds of datacenters) [23, 25]. (f) CodedBulk, using network coding, not only achieves optimal throughput but also admits efficient algorithms to compute the corresponding network codes. More discussion in §2.2.

2.2 Network coding background

Suppose a source wants to send a large file to a single destination, and that it is allowed to use as many paths as possible. If there are no other flows in the network, the maximum achievable throughput (the amount of data received by the destination per unit time) is given by the well-known max-flow min-cut theorem—the achievable throughput is equal to the capacity of the min-cut between the source and the destination in the induced graph. The corresponding problem for a source sending a file to multiple destinations was an open problem for decades. In 2000, a now celebrated paper [9] established that, for a multicast transfer, the maximum achievable throughput is equal to the minimum of the min-cuts between the source and individual destinations. This is also optimal. For general directed graphs, achieving this throughput is not possible using solutions where intermediate nodes simply forward or mirror the incoming data—it necessarily requires intermediate nodes to perform certain computations over the incoming data before forwarding the data [9, 30, 33].

For our network model that captures full-duplex links, network coding achieves optimal throughput (since it subsumes solutions that do not perform coding); however, it is currently not known whether optimal throughput can be achieved without network coding [10, 34]. Figure 1 demonstrates the space of existing solutions, using a bulk transfer instance from our evaluation (§4) on the Internet2 topology. We present additional discussion and examples in [46]. Single-path (also referred to as multiple unicast) solutions, where the source transfers data along a single path to each individual destination, can be suboptimal because they neither utilize all the available network bandwidth, nor do they allow intermediate nodes to forward/mirror data to other destinations. Multi-path solutions, where the source transfers data along all edge-disjoint paths to each individual destinations (paths across destinations do not need to be edge disjoint), can be suboptimal because they do not allow intermediate nodes to forward/mirror data to other destinations.

The current state-of-the-art solutions for our network model are based on Steiner tree (or, more precisely, Steiner arborescence) packing [7, 18, 34]. These solutions use multiple paths, and allow intermediate nodes to mirror and forward the data; however, they can be suboptimal because the problem of computing optimal Steiner tree (or arborescence) packing is NP-hard, and approximation algorithms need to be used [12]. To demonstrate the limitations of existing Steiner packing solutions, consider the example shown in Figure 1(d): here, once the shown Steiner tree is constructed, no additional Steiner trees can be packed in a manner that higher throughput can be achieved. Figure 1(e) demonstrates the complexity of computing an optimal solution (that we constructed by hand)—to achieve the optimal solution shown in the figure, one must explore intermediate solutions that use a suboptimal Steiner tree (shown in blue color). Today, computing such an optimal solution requires a brute force approach, which is unlikely to scale to inter-datacenter network sizes. Thus, we must use approximate suboptimal solutions; to the best of our knowledge, the state-of-the-art algorithms for computing approximate Steiner packing solutions for our network model do not even admit polylogarithmic approximation factors [11, 21].

Network coding avoids the aforementioned limitations of existing solutions by allowing intermediate nodes to perform certain computations, which subsume forwarding and mirroring, on data (as shown in the Figure 1(f) example)—it utilizes multiple paths, guarantees optimal throughput, and admits efficient computation of network codes that achieve optimal throughput [24]. Thus, while designing optimal non-coded solutions for bulk transfers remains an open problem, we can efficiently achieve throughput optimality for inter-datacenter bulk transfers *today* using network coding.

2.3 Resolving pragmatic barriers

While network coding is a powerful technique, its applications to wired networks have been limited in the past due to several pragmatic challenges. In this subsection, we use the example in Figure 1(f) to discuss these challenges, and how inter-datacenter WANs allow overcoming these challenges.

Buffering and computation at intermediate nodes. Network coding requires intermediate nodes to buffer data and perform computations. For instance, the bottom-center node in Figure 1(f) needs to perform XOR operations on packets from two flows A and $A \oplus B$. This requires the node to have storage (to buffer packets from A and $A \oplus B$), and computation resources (to compute $A \oplus (A \oplus B) = B$) in the data plane. While this was challenging in traditional ISP networks, inter-datacenter WANs allow overcoming this barrier easily: as noted in prior studies [31], each node in an inter-datacenter WAN is a datacenter with compute and storage resources that are significantly cheaper, more scalable and faster to deploy than inter-datacenter bandwidth.

Computing and installing network codes. Nodes in Figure 1(f) perform specific actions (forward, mirror, code-and-forward and code-and-mirror). These actions are specified using network codes, computing which requires a priori knowledge of the network topology, and the source and the set of destinations for each bulk transfer. This information was hard to get in ISP networks; however, inter-datacenter WANs already collect and use this information [23, 25, 31, 49]. Network coding also requires transmissions from end-hosts to be coordinated by the controller. In inter-datacenter WAN scenarios, this is feasible as a single entity controls the end-hosts as well as the network. Existing SDN infrastructure [23, 25] is also useful for this purpose—a centralized controller can compute the code, and can populate the forwarding tables of intermediate nodes (using existing support for MPLS tags and multi-path routing) before the bulk transfer is initiated.

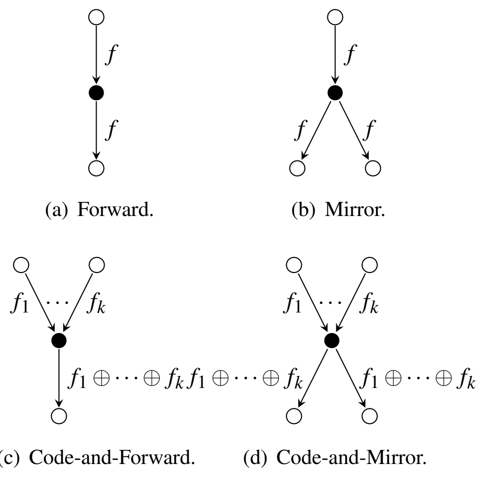


Figure 2: Four basic coding functions available at each intermediate node to implement the network code generated by CodedBulk.

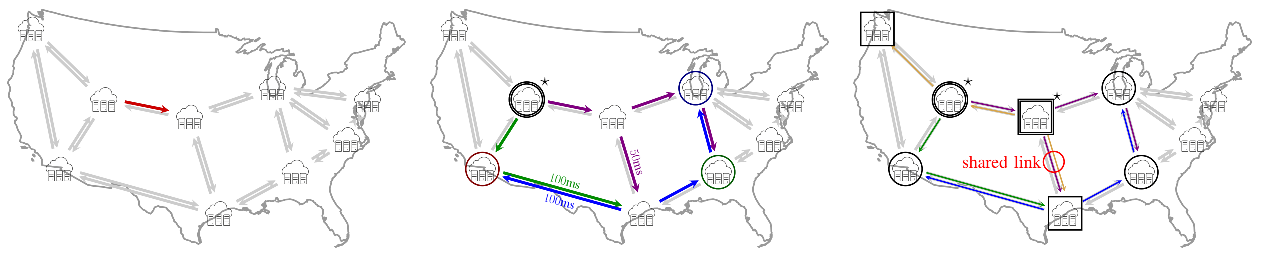
Existing algorithms [24] for computing network codes run in polynomial time, but may not scale to networks with millions of nodes and edges; however, this is not a concern for CodedBulk since inter-datacenter WANs comprise of only hundreds of nodes and links. Computation and installation of network codes, and buffering of data at intermediate nodes may also lead to increased latency for bulk transfers. However, since bulk transfers are not latency-sensitive [23, 25], a slight increase in latency to achieve significantly higher throughput for bulk transfers is a favorable tradeoff [31].

2.4 CodedBulk design overview

The high-level CodedBulk design for a single bulk transfer case can be described using the following five steps:

1. The source node, upon receiving a bulk transfer request, notifies the controller of the bulk transfer. The notification contains the source identifier, the identifiers for each destination, and an optional policy on the set of links or intermediate nodes not to be used (*e.g.*, for security and isolation purposes).
2. The controller maintains a static topology of the inter-datacenter network graph. While optimizations are possible to exploit real-time traffic information, the current CodedBulk implementation does not use such optimizations. The controller computes, for each destination, the set of edge-disjoint paths between the source and the destination, along with the bandwidth for each path. Using these paths, the controller computes the network code for the bulk transfer using the network coding algorithm from [24][1]. The network code comprises of the routing and forwarding

[1]The network coding algorithm in [24] requires as input a directed acyclic graph. However, the multipath set in our construction may lead to a cyclic graph. We use an extension similar to the original network coding paper [9] to generate network codes for cyclic graphs. Please see [46] for details.



(a) Reason: Interactive traffic.   (b) Reason: Non-uniform delay.   (c) Reason: Multiple bulk transfers.

Figure 3: Understanding asymmetric link problem. (a) Due to sporadic high-priority interactive traffic (*e.g.*, the one shown in red), different links may have different (time-varying) bandwidths; (b) If network links have significantly different round trip times, naïvely implementing traditional network coding would require large amount of fast data plane storage to buffer data that arrives early at nodes; (c) multiple concurrent bulk transfers, especially those that end up sharing links, make it hard to efficiently realize traditional network coding solutions that assume a single bulk transfer at all times. Detailed discussion in §3.1.

information for each flow, and the computations done at each intermediate node for the flows arriving at that node. These codes can be expressed as a combination of four basic functions shown in Figure 2.

3. Once the network code is computed, the controller installs the network code on each node that participates in the bulk transfer. We discuss, in §3.3, a mechanism to implement the forwarding and routing functions that requires no changes in existing inter-datacenter WAN infrastructure.

4. Once the code is installed, the controller notifies the source. The source partitions the bulk transfer file into multiple subfiles (defined by the code) and then initiates the bulk transfer using CodedBulk, as described in the remainder of the paper. For instance, for the example of Figure 1(f), the source divides the file into two subfiles (A and B) of equal sizes and transmits them using the code shown in the figure. Each intermediate node *independently* performs CodedBulk's hop-by-hop flow control mechanism. Importantly, a "hop" here refers to a datacenter on the network topology graph. CodedBulk assumes that interactive traffic is always sent with the highest priority, and needs two additional priority levels.

5. Once the bulk transfer is complete, the source notifies the controller. The controller periodically uninstalls the inactive codes from all network nodes.

The core of CodedBulk's mechanisms are to efficiently enable the fourth step. We describe these in the next section.

3 CodedBulk Design

We describe the core techniques in CodedBulk design and implementation. We start by building an in-depth understanding of the asymmetric link problem (§3.1). We then describe how CodedBulk resolves the asymmetric link problem using a custom-designed hop-by-hop flow control mechanism (§3.2). Finally, we discuss the virtual link abstraction that enables implementation of CodedBulk without any modifications in underlying transport- and network-layer protocols (§3.3).

3.1 Understanding fundamental barriers

We start by building an in-depth understanding of the asymmetric link bandwidth problem, and how it renders techniques in network coding literature infeasible in practice. We use Figure 3 for the discussion in this subsection.

Asymmetric links due to sporadic interactive traffic. Inter-datacenter WANs transfer both latency-sensitive interactive traffic (*e.g.*, user commits, like emails and documents) and bandwidth-intensive bulk traffic [23, 25]. While interactive traffic is low-volume, it is unpredictable and is assigned higher priority. This leads to two main challenges. First, links may have different bandwidths available at different times for bulk transfers (as shown in Figure 3(a)). Second, the changes in available bandwidth may be at much finer-grained timescales than the round trip times between geo-distributed datacenters.

Traditional network coding literature does not consider the case of interactive traffic. An obvious way to use traditional network coding solutions for non-uniform link bandwidths is to use traffic shaping to perform network coding on the minimum of the available bandwidth across all links. For instance, in the example of Figure 3(a), if the average load induced by interactive traffic is $0.1\times$ link bandwidth, then one can use network coded transfers only on $0.9\times$ bandwidth. However, the two challenges discussed above make this solution hard, if not infeasible: bandwidths are time-varying, making static rate allocation hard; and, bandwidth changing at much fine-grained timescales than geographic round trip times makes it hard to do dynamic rate allocation.

Asymmetric links due to non-uniform delay. Traditional network coding solutions, at least the practically feasible ones [24], require computations on data arriving from multiple flows in a deterministic manner: packets that need to be coded are pre-defined (during code construction) so as to allow the destinations to decode the original data correctly. To achieve this, existing network coding solutions make one of the two assumptions: either the latency from the source to each individual node is uniform; or, unbounded storage

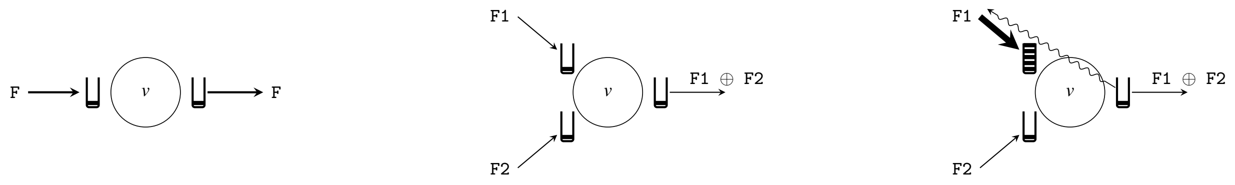


Figure 4: Understanding hop-by-hop flow control for a single bulk transfer. (left) if the outgoing link has enough bandwidth to sustain the rate of incoming traffic (flow `F` in this example), then all buffers will remain unfilled and flow control will not be instantiated; (center) the same scenario as the left figure holds as long as the two conditions hold: (1) both flows that need to be coded at some node v send at the same rate; and (2) the outgoing link has enough bandwidth to sustain the rate of incoming traffic; (right) If two flows need to be coded at some node v, and one of the flows `F1` is sending at higher rate, then the `Rx` buffer for `F1` will fill up faster than it can be drained (due to v waiting for packets of `F2`) and flow control to the downstream node of `F1` will be triggered, resulting in rate reduction for flow `F1`. Detailed discussion in §3.2.

at intermediate nodes to buffer packets from multiple flows. Neither of these assumptions may hold in practice. The delay from the source to individual intermediate nodes can vary by hundreds of milliseconds in a geo-distributed setting (Figure 3(b)). Keeping packets buffered during such delays would require an impractical amount of high-speed storage for high-bandwidth inter-datacenter WAN links: if links are operating at terabits per second of bandwidth, each intermediate node would require hundreds of gigabits or more of storage.

Asymmetric links due to simultaneous bulk transfers. Traditional network coding literature considers only the case of a single bulk transfer. Designing throughput-optimal network codes for multiple concurrent bulk transfers is a long-standing open problem. We do not solve this problem; instead, we focus on optimizing throughput for individual bulk transfers while ensuring that the network runs at high utilization.

Achieving the above two goals simultaneously turns out to be hard, due to each individual bulk transfer observing different delays (between respective source to intermediate nodes) and available link bandwidths due to interactive traffic. Essentially, as shown in Figure 3(c), supporting multiple simultaneous bulk transfers requires additional mechanisms for achieving high network utilization.

3.2 CodedBulk's hop-by-hop flow control

Network coding, by its very nature, breaks the end-to-end semantics of traffic between a source-destination pair, thus necessitating treating the traffic as a set of flows between the intermediate nodes or hops. Recall that a "hop" here refers to a (resource-rich) datacenter on the network graph. To ensure that we do not lose packets at intermediate nodes in spite of the fact that they have limited storage, we rely on a hop-by-hop flow control mechanism—a hop pushes back on the previous hop when its buffers are full. This pushback can be implicit (*e.g.*, TCP flow control) or explicit.

Hop-by-hop flow control is an old idea, dating back to the origins of congestion control [41, 45]. However, our problem is different: traditional hop-by-hop flow control mechanisms operate on individual flows—each downstream flow depends on precisely one upstream flow; in contrast, CodedBulk operates on "coded flows" that may require multiple upstream flows to be encoded at intermediate nodes. Thus, a flow being transmitted at a low rate can affect the overall performance of the transfer (since other flows that need to be encoded with this flow will need to lower their rate as well). This leads to a correlated rate control problem. For instance, in Figure 2(c) and Figure 2(d), flows f_1 to f_k must converge to the same rate so that the intermediate node can perform coding operations correctly without buffering large number of packets. To that end, CodedBulk's hop-by-hop flow control mechanism maintains three invariants:

- All flows within the same bulk transfer that need to be encoded at any node must converge to the same rate;
- All flows from different bulk transfers competing on the congested link bandwidth must converge to a max-min fair bandwidth allocation;
- The network is deadlock-free.

CodedBulk maintains these invariants using a simple idea: careful partitioning of buffer space to flows within and across bulk transfers. The key insight here, that follows from early work on buffer sharing [45], is that for large enough buffers, two flows congested on a downstream link will converge to a rate that corresponds to the fair share of the downstream link bandwidth. We describe the idea of CodedBulk's hop-by-hop flow control mechanism using two scenarios: single isolated bulk transfer and multiple concurrent bulk transfers.

Single bulk transfer. First consider the two simpler cases of forward (Figure 2(a)) and mirror (Figure 2(b)). These cases are exactly similar to traditional congestion control protocols, and hence do not require any special mechanism for buffer sharing. The main challenge comes from Code-and-Forward (Figure 2(c)) and Code-and-Mirror (Figure 2(d)). For these cases, the invariant we require is that the flows being used to compute the outgoing data converge to the same rate since otherwise packets belonging to the flows sending at a higher rate will need to be buffered at the node, requiring high storage. This is demonstrated in Figure 4, center and right figures.

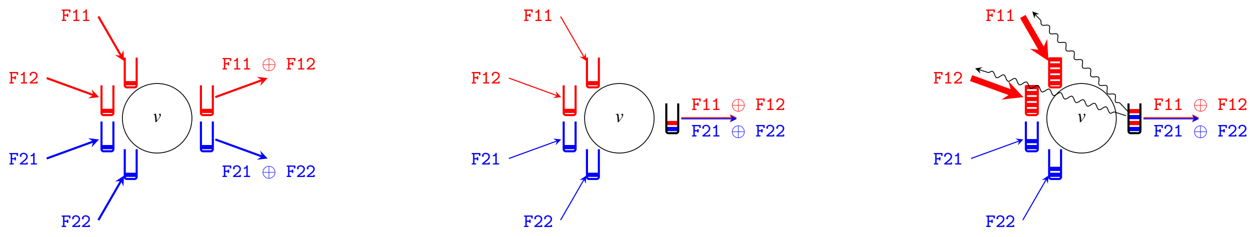


Figure 5: If concurrent bulk transfers use completely different outgoing links (left) or use the same outgoing link but with enough bandwidth (center), the hop-by-hop flow control mechanism does not get triggered. However, if the outgoing link is bandwidth-bottlenecked, and one of the bulk transfers is sending at higher rate (say the red one), then the buffers for the red flows will fill up faster than the buffers for blue flows; at this point, hop-by-hop flow control mechanism will send a pushback to the downstream nodes of the red flows, resulting in reduced rate for the red flows. Detailed discussion in §3.2.

Our insight is that a buffer partitioning mechanism that assigns *non-zero* buffers to each incoming flow maintains the second and the third invariants. It is known that non-zero buffer allocation to each flow at each link leads to deadlock-freedom [45]. It is easy to see that the second invariant also holds—if one of the flows sends at a rate higher than the other (Figure 4(right)), the buffer for this flow will fill up faster than the buffer for the other flow, the flow control mechanism will be triggered, eventually reducing the rate of the flow.

Multiple simultaneous bulk transfers. CodedBulk handles each bulk transfer independently using its hop-by-hop flow control mechanism. Again, we provide intuition using an example. Consider two simultaneous bulk transfers at some intermediate node. If the two bulk transfers use different incoming and outgoing links, these transfers remain essentially independent. So, consider the case when the two bulk transfers compete on one of the incoming or outgoing links. We first discuss when they compete on one of the outgoing links (see Figure 5). If the sum of "coded rates" for individual bulk transfers is less than the outgoing link bandwidth, no flow control is triggered and hence max-min fairness is achieved.

The situation becomes more interesting when the sum of coded rates for individual bulk transfers is greater than the outgoing link bandwidth. In this case, suppose the coded rate of the first bulk transfer is greater than the second one. Then, since outgoing link is shared equally across the two bulk transfers, the buffers for the flows in the first bulk transfer will fill more quickly, leading to triggering the flow control. Thus, flows in the second bulk transfer will reduce the transmission rate finally converging to outgoing link being shared equally across the two coded bulk transfers.

Multi-priority transfers to fill unfilled pipes. Asymmetric link problem, despite our hop-by-hop flow control mechanism, can lead to "unfilled pipes" (Figure 6). Essentially, due to different bulk transfers bottlenecked at different links, no more coded traffic can be pushed into the network despite some links having available bandwidth. CodedBulk fills such unfilled pipes by sending uncoded data; however, to ensure minimal impact on the coded traffic, CodedBulk uses a lower

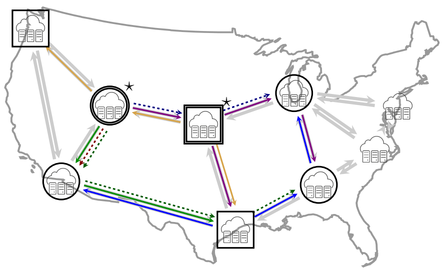


Figure 6: By sending non-coded flows at lower priority (the gray traffic), CodedBulk exploits the "unfilled pipes" left by coded traffic.

priority level for the uncoded data. Thus, CodedBulk uses three priority levels—the highest priority is for interactive traffic, the medium priority for coded traffic, and a lower priority level for uncoded traffic.

3.3 Virtual links

CodedBulk's hop-by-hop flow control mechanism from the previous section addresses the asymmetric link problem, at a design level. In this subsection, we first discuss a challenge introduced by network coding in terms of efficiently implementing the hop-by-hop flow control mechanism. We then introduce the abstraction of virtual links, that enables an efficient realization of CodedBulk's flow control mechanism without any changes in the underlying transport protocol. For this subsection, we use TCP as the underlying congestion control mechanism; however, the idea generalizes to any transport protocol that supports flow control.

The challenge. In traditional store-and-forward networks, implementing hop-by-hop flow control is simple: as data for a flow is received in the Rx buffer, it can be directly copied to the Tx buffer of the next hop, either using blocking or non-blocking system calls. When implementing network coding, this becomes non-trivial—since data from multiple flows needs to be coded together, neither blocking nor non-blocking calls can be used since these calls fundamentally operate on individual flows. For instance, consider the case of Figure 1(f), where a node needs to compute $(A \oplus B) \oplus A$ using packets

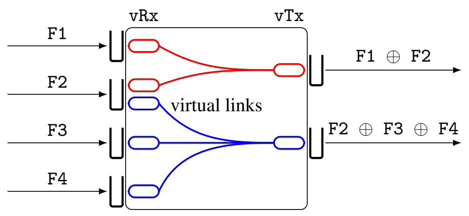


Figure 7: The figure demonstrates the virtual link abstraction used by CodedBulk to implement its hop-by-hop flow control mechanism without any modifications in the underlying network stack.

from the two flows. Blocking calls require expensive coordination between two buffers since the node requires data from *both* flows to be available before it can make progress. Non-blocking calls cannot be used either—the call will return the data from one of the flows, but this data cannot be operated upon until the data from the other flow(s) is also available. The fundamental challenge here is that we need efficient ways to block on multiple flows, and return the call only when data is available in all flows that need to be coded.

It may be tempting to have a shared buffer across different flows that need to be coded together. The problem, however, is that shared buffers will lead to deadlocks [41]—if one of the flows is sending data at much higher rate than the other flows, it will end up saturating the buffer space, the other flows will starve, and consequently the flow that filled up the buffer will also not make progress since it waits to receive data from other flows to be coded with. As discussed in §3.2, non-zero buffer allocation to each individual flow is a necessary condition for avoiding deadlocks in hop-by-hop flow control mechanisms.

Virtual links (see Figure 7). CodedBulk assigns each individual bulk transfer a virtual link per outgoing physical link; each virtual link has a single virtual transmit buffer `vTx` and as many virtual receive buffers `vRx` as the number of flows to be coded together for that outgoing link. For instance, consider four incoming flows in a bulk transfer `F1, F2, F3, F4` such that `F1 ⊕ F2` is forwarded on one of outgoing physical links, and `F2 ⊕ F3 ⊕ F4` is forwarded on another outgoing physical link. Then, CodedBulk creates two virtual links each having one `vTx`; the first virtual link has two `vRx` (one for `F1` packets and another for `F2` packets) and the second virtual link has three `vRx` (one for each of `F2, F3` and `F4` packets). Virtual links are created when the controller installs the network codes, since the knowledge of the precise network code to be used for the bulk transfer is necessary to create virtual links. As new codes are installed, CodedBulk reallocates the space to each `vTx` and `vRx`, within and across virtual links, to ensure that all virtual buffers have non-zero size.

Using these virtual links resolves the aforementioned challenge with blocking and non-blocking calls. Indeed, either of the calls can now be used since the "correlation" between the flows is now captured at the virtual link rather than at the flow control layer. Data from the incoming socket buffers for individual flow is now copied to their respective `vRx` buffers, either using blocking or non-blocking calls. A separate thread asynchronously checks when the size of all the `vRx` buffers is non-zero (each buffer has at least one packet); and when this happens, performs the coding operations and copies the resulting packet to the corresponding `vTx`.

4 Evaluation

We implement CodedBulk in C++ and use TCP Cubic as the underlying transport protocol. We use default TCP socket buffers, with interactive traffic sent at higher priority than bulk transfers (using TCP differentiated services field) set using standard Linux socket API. To enforce priority scheduling, we use Linux tc at each network interface.

We now evaluate CodedBulk implementation over two real geo-distributed cloud testbeds. We start by describing the experiment setup (§4.1). We then discuss the results for CodedBulk implementation over a variety of workloads with varying choice of source and destination nodes for individual bulk transfers, interactive traffic load, number of concurrent bulk transfers, and number of destinations in individual bulk transfers (§4.2). Finally, we present scalability of our CodedBulk prototype implementation in software and hardware (§4.3).

4.1 Setup

Testbed details. To run our experiments, we use two testbeds that are built as an overlay on geo-distributed datacenters from Amazon AWS. Our testbeds use 13 and 9 geo-distributed datacenters organized around B4 [25] and Internet2 [5] topologies, respectively. The datacenter locations are chosen to closely emulate the two topologies and the corresponding geographical distances and latencies. Within each datacenter, we take a high-end server; for every link in the corresponding topology, we establish a connection between the servers across various datacenter using the inter-datacenter connectivity provided by Amazon AWS. To reduce cost of experimentation, we throttle the bandwidth between each pair of servers to 200 Mbps for our experiments. The precise details on the inter-datacenter connectivity provided by Amazon AWS, whether they use public Internet or dedicated inter-datacenter links, is not publicly known. We run all the experiments for each individual figure within a short period of time; while the inter-datacenter links provided by Amazon AWS may be shared and may cause interference, we observe fairly consistent inter-datacenter bandwidth during our experiments. We use a server in one of the datacenters to act as the centralized controller (to compute and install network codes on all servers across our testbed).

Workloads. As mentioned earlier, the benefits of network coding depend on the underlying network topology, the number of destinations in individual bulk transfers, the location of the source and the set of destinations in each bulk transfer, the number of concurrent transfers and interactive traffic load.

While there are no publicly available datasets or workloads for inter-datacenter bulk transfers, several details are known. For instance, Facebook [43], Netflix [1], Azure SQL database [3] and CloudBasic SQL server [4] perform replication to (dynamically) locate their datasets closer to the customers; for such applications, the destinations for each replica are selected based on the diurnal traffic patterns and customer access patterns. Many other applications [13, 19, 23, 25, 31, 50] perform replication levels based on user needs, usually for fault tolerance; for such applications, the choice of destinations may be under the control of the service provider.

We perform experiments to understand the envelope of workloads where CodedBulk provides benefits. Our evaluation performs sensitivity analysis against all parameters—we use two inter-datacenter network topologies, interactive traffic load varying from $0.05-0.2\times$ of the link bandwidth, the number of destinations/replicas in individual bulk transfers varying from 2 to maximum possible (depending on the topology), and the number of concurrent bulk transfers varying from 1 to the maximum possible (depending on the topology). For each setting, we run five experiments; for individual bulk transfers within each experiment, we choose a source uniform randomly across all nodes, and choose the destinations from the remaining nodes. Each node can be the source of only a single bulk transfer but may serve as a destination for other bulk transfers; furthermore, each node may serve as a destination for multiple bulk transfers. We present the average throughput across all experiments, as well as the variance (due to different choices of the source and set of destination across different experiments).

We generate interactive traffic between every pair of datacenters, with arrival times such that the overall load induced by the interactive traffic varies between $0.05-0.2\times$ of the link bandwidth; while 0.2 load is on the higher end in real-world scenarios [23, 25], it allows us to evaluate extreme workloads. Interactive traffic is always assigned the highest priority and hence, all our evaluated schemes will get the same interactive traffic throughput. Our results, thus, focus on bulk traffic throughput.

As mentioned above, there are no publicly available datasets or workloads for inter-datacenter bulk transfers. We make what we believe are sensible choices, state these choices explicitly, and to whatever extent possible, evaluate the sensitivity of these choices on our results. Nonetheless, our results are dependent on these choices, and more experience is needed to confirm whether our results generalize to workloads observed in large-scale deployments.

Evaluated schemes. We compare CodedBulk with three mechanisms for bulk data transfers discussed earlier in Figure 1—single-path, multi-path, and Steiner arborescence packing—each of which take the graph described in §2.1 as an input. For the single-path mechanism, the bulk traffic is transferred along the shortest path between the source and each destination; when multiple choices are available, the mechanism selectively picks paths that minimize total bandwidth usage (*e.g.*, to send bulk traffic to two destinations d_1, d_2, the mechanism prefers the path $s \rightarrow d_1 \rightarrow d_2$, where d_1 can simply forward the data to d_2, over two different paths $s \rightarrow d_1$ and $s \rightarrow v \rightarrow d_2$ for some other node v). The multi-path mechanism selects edge-disjoint paths from the source to each destination so as to greedily minimize the sum of the path lengths. Our third baseline is a state-of-the-art Steiner arborescence based multicast mechanism that allows each node in the network (including the destinations) to forward (Figure 2(a)) and mirror (Figure 2(b)) incoming data. To compute the Steiner arborescence, we use the algorithm in [48] that is also used in other Steiner arborescence based inter-datacenter multicast proposals [38, 39, 40]. We take the arborescence computed by the algorithm, and integrate it with a store-and-forward model, along with TCP for transfers between every pair of nodes in the Steiner arborescence. For concurrent bulk transfers, paths and Steiner arborescence are computed independently for each individual bulk transfer.

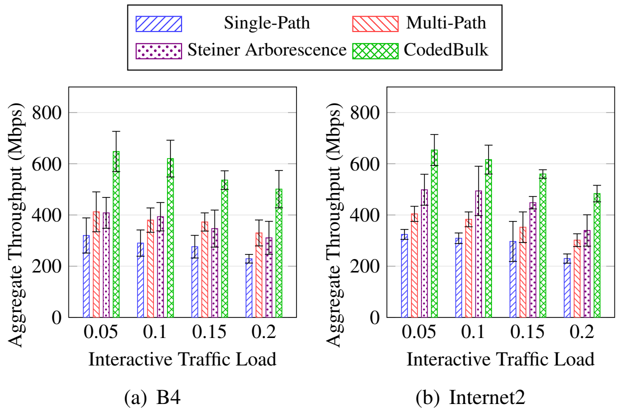
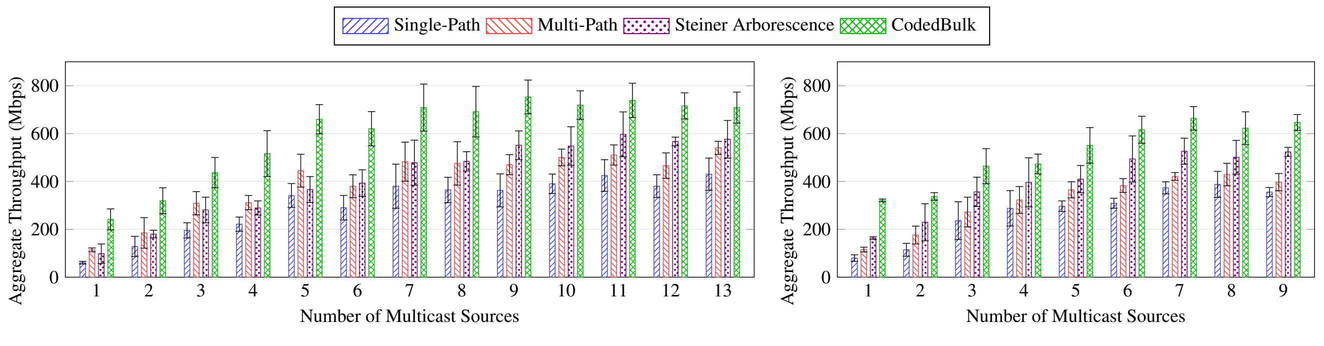


Figure 8: Performance of various bulk transfer mechanisms for varying interactive traffic load. For the B4 topology, CodedBulk improves bulk transfer throughput by $1.9-2.2\times$, $1.4-1.6\times$ and $1.5-1.6\times$ compared to single-path, multi-path, and Steiner arborescence based mechanisms, respectively. For the Internet2 topology, corresponding numbers are $1.9-2.1\times$, $1.6\times$, and $1.2-1.4\times$ (discussion in §4.2).

For CodedBulk, we use a finite field size of 2^8, that is all finite field operations are performed on individual bytes; this finite field size is sufficient for inter-datacenter networks with as many as 128 datacenters. We could have used a smaller finite field size since our topologies are much smaller than real inter-datacenter network topologies; however, this allow us to keep the operations byte aligned, which simplifies CodedBulk software and hardware implementation.

Performance metric. Our primary metric is the aggregate throughput for bulk transfers. For each individual bulk transfer, the throughput is computed as the maximum throughput at which the source can send to *all* destinations. We then calculate the aggregate throughput by summing up the throughput of all bulk transfers.



(a) B4 (b) Internet2

Figure 9: Performance of various bulk transfer mechanisms for varying number of concurrent bulk transfers. CodedBulk improves the bulk transfer throughput by $1.6-4\times$, $1.3-2.8\times$ and $1.2-2.5\times$ when compared to single-path, multi-path, and Steiner arborescence based mechanisms, respectively (discussion in §4.2).

4.2 Geo-distributed Testbed Experiments

We compare CodedBulk with the three baselines for varying interactive traffic loads, varying number of concurrent bulk transfers and varying number of replicas per bulk transfer.

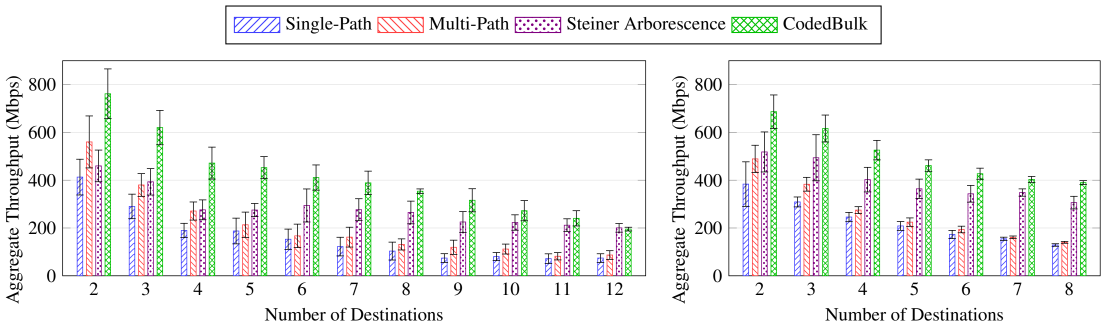
Varying interactive traffic load. Figure 8 presents the achievable throughput for each scheme with varying interactive traffic load. For this experiment, we use 3-way replication and 6 concurrent transfers (to capture the case of Facebook, Netflix, Azure SQL server and CloudBasic SQL server as discussed above), and vary the interactive traffic load from $0.05-0.2\times$ of the link bandwidth.

As expected, the throughput for all mechanisms decreases as interactive traffic load increases. Note that, in corner-case scenarios, the multi-path mechanism can perform slightly worse than single-path mechanism for multiple concurrent bulk transfers due to increased interference across multiple flows sharing a link, which in turn results in increased convergence time for TCP (see [46] for a concrete example). Overall, CodedBulk improves the bulk traffic throughput over single-path, multi-path and Steiner arborescence mechanisms by $1.9-2.2\times$, $1.4-1.6\times$ and $1.2-1.6\times$, respectively, depending on the interactive traffic load and the network topology. Single-path mechanisms perform poorly because they do not exploit all the available bandwidth in the network. Both multi-path and Steiner arborescence based mechanisms exploit the available bandwidth as much as possible. However, multi-path mechanisms suffer since they do not allow intermediate nodes to mirror and forward to the destinations. Steiner arborescence further improves upon multi-path mechanisms by allowing intermediate nodes to mirror and forward data, but they suffer due to approximation algorithm often leading to suboptimal solutions. CodedBulk's gains over multi-path and Steiner arborescence mechanisms are, thus, primarily due to CodedBulk's efficient realization of network coding—it not only uses all the available links, but also computes the optimal coding strategy (unlike Steiner arborescence mechanism that uses an approximation algorithm). The Steiner arborescence mechanism performs better on Internet2 topology because of its sparsity—fewer links in the network means a Steiner arborescence solution is more likely to be the same as network coding solution due to fewer opportunities to perform coding. Nevertheless, CodedBulk outperforms Steiner arborescence based mechanism by $1.4\times$.

Varying number of concurrent bulk transfers. Figure 9 shows the performance of the four mechanisms with varying number of concurrent transfers. For this evaluation, we use the same setup as earlier—3-way replication, multiple runs with each run selecting different sources and set of destinations, etc.—with the only difference being that we fix the interactive traffic load to 0.1 and vary the number of concurrent bulk transfers. With larger number of concurrent bulk transfers, Steiner arborescence mechanisms slightly outperform multi-path due to improved arborescence construction. Nevertheless, CodedBulk provides benefits across all sets of experiments, achieving $1.2-2.5\times$ improvements over Steiner arborescence based mechanisms. The gains are more prominent for B4 topology and for fewer number of concurrent transfers, since CodedBulk gets more opportunities to perform network coding at intermediate nodes in these scenarios.

Varying number of destinations/replicas per bulk transfer. Figure 10 shows the performance of the four mechanisms with varying number of destinations/replicas for individual bulk transfers. For this evaluation, we use the same setup as Figure 8—6 concurrent bulk transfers, multiple runs with each run selecting different sources and set of destinations, etc.—with the only difference being that we fix the interactive traffic load to 0.1 and vary the number of destinations/replicas per bulk transfer from 2 to the maximum allowable replicas for individual topologies. Notice the results show the aggregate throughput per destination.

As the number of destinations per bulk transfer increases, the per-destination throughput decreases for all schemes (although, as expected, the sum of throughput of all destinations increases). Note that multi-path outperforming single-path



(a) B4 (b) Internet2

Figure 10: Performance of various bulk transfer mechanisms for varying number of destinations/replicas per bulk transfer. CodedBulk improves the bulk transfer throughput over single-path and multi-path mechanisms by $1.8-4.3\times$ and $1.4-2.9\times$, respectively, depending on the number of destinations in each bulk transfer and depending on the topology. CodedBulk outperforms Steiner arborescence mechanisms by up to $1.7\times$ when the number of destinations is not too large. When each bulk transfer creates as many replicas as the number of datacenters in the network, CodedBulk performs comparably with Steiner arborescence. Note that the aggregate bulk throughput reduction is merely because each source is transmitting to increasingly many destinations, but the metric only captures the average throughput per destination. Discussion in §4.2.

and Steiner arborescence based mechanism in Figure 10(a) is primarily due to B4 topology being dense, thus providing enough path diversity to offset the benefits of approximate Steiner arborescence construction. Figure 10(a) and 10(b) show that CodedBulk outperforms single-path and multi-path mechanisms by $1.8-4.3\times$ and $1.4-2.9\times$, depending on the number of destinations and on the topology; moreover, the relative gains of CodedBulk improve as number of destinations increases. The comparison with Steiner arborescence based mechanism is more nuanced. CodedBulk achieves improved performance when compared to Steiner arborescence based mechanism when number of destinations is less than 10 for B4 topology, and less than 6 for Internet2 topology. The performance difference is minimal for larger number of destination. The reason is that for larger number of replicas/destinations, each source is multicasting to almost all other nodes in the network; in such cases, the benefits of coding reduce when compared to forwarding and mirroring of data at intermediate nodes and at the destination nodes as in Steiner arborescence based mechanism. Thus, the benefits of CodedBulk may be more prominent when the number of replicas is a bit smaller than the total number of datacenters in the network.

4.3 Microbenchmarks

We now evaluate CodedBulk performance in terms of scalability of its software and hardware implementations. Our goal here is to demonstrate the feasibility of CodedBulk implementation; deployment of CodedBulk in large-scale systems would typically require much more optimized implementation since the traffic volume is expected to be much higher.

Software implementation. CodedBulk software implementation runs on commodity datacenter servers, performing network coding as discussed in §3. Figure 11 shows the scalability of CodedBulk software implementation. We observe that CodedBulk implementation scales well with number of cores,

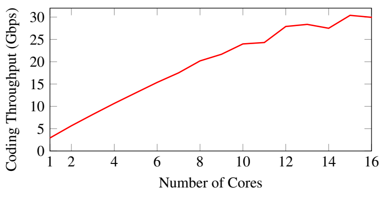


Figure 11: CodedBulk implementation performs network coding for as much as 31Gbps worth of traffic using a commodity 16 core server, achieving roughly linearly coding throughput scalability with number of cores.

Element	Used	Available	Utilization
LUT	69052	433200	15.94%
BRAM	1365	1470	92.86%

Table 1: Resource utilization of CodedBulk implementation on Xilinx Virtex-7 XC7VX690T FPGA (250 MHz clock). Our implementation provides up to 31.25 Gbps throughput with 15.94% LUTs and 92.86% BRAMs. No DSP is needed in our design.

with a single 16-core server being able to perform network coding at line rate for as much as 31Gbps worth of traffic.

Hardware implementation. We have synthesized an end-to-end CodedBulk implementation on an FPGA. For our CodedBulk hardware implementation, we had two choices. First, we could implement a finite field engine that performs finite field operations during the network coding process; or second, we could precompute and store finite field operation results, and use a simple look up table while performing network coding operations. The first approach requires multiple clock cycles to encode two bytes from two different packets; the second approach trades off BRAM to save cycles during coding operations. Since CodedBulk uses a small finite field size (2^8), the second approach offers a better tradeoff — it requires just 256×256 byte look up table per 16 bytes for individual operations to complete in one cycle. We replicate the lookup table accordingly to perform network coding for all bytes in an MTU-sized packet within a single clock cycle. Table 1 shows the results for CodedBulk hardware implementation on Xilinx Virtex-7 XC7VX690T FPGA, which offers 100, 200, and 250 MHz fabric clocks. With respect to the clocks, our FPGA-based codec can achieve throughput 12.5, 25, and 31.25 Gbps. Without needing any DSP, our hardware design consumes 92.86% BRAMs and only 15.94% LUTs.

We believe that trading off compute and storage resources to improve inter-datacenter bulk transfer throughput is a favorable tradeoff to make. However, more experience from industry is needed to do a thorough cost/benefit analysis.

5 Related Work

We have already discussed the differences between CodedBulk's goals and the traditional multicast problem in ISP networks; it would be futile to attempt to summarize the vast amount of literature from ISP multicast problem. We compare and contrast CodedBulk with two more closely related key areas of research: inter-datacenter WAN transfers, and network coding applications in computer networks.

Inter-datacenter bulk transfers. There has been significant amount of recent work on optimizing inter-datacenter bulk transfers [26, 27, 31, 32, 38, 39, 40, 49, 52]. These works optimize inter-datacenter bulk transfers along a multitude of performance metrics, including improving flow completion time [27, 38, 39, 40, 49, 52], and throughput for bulk transfers [26, 31, 32]. CodedBulk's goals are aligned more closely with the latter, and are complementary to the former—CodedBulk improves the throughput for bulk transfers; any bulk transfer scheduling mechanism can be used on top of CodedBulk to meet the needs for timely transfers.

As discussed earlier, the state-of-the-art approach for high-throughput inter-datacenter bulk transfers are based on packing of Steiner arborescence: here, each intermediate node as well as destination nodes are allowed to forward and mirror data toward other destination nodes. Several recent inter-datacenter bulk transfer proposals [38, 39, 40] are based on this approach. Our evaluation in §4 shows that CodedBulk achieves throughput improvements over state-of-the-art Steiner arborescence based mechanisms in a wide variety of scenarios. This is because all prior techniques are limited by network capacity, and by limitations of existing non-coded techniques to achieve this capacity.

CodedBulk, by using network coding, achieves improvement in throughput for bulk transfers by trading off a small amount of compute and storage resources.

Network coding in computer networks. Network coding has successfully been applied to achieve higher throughput in wireless networks [20, 29], in TCP-based networks [44], in content distribution [17, 35, 36], in peer-to-peer communication [16], to name a few; please see [15] for additional applications of network coding. Our goals are complementary—enabling network coding for high-throughput inter-datacenter WAN bulk transfers by exploiting the unique characteristics of these networks. Throughout the paper, we have outlined the unique challenges introduced by applications of network coding in wired networks, and how CodedBulk overcomes these challenges. Our design can be applied to any of the applications where network coding is useful.

Network code construction algorithms. Early incarnations of network coding solutions used a technique referred to as random linear network coding [9, 22]. These random linear network codes have the benefit of being independent of the network topology. However, they have high implementation cost: they require complex operations at intermediate nodes (due to computations over large finite field sizes and due to requiring additional packet header processing). In addition, realizing random linear network codes in practice also requires changes in packet header format. Follow-up research has led to efficient construction of network codes [24]—for a bulk transfer to T destinations, it suffices for intermediate nodes to perform computations over a finite field of size at most $2|T|$; if the min-cut is h, the complexity of computations at the source and at the destination are $O(h)$ and $O(h^2)$, respectively. In §4.1, we discussed how at the inter-datacenter WAN scale, these computations entail simple and efficient byte-level XOR operations. Furthermore, these codes can be realized without any changes in the packet header format. CodedBulk, thus, uses the network code construction algorithm of [24].

6 Conclusion

We have presented the design, implementation and evaluation of CodedBulk, an end-to-end system for high-throughput inter-datacenter bulk transfers. CodedBulk uses network coding, a technique that guarantees optimal throughput for individual bulk transfers. To achieve this, CodedBulk resolves the many pragmatic and fundamental barriers faced in the past in realizing the benefits of network coding in wired networks. Using an end-to-end implementation of CodedBulk over a geo-distributed inter-datacenter network testbed, we have shown that CodedBulk improves throughput for inter-datacenter bulk transfers by $1.2-2.5\times$ when compared to state-of-the-art mechanisms that do not perform coding.

Acknowledgments

We would like to thank our shepherd Keith Winstein, anonymous NSDI reviewers, Rashmi Vinayak and Robert Kleinberg for their insightful feedback. We would also like to thank the students in Cornell graduate computer networking class (CS6450) for many helpful discussions during the early parts of this project. This work was supported in part by a Google PhD fellowship, NSF 1704742, ONR N00014-17-1-2419, and gifts from Snowflake and Cornell-Princeton Network Programming Initiative.

References

[1] [ARC 305] How Netflix leverages multiple regions to increase availability. https://tinyurl.com/4mac28jy.

[2] Cisco annual Internet report (2018–2023) white paper. https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html.

[3] Creating and using active geo-replication – Azure SQL database. https://docs.microsoft.com/en-us/azure/azure-sql/database/active-geo-replication-overview.

[4] Geo-replication/multi-AR. http://cloudbasic.net/documentation/geo-replication-active/.

[5] The Internet2 network. https://internet2.edu/.

[6] Mapping Netflix: Content delivery network spans 233 sites. http://datacenterfrontier.com/mapping-netflix-content-delivery-network/.

[7] Steiner tree problem. https://en.wikipedia.org/wiki/Steiner_tree_problem.

[8] Using replication across multiple data centers. https://docs.oracle.com/cd/E19528-01/819-0992/6n3cn7p3l/index.html.

[9] R. Ahlswede, N. Cai, S.-Y. R. Li, and R. W. Yeung. Network information flow. *IEEE Transactions on Information Theory*, 46(4):1204–1216, 2000.

[10] M. Braverman, S. Garg, and A. Schvartzman. Coding in undirected graphs is either very helpful or not helpful at all. In *ITCS*, 2017.

[11] M. Charikar, C. Chekuri, T.-Y. Cheung, Z. Dai, A. Goel, S. Guha, and M. Li. Approximation algorithms for directed steiner problems. *Journal of Algorithms*, 33(1):73–91, 1999.

[12] J. Cheriyan and M. R. Salavatipour. Hardness and approximation results for packing steiner trees. *Algorithmica*, 45(1):21–43, 2006.

[13] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems*, 31(3):8, 2013.

[14] R.-J. Essiambre and R. W. Tkach. Capacity trends and limits of optical communication networks. *Proc. IEEE*, 100(5):1035–1055, 2012.

[15] C. Fragouli, J.-Y. Le Boudec, and J. Widmer. Network coding: an instant primer. *ACM SIGCOMM Computer Communication Review*, 36(1):63–68, 2006.

[16] C. Gkantsidis, J. Miller, and P. Rodriguez. Comprehensive view of a live network coding p2p system. In *IMC*, 2006.

[17] C. Gkantsidis and P. R. Rodriguez. Network coding for large scale content distribution. In *INFOCOM*, 2005.

[18] M. X. Goemans and Y.-S. Myung. A catalog of steiner tree formulations. *Networks*, 23(1):19–28, 1993.

[19] A. Gupta, F. Yang, J. Govig, A. Kirsch, K. Chan, K. Lai, S. Wu, S. G. Dhoot, A. R. Kumar, A. Agiwal, S. Bansali, M. Hong, J. Cameron, M. Siddiqi, D. Jones, J. Shute, A. Gubarev, S. Venkataraman, and D. Agrawal. Mesa: Geo-replicated, near real-time, scalable data warehousing. *VLDB*, 2014.

[20] J. Hansen, D. E. Lucani, J. Krigslund, M. Médard, and F. H. Fitzek. Network coded software defined networking: Enabling 5G transmission and storage networks. *IEEE Communications Magazine*, 53(9):100–107, 2015.

[21] M. Hauptmann and M. Karpiński. *A compendium on Steiner tree problems*. Inst. für Informatik, 2013.

[22] T. Ho, M. Médard, R. Koetter, D. R. Karger, M. Effros, J. Shi, and B. Leong. A random linear network coding approach to multicast. *IEEE Transactions on Information Theory*, 52(10):4413–4430, 2006.

[23] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven WAN. In *SIGCOMM*, 2013.

[24] S. Jaggi, P. Sanders, P. A. Chou, M. Effros, S. Egner, K. Jain, and L. M. Tolhuizen. Polynomial time algorithms for multicast network code construction. *IEEE Transactions on Information Theory*, 51(6):1973–1982, 2005.

[25] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a globally-deployed software defined WAN. In *SIGCOMM*, 2013.

[26] X. Jin, Y. Li, D. Wei, S. Li, J. Gao, L. Xu, G. Li, W. Xu, and J. Rexford. Optimizing bulk transfers with software-defined optical WAN. In *SIGCOMM*, 2016.

[27] S. Kandula, I. Menache, R. Schwartz, and S. R. Babbula. Calendaring for wide area networks. In *SIGCOMM*, 2014.

[28] S. Katti, S. Gollakota, and D. Katabi. Embracing wireless interference: Analog network coding. In *SIGCOMM*, 2012.

[29] S. Katti, H. Rahul, W. Hu, D. Katabi, M. Médard, and J. Crowcroft. XORs in the air: Practical wireless network coding. In *SIGCOMM*, 2006.

[30] R. Koetter and M. Médard. An algebraic approach to network coding. *IEEE/ACM Transactions on Information Theory*, 11(5):782–795, 2003.

[31] N. Laoutaris, M. Sirivianos, X. Yang, and P. Rodriguez. Inter-datacenter bulk transfers with NetStitcher. In *SIGCOMM*, 2011.

[32] N. Laoutaris, G. Smaragdakis, R. Stanojevic, P. Rodriguez, and R. Sundaram. Delay tolerant bulk data transfers on the Internet. *IEEE/ACM Transactions on Networking*, 21(6):1852–1865, 2013.

[33] S.-Y. R. Li, R. W. Yeung, and N. Cai. Linear network coding. *IEEE Transactions on Information Theory*, 49(2):371–381, 2003.

[34] Z. Li, B. Li, and L. C. Lau. A constant bound on throughput improvement of multicast network coding in undirected networks. *IEEE Transactions on Information Theory*, 55(3):1016–1026, 2009.

[35] Z. Liu, C. Wu, B. Li, and S. Zhao. UUSee: Large-scale operational on-demand streaming with random network coding. In *INFOCOM*, 2010.

[36] E. Magli, M. Wang, P. Frossard, and A. Markopoulou. Network coding meets multimedia: A review. *IEEE Transactions on Multimedia*, 15(5):1195–1212, 2013.

[37] P. P. Mishra and H. Kanakia. A hop by hop rate-based congestion control scheme. In *SIGCOMM*, 1992.

[38] M. Noormohammadpour, S. Kandula, C. S. Raghavendra, and S. Rao. Efficient inter-datacenter bulk transfers with mixed completion time objectives. *Computer Networks*, 164:106903, 2019.

[39] M. Noormohammadpour, C. S. Raghavendra, S. Kandula, and S. Rao. QuickCast: Fast and efficient inter-datacenter transfers using forwarding tree cohorts. In *INFOCOM*, 2018.

[40] M. Noormohammadpour, C. S. Raghavendra, S. Rao, and S. Kandula. DCCast: Efficient point to multipoint transfers across datacenters. In *HotCloud*, 2017.

[41] C. Özveren, R. Simcoe, and G. Varghese. Reliable and efficient hop-by-hop flow control. In *SIGCOMM*, 1994.

[42] G. Ramamurthy and B. Sengupta. A predictive hop-by-hop congestion control policy for high speed networks. In *INFOCOM*, 1993.

[43] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *SIGCOMM*, 2015.

[44] J. K. Sundararajan, D. Shah, M. Médard, M. Mitzenmacher, and J. Barros. Network coding meets TCP. In *INFOCOM*, 2009.

[45] A. S. Tanenbaum and D. Wetherall. *Computer Networks, 5th Edition*. Pearson, 2011.

[46] S.-H. Tseng, S. Agarwal, R. Agarwal, H. Ballani, and A. Tang. Codedbulk: Inter-datacenter bulk transfers using network coding. Technical report, https://github.com/SynergyLab-Cornell/codedbulk.

[47] M. Vuppalapati, J. Miron, R. Agarwal, D. Truong, A. Motivala, and T. Cruanes. Building an elastic query engine on disaggregated storage. In *NSDI*, 2020.

[48] D. Watel and M.-A. Weisser. A practical greedy approximation for the directed Steiner tree problem. *Journal of Combinatorial Optimization*, 32(4):1327–1370, 2016.

[49] Y. Wu, Z. Zhang, C. Wu, C. Guo, Z. Li, and F. C. Lau. Orchestrating bulk data transfers across geo-distributed datacenters. *IEEE Transactions on Cloud Computing*, 5(1):112–125, 2017.

[50] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha. SPANStore: Cost-effective geo-replicated storage spanning multiple cloud services. In *SOSP*, 2013.

[51] Y. Yi and S. Shakkottai. Hop-by-hop congestion control over a wireless multi-hop network. *IEEE/ACM Transactions on Networking*, 15(1):133–144, 2007.

[52] H. Zhang, K. Chen, W. Bai, D. Han, C. Tian, H. Wang, H. Guan, and M. Zhang. Guaranteeing deadlines for inter-data center transfers. *IEEE/ACM Transactions on Networking*, 25(1):579–595, 2017.

Twenty Years After: Hierarchical Core-Stateless Fair Queueing

Zhuolong Yu
Johns Hopkins University

Jingfeng Wu
Johns Hopkins University

Vladimir Braverman
Johns Hopkins University

Ion Stoica
UC Berkeley

Xin Jin
Peking University

Abstract

Core-Stateless Fair Queueing (CSFQ) is a scalable algorithm proposed more than two decades ago to achieve fair queueing without keeping per-flow state in the network. Unfortunately, CSFQ did not take off, in part because it required protocol changes (i.e., adding new fields to the packet header), and hardware support to process packets at line rate.

In this paper, we argue that two emerging trends are making CSFQ relevant again: (i) cloud computing which makes it feasible to change the protocol within the same datacenter or across datacenters owned by the same provider, and (ii) programmable switches which can implement sophisticated packet processing at line rate. To this end, we present the first realization of CSFQ using programmable switches. In addition, we generalize CSFQ to a multi-level hierarchy, which naturally captures the traffic in today's datacenters, e.g., tenants at the first level and flows of each tenant at the second level of the hierarchy. We call this scheduler Hierarchical Core-Stateless Fair Queueing (HCSFQ), and show that it is able to accurately approximate hierarchical fair queueing. HCSFQ is highly scalable: it uses just a single FIFO queue, does not perform per-packet scheduling, and only needs to maintain state for the interior nodes of the hierarchy. We present analytical results to prove the lower bounds of HCSFQ. Our testbed experiments and large-scale simulations show that CSFQ and HCSFQ can provide fair bandwidth allocation and ensure isolation.

1 Introduction

Fair queueing is a canonical mechanism to provide fair bandwidth allocation to network traffic by ensuring that each flow gets its fair share irrespective of the other flows. This way, fair queueing enforces isolation between competing flows, which ensures that normal flows are protected from ill-behaving flows. There is a long history of research on fair queueing [1–12]. Many of the proposed solutions require to maintain per-flow state in the switch, and rely on complex data structures and scheduling algorithms to realize fair queueing.

Core-Stateless Fair Queueing (CSFQ) [13] is a scalable algorithm to realize fair queueing. Compared to the alternatives, CSFQ has the unique property that it does not maintain per-flow state in the network. With CSFQ, the sources or switches at the edge classify traffic into flows and estimate per-flow rate. In turn, the switches in the network estimate the fair rate, and use probabilistic dropping to regulate each flow to its fair rate without maintaining per-flow state.

While CSFQ was proposed more than twenty years ago, it has not taken off. This is primarily due to two reasons. First, it requires changes to the IP protocol (i.e., adding a field to the IP header) and coordination across all switches (routers) in the network. Second, CSFQ requires switches to estimate the fair rate, compute a drop probability, and update the header of each packet. To perform these operations at line rate we need hardware support. These challenges are exacerbated by the fact that routers belong to different, often competing, Internet Service Provides (ISPs), which would all need to cooperate to upgrade their infrastructures to support CSFQ.

However, two emerging technologies are making CSFQ relevant again: (i) the advent of cloud computing and (ii) the increased popularity of programmable switches. Cloud providers own large datacenters consisting of many thousands of servers. Since a datacenter is typically owned by a single administrative entity (cloud provider) that controls both the software and hardware, it is relatively easy for a cloud provider to upgrade all its switches and servers to support CSFQ. FairCloud [14] proposes to apply CSFQ for network isolation in datacenters, but it does not have a hardware implementation for CSFQ. The emergence of programmable switches makes it possible to implement sophisticated packet processing at line rate. In particular, as we will show in this paper, existing programmable switches are powerful enough to support CSFQ at line rate.

While datacenter deployment removes the adoption barriers for CSFQ, it also raises new challenges. In particular, while CSFQ has been designed for a flat hierarchy, the traffic in today's datacenters is naturally structured in a multi-level hierarchy. For example, at the top level we typically have tenants and at the bottom level we have the flows of those tenants. The mechanism of choice to manage such traffic is hierarchical fair queueing [9,10,15], where each non-leaf node distributes its excess bandwidth (i.e., the bandwidths unused by some of its children) across its children. This allocation policy is consistent with a per-tenant payment granularity, i.e., network resources are divided between tenants in proportion to their payments [14]. In this case, if a flow of a tenant stops sending data, that tenant would want to re-allocate the flow's bandwidth to its other flows, and not to the flows of other tenants in the datacenter.

However, implementing hierarchical fair queueing is challenging. Existing solutions require per-flow state, and more importantly, require complex queue management and packet transfers in a hierarchy of queues [9, 10, 15]. Because of the implementation complexity, hierarchical fair queueing is not supported by today's high-speed hardware switches.

To address this challenge, we propose Hierarchical Core-Stateless Fair Queueing (HCSFQ). CSFQ only provides fair queueing, not hierarchical fair queueing. Directly extending CSFQ to support hierarchical fair queueing would require a hierarchy of queues. HCSFQ is able to accurately approximate hierarchical fair queueing and it is highly scalable. The key difference of our approach is that HCSFQ requires only a *single* queue, not a hierarchy of queues. HCSFQ also requires *no* packet scheduling. HCSFQ recursively computes the fair rate of each node starting from the root, and then limits the rate of each flow to its fair share rate. To the best of our knowledge, HCSFQ is the *first* solution that enables hierarchical fair queueing on commodity hardware at line rate while requiring neither per-flow state nor hierarchical queue management.

An important distinction of HCSFQ from CSFQ is that HCSFQ keeps the state of the interior nodes of the hierarchy in the switch. The state of the interior nodes is necessary to support hierarchical fair queueing, as the fair share rates of distinct interior nodes are typically *different*. The excess bandwidth of a flow is *only* shared with its *sibling* flows. That is, if a flow changes its sending rate, it would impact the fair rate of the sibling flows, but not necessarily of other flows in the hierarchy. Note that similar to CSFQ, HCSFQ does not maintain *per-flow* state (i.e., the state of the leaf nodes). Fortunately, for today's multi-tenant clouds, the number of tenants is orders of magnitude smaller than the number of flows, and commodity switches have sufficient on-chip memory to maintain the state for these interior nodes.

We exploit the capability of programmable switching to provide the first realization of CSFQ and HCSFQ on commodity hardware. While conceptually simple, implementing these schedulers on a programmable switch raises several technical challenges. First, they use a complex formula to estimate the rates, which includes several floating-point multiplication, divisions and exponentiation operations. Unfortunately, these operations are not supported by today's programmable switches. To get around this challenge, we leverage high-precision timestamps and a window-based mechanism to estimate these rates. Second, these algorithms rely on probabilistic packet dropping to limit the flows to their fair rates. Unfortunately, probabilistic packet dropping cannot be directly implemented in these switches. We discretize the probability computation to approximate the dropping probability with bounded error. To discretize these probabilities we leverage the switch's random number generator and take advantage of multiple stages. Third, computing the fair rate exhibits a circular dependency. Unfortunately, the switch data plane consists of a multi-stage processing pipeline, and the later stages cannot modify the state in the previous stages. To address it, we judiciously use packet recirculation, and periodically update the fair rate to minimize recirculation overhead.

In summary, we make the following contributions.

- We extend CSFQ to HCSFQ, the first scalable, practical solution to implement hierarchical fair queueing on commodity hardware at line rate with no per-flow state and no hierarchical queue management.
- We exploit the capability of programmable switching ASICs to provide the first data plane design for CSFQ and HCSFQ.
- We implement a prototype of CSFQ and HCSFQ on a Barefoot Tofino Wedge 100BF-65X switch. Our experiments show that CSFQ and HCSFQ can provide fair bandwidth allocation and ensure isolation.

2 Background and Motivation

Our work is motivated by the need for network isolation in multi-tenant datacenters. CSFQ is a scalable solution for fair queueing. We review the background of CSFQ, and identify the opportunities for CSFQ in modern datacenters.

2.1 Core-Stateless Fair Queueing

Fair queueing provides max-min fairness for competing flows. A max-min fair bandwidth allocation is one that any increase of the allocation to some flows would necessarily decrease the allocation of some other flows. The basic way to realize fair queueing in a switch is to keep one queue for each flow and use a scheduling algorithm to pick which queue to dequeue a packet each time. There has been decades of research on fair queueing [1–12]. While we leave the extensive discussion to related work (§7), we emphasize that most solutions are not scalable because of the need to maintain *per-flow* state to classify flows and shape their rates with per-flow queues and complex queue management. As a result, commodity switches only support 10–20 queues.

CSFQ is a *scalable* algorithm to achieve fair queueing with a unique property that it does *not* maintain per-flow state in the network. Figure 1 shows the architecture of CSFQ. CSFQ divides the network into edge and core. The switches

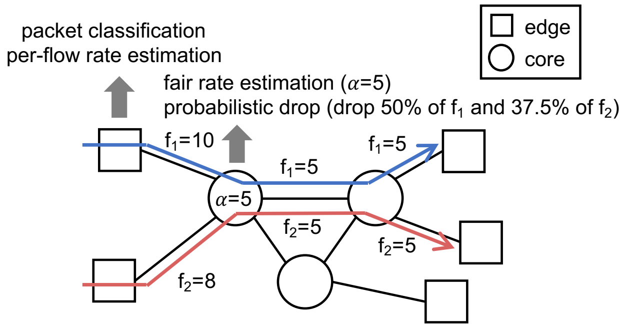
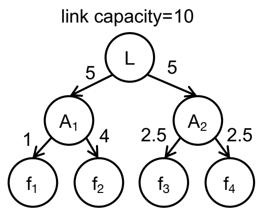


Figure 1: Core-Stateless Fair Queueing.

flow	f_1	f_2	f_3	f_4
arrival rate	1	4	5	5
bandwidth allocation	1	3	3	3

(a) Fair queueing.



(b) Hierarchical fair queueing.

Figure 2: Fair queueing and hierarchical fair queueing.

or hosts at the edge, which do maintain per-flow state, use per-flow state to classify packets into flows and estimate per-flow arrival rate. Then the arrival rate of each flow is *carried* in a custom packet *header*. The switches in the core only estimate the *total* arrival rate of all flows, and then use it to estimate the *fair share rate* with an iterative algorithm. The switches compare the per-flow arrival rate in the packet header with the fair share rate to compute a drop probability, and drop packets to shape the rate of each flow to the fair share rate.

The key benefit of CSFQ is that the complexity (packet classification and flow rate estimation with per-flow state) is moved to the edge, making the core extremely simple. A core switch only maintains the state for *aggregate* variables (total arrival rate, total accepted rate and fair share rate), and only uses *one* queue for packet buffering. More importantly, the complexity of a core switch does *not* change with the number of flows, making the core scale-free.

2.2 Opportunities

CSFQ did not take off because it requires cooperation between ISPs to provide end-to-end isolation for Internet flows, and requires protocol and hardware changes. After twenty years, we believe the time for CSFQ has come because of two opportunities.

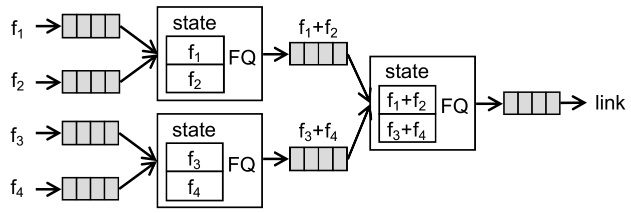
The first opportunity is from cloud computing. Cloud computing has become the fundamental infrastructure of today's Internet. Datacenters power large-scale Internet services we use everyday such as search, social networking and e-commerce, and enterprises are increasingly moving their workloads to the cloud. Fair bandwidth allocation and network isolation for datacenter networks is an important problem [14, 16–28]. While there has been many fair queueing algorithms proposed in the past [1–12], they are rarely deployed in practice because they need to maintain per-flow state in switches but switches can only support 10–20 queues. CSFQ provides a scalable solution to address this problem. Datacenter operators control the entire infrastructure, including both software and hardware. Adopting CSFQ to enforce isolation for datacenter networks naturally eliminates the need of cooperation between different operators or ISPs, as a datacenter network is under a single administrative domain.

The second opportunity is from programmable switching ASICs. Traditional switching ASICs are fixed-function, and adding a new feature like CSFQ requires switch vendors to design a new ASIC. Emerging programmable switching ASICs, such as Barefoot Tofino [29], Broadcom Trident 4 [30] and Cavium XPliant [31], allow users to program the data plane and develop new features. Specifically, to implement CSFQ on a programmable switch, we can program the parser to parse the custom header of CSFQ (to carry per-flow rate), program the match-action tables to implement the CSFQ algorithm, and program the on-chip memory to store the aggregate state. Because a datacenter network is under a single administrative domain, it is easy for the operator to adopt the protocol and hardware changes with programmable switching ASICs.

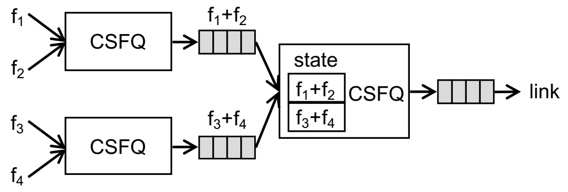
3 Hierarchical Fair Queueing

A multi-tenant cloud has a natural two-layer hierarchy, with the tenants at the first layer and the flows of each tenant at the second layer. Network isolation for multi-tenant datacenters naturally requires hierarchical fair queueing. CSFQ only supports fair queueing, but not hierarchical fair queueing. Hierarchical fair queueing provides fair queueing in a hierarchical manner. Flows are grouped into flow aggregates in multiple layers. The root of the tree includes all the flows. Each node in the tree includes a subset of the flows, called a *flow aggregate*, and fairly allocates its bandwidth to its child nodes. This is done recursively until leaf nodes, each of which contains one flow. The flows are broadly defined, e.g., based on five-tuple or network management considerations. In the case of multi-tenant clouds, it is a two-layer bandwidth allocation. The bandwidth is first allocated to the tenants in the first layer, and then each tenant allocates its bandwidth to its own flows in the second layer.

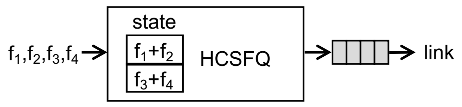
Fair queueing allocates bandwidth fairly to competing flows, and is work conserving, i.e., unused bandwidth share of a flow can be allocated to other flows. The key benefit of hierarchical fair queueing is that it allows unused share of a flow to be allocated to other flows in the same flow aggregate, instead of being shared by all the flows. Fair queueing can be considered as a special case of hierarchical fair queueing that contains only one layer. Two-layer fair queueing for multi-tenant clouds is desirable because the payment is based on



(a) Traditional design of hierarchical fair queueing.



(b) Naive design to extend CSFQ for hierarchical fair queueing.



(c) HCSFQ design.

Figure 3: Comparison of traditional hierarchical fair queueing design, naive design to extend CSFQ, and HCSFQ design.

per tenant. A tenant would want to share its bandwidth only between its own flows, as long as it has sufficient demand.

Example. We use an example in Figure 2 to contrast hierarchical fair queueing with fair queueing. There are four flows, i.e., f_1, f_2, f_3, and f_4. The arrival rates of the four flows are 1, 4, 5, and 5, respectively. The link capacity is 10. With only fair queueing, the unused share of f_1 is evenly allocated to all other three flows. As shown in Figure 2(a), the bandwidth allocation to the four flows is (1, 3, 3, 3). Suppose that f_1 and f_2 are in one flow aggregate (A_1), and f_3 and f_4 are in the other (A_2). With hierarchical fair queueing, the unused fair share of f_1 is only allocated to f_2, instead of also being shared by f_3 and f_4. Figure 2(b) shows the bandwidth allocation with two-layer hierarchical fair queueing, where the flows receive 1, 4, 2.5, and 2.5, respectively.

Challenge. Hierarchical fair queueing is known to be challenging to realize in switches at high speed. A traditional design to support hierarchical fair queueing is to leverage a hierarchy of queues, and each node in the hierarchy implements fair queueing for the queues of its child nodes. Figure 3(a) shows an example of such a design to support the two-layer hierarchy in Figure 2(b). This design has two major problems. First, the amount of state and the number of queues needed by this design is proportional to the number of nodes in the hierarchy. It needs to maintain per-flow state and the state of each interior node in the tree. Second, the design involves complex queue management with a hierarchy of queues, as packets need to be moved between queues in different layers. CSFQ does not require maintaining per-flow state, but naively extending CSFQ to support hierarchical fair queueing would still require a hierarchy of queues as shown in Figure 3(b). These two factors together make the design hard to scale to support a large number of flows. As a result, hierarchical fair queueing is not supported by today's high-speed switches.

4 HCSFQ Design

We propose Hierarchical Core-Stateless Fair Queueing (HCSFQ), which generalizes CSFQ to support hierarchical fair queueing. HCSFQ is the first scalable solution that enables hierarchical fair queueing on commodity hardware at line rate without per-flow state and complex hierarchical queue management.

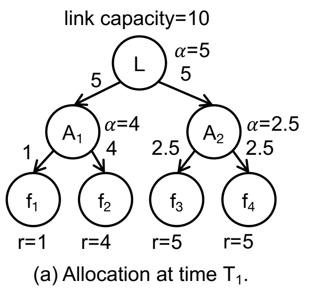
We give a high-level overview of HCSFQ in Figure 3(c). In contrast to the traditional design in Figure 3(a), HCSFQ has two unique properties: (i) it does not maintain per-flow state, but only keeps the state of interior nodes; (ii) it does not require a hierarchy of queues, but only uses one queue. These two properties together dramatically simplify the design, making HCSFQ amenable to be implemented on high-speed switches under strict timing and resource constraints.

The major distinction between HCSFQ and CSFQ is that HCSFQ needs to maintain the state of interior nodes. This is necessary because HCSFQ aims to provide hierarchical fair bandwidth allocation for a flow hierarchy. Note that the naive design of extending CSFQ in Figure 3(b) also requires maintaining the state of interior nodes. In fair queueing, CSFQ only requires to keep one fair share rate, which is the same for *all* flows. But in hierarchical fair queueing, the fair share rates for different flows can be *different* if two flows are not siblings (i.e., do not have the same parent node). If a flow changes its rate, it would affect the fair share rate of its sibling flows, but not necessarily those of non-sibling flows. Figure 4 illustrates this with a concrete example. There is a two-layer hierarchy with four flows. At time T_1, the arrival rates for the four flows are 1, 4, 5, and 5 (the same as Figure 2). The fair share rate at L is 5, and those at A_1 and A_2 are 4 and 2.5. Then at time T_2, f_1 increases its arrival rate from 1 to 2. Then under fair bandwidth allocation, the new fair share rate for the subtree under A_1 becomes 3, so that f_1 receives 2 and f_2 receives 3. The rate change of f_1, however, does not effect the fair share rate for f_3 and f_4. This is because f_3 and f_4 are not sibling nodes of f_1.

CSFQ can be considered as a special case of HCSFQ which contains only one layer, and as such, it only carries the state for one interior node—the root.

4.1 Fluid Model

We first use a fluid model to formalize hierarchical fair queueing. The fluid model considers a switch with output link capacity C, and the flows are modeled as a continuous stream of bits. The flow hierarchy is represented as a directed graph $G(V,E)$, where V is the set of nodes and E is the set of edges.



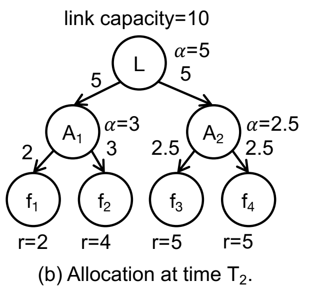


Figure 4: The flow arrival rates change from T_1 to T_2. It is necessary for the switch to keep the state for the interior nodes of the hierarchy in order to realize hierarchical fair queueing.

A node $v \in V$ represents a flow aggregate (i.e., a set of flows), where $r(v)$ is the arrival rate of the flow aggregate and $c(v)$ is the capacity allocated to v. A directed edge $e(v,u) \in E$ represents that u is a child of v.

Max-min fair bandwidth allocation ensures that the flows that are bottlenecked by a link receives the same output rate, which we call the fair share rate. Let $\alpha(v)$ be the fair share rate that node v allocates to its children. If max-min fair bandwidth allocation is achieved, for a child node u of node v, the flow aggregate at u receives a bandwidth allocation of $c(u) = min(r(u), \alpha(v))$. The arrival rate of v is the sum of the arrival rates of its children, i.e., $r(v) = \sum_{e(v,u)\in E} r(u)$. If $r(v) > c(v)$, the arrival rate of v exceeds the capacity allocated to v, and the fair rate $\alpha(v)$ is the unique solution to

$$c(v) = \sum_{e(v,u)\in E} \min(\alpha(v), r(u)). \tag{1}$$

If $r(v) \le c(v)$, the arrival rate of v is no more than the capacity allocated to v, and all flows in v can be forwarded without dropping packets. In this case, by convention we have

$$\alpha(v) = \max_{e(v,u)\in E} r(u). \tag{2}$$

The fair rate computation is done recursively from the root to the leaf nodes. When v is the root, we have $c(v) = C$, where C is the link capacity. Then starting from the root, we can compute $c(v)$ and $\alpha(v)$ for each node in the tree.

Based on this fluid model, there is a simple algorithm to achieve max-min fair bandwidth allocation. In this algorithm, we first use the recursive computation to compute $\alpha(v.parent)$ for each leaf node v, which is the fair share rate allocated by v's parent to v. If $r(v) \le \alpha(v.parent)$, then no bits need to be dropped; otherwise, a fraction of $(r(v) - \alpha(v.parent))/r(v)$ need to be dropped. Therefore, achieve max-min fair bandwidth allocation, each incoming bit of the flow in v is dropped by probability

$$\max(0, 1 - \frac{\alpha(v.parent)}{r(v)}). \tag{3}$$

4.2 HCSFQ Algorithm

The HCSFQ algorithm realizes the conceptual fluid algorithm in a real switch. Similar to CSFQ, HCSFQ does not maintain per-flow state, and only requires a single FIFO queue for packet buffering (Figure 3). The algorithm relies on two building blocks from CSFQ, which are arrival rate estimation and fair share rate estimation, and applies them recursively to compute the fair share rate for each leaf node.

Arrival rate estimation. The arrival rate estimation is used to estimate the arrival rate of a flow aggregate for a node in the hierarchy. Like CSFQ, it uses the canonical exponential averaging mechanism in networking for rate estimation. Let t_i and l_i be the arrival time and length of the i^{th} packet of the flow aggregate in node v. We use $r(v)$ to denote the estimated arrival rate of v. It is updated each time a new packet of v arrives, based on the following equation,

$$r(v)_{new} = (1 - e^{T_i/K})\frac{l_i}{T_i} + e^{T_i/K} r(v)_{old}, \tag{4}$$

where $T_i = t_i - t_{i-1}$ and K is a constant.

Fair share rate estimation. The fair share rate estimation is used to estimate the fair share rate that a node allocates to its children. The capacity of node v is $c(v)$. Eq.(4) gives the arrival rate of the node $r(v)$. If $r(v) \le c(v)$, then $\alpha(v)$ is calculated using Eq.(2). Otherwise, $\alpha(v)$ should be the unique solution to Eq.(1). We apply the iterative algorithm in CSFQ to approximately solve the equation. Specifically, for each node v, we maintain the accepted rate estimation $f(v)$, which is updated with Eq.(4) if the packet is not dropped. Then, $\alpha(v)$ is approximately computed with the following formula,

$$\alpha(v)_{new} = \alpha(v)_{old}\frac{c(v)}{f(v)}. \tag{5}$$

Note that the computation of $\alpha(v)$ is iterative. It converges to the solution of Eq.(1) after several iterations, i.e., processing several packets. Similar to CSFQ, HCSFQ also uses a window of size K_c to account for inaccuracies introduced by exponential averaging in rate estimation. That is, $\alpha(v)$ is updated only if the node is congested ($r(v) > c(v)$) or uncongested ($r(v) \le c(v)$) for an interval of length K_c.

Packet state. A packet *pkt* carries two pieces of state in the packet header, which are *pkt.r* and *pkt.nodes*.

- *pkt.r* is the arrival rate estimate of the flow the packet belongs to.
- *pkt.nodes* is a list of node IDs that indicate the flow aggregates the packet belongs to in the flow hierarchy, excluding the leaf. For example, in Figure 2, if a packet *pkt* belongs to f_1 or f_2, then $pkt.nodes = [L, A_1]$.

CSFQ only carries *pkt.r* in the packet header as there is no flow hierarchy. HCSFQ additionally carries *pkt.nodes* to track the set of flow aggregates the packet belongs to in the hierarchy. Similar to CSFQ, both *pkt.r* and *pkt.nodes* are

Algorithm 1 HCSFQ(pkt)

```
1:  $cur\_\alpha \leftarrow 0$
2:  for $v \in pkt.nodes$ do
        // estimate arrival rate
3:      $r[v] \leftarrow estimate\_rate(pkt)$
4:      $cur\_\alpha \leftarrow \alpha[v]$
    // calculate drop probability
5:  $prob \leftarrow max(0, 1 - cur\_\alpha/pkt.r)$
6:  if $prob > rand(0,1)$ then
7:      $drop\_flag \leftarrow TRUE$
8:  for $v \in pkt.nodes$ do
        // estimate accepted rate
9:      if $drop\_flag$ is $False$ then
10:         $f[v] \leftarrow estimate\_rate(pkt)$
        // allocate bandwidth
11:     if $v$ is root then
12:         $c[v] \leftarrow$ link capacity
13:     else
14:         $c[v] \leftarrow min(\alpha[v.parent], r[v])$
        // update fair share rate
15:     if $r[v] > c[v]$ then
16:         if $congest\_flag[v]$ is $FALSE$ then
17:             $congest\_flag[v] \leftarrow TRUE$
18:             $start\_time \leftarrow current\_time$
19:         else if $current\_time - start\_time > K_c$ then
20:             $\alpha[v] \leftarrow \alpha[v] \cdot c[v]/f[v]$
21:             $start\_time \leftarrow current\_time$
22:     else
23:         if $congest\_flag[v]$ is $TRUE$ then
24:             $congest\_flag[v] \leftarrow FALSE$
25:             $start\_time \leftarrow current\_time$
26:             $tmp\_\alpha[v] \leftarrow 0$
27:         else if $current\_time - start\_time \le K_c$ then
28:             $child\_r \leftarrow v.next = NULL\ ?\ pkt.r : r[v.next]$
29:             $tmp\_\alpha[v] \leftarrow max(tmp\_\alpha[v], child\_r)$
30:         else
31:             $\alpha[v] \leftarrow tmp\_\alpha[v]$
32:             $start\_time \leftarrow current\_time$
33:             $tmp\_\alpha[v] \leftarrow 0$
34:     $cur\_\alpha \leftarrow \alpha[v]$
    // drop or enqueue pkt
35: if $drop\_flag$ then
36:     $drop(pkt)$
37: else
38:     $enqueue(pkt)$
    // update the packet rate
39: $pkt.r \leftarrow min(cur\_\alpha, pkt.r)$
```

inserted at the edge. An edge switch (e.g., a software switch, a NIC or a ToR switch in datacenter networks) performs packet classification to get $pkt.nodes$, and uses Eq.(4) to estimate the flow rate $pkt.r$. Both $pkt.r$ and $pkt.nodes$ are transparent to end hosts and are removed by the switch at the last hop.

Hierarchical computation. The main difference between HCSFQ and CSFQ is that HCSFQ performs fair share rate estimation *recursively* in a hierarchical manner. In CSFQ, the arrival rate estimation for each flow is done at the edge, and a core switch only estimates the total arrival rate. In HCSFQ, because there is a hierarchy of flow aggregates, a core switch additionally estimates the arrival rate for each flow aggregate (i.e., the internal nodes in the tree). Similarly, in CSFQ, a core switch only calculates a fair share rate for the link, while in HCSFQ, a core switch additionally calculates a fair share rate for each flow aggregate. Importantly, the fair share rate estimation in HCSFQ is used to *bridge* the computation of different layers together. That is, for node v, the allocated bandwidth $c(v)$ is used to estimate the fair share rate $\alpha(v)$, which is then used to compute the allocated bandwidth of its children, i.e., $c(u)$ for $u \in v.children$, in the next layer.

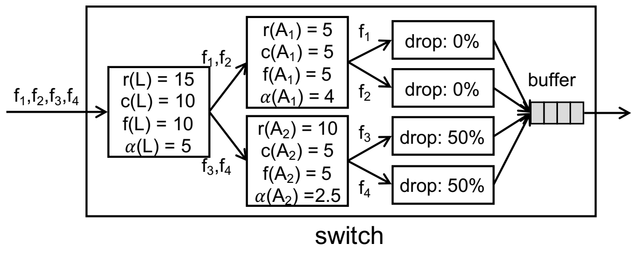


Figure 5: Example of the HCSFQ algorithm to provide hierarchical fair queueing for the scenario in Figure 2(b).

Algorithm 1 shows the pseudo code of the HCSFQ algorithm. When a packet pkt arrives at the switch, the switch updates the arrival rate estimate for each flow aggregate the packet belongs to using Eq.(4), and gets the fair share rate of the flow (line 1-4). Then the switch computes the dropping probability based on Eq.(3) and decides whether to drop the packet (line 5-7). After this, the switch recursively updates the fair share rate of each flow aggregate in the hierarchy (line 8-34). Based on whether the packet is dropped, the switch updates the accepted rate estimate for each flow aggregate (line 9-10). If node v is the root, then all flows are under this node, and its allocated capacity is the link capacity (line 11-12); otherwise, its allocated capacity is the max of the fair share rate allocated by its parent and its arrival rate (line 13-14). If the arrival rate of v is bigger than its allocated capacity, then the node is congested, and the fair share rate is updated based on Eq.(5) (15-21); otherwise, the fair share rate is the max arrival rate of its children, i.e., based on Eq.(2) (line 22-33). Note that we use a window of length K_c for fair share update to account for inaccuracies in rate estimation. Based on the dropping decision, the switch drops or enqueues the packet (line 35-38). Finally, the arrival rate $pkt.r$ is updated and will be used by the next-hop switch (line 39). Note that the loops (line 2-4 and line 8-34) are done in one pass and the fair share rate is updated based on $c[v.parent]$ from the last round.

Figure 5 illustrates how the algorithm works to realize hierarchical fair queueing for the example in Figure 2. At the root, the total arrival rate of all flows $r(L)$ is 15, and the capacity $c(L)$ is the link capacity 10, which is below the arrival rate. The root fairly allocates the capacity to the two flow aggregates, A_1 and A_2. The figure shows the stable state when the accepted rates and fair share rates of all the nodes have converged. After convergence, the accepted rate $f(L)$ is 10, and the fair share rate $\alpha(L)$ is 5. At node A_1, the arrival rate $r(A_1)$, which is the sum of $r(f_1)$ and $r(f_2)$, is 5, and the allocated capacity $c(A_1)$ is 5. The fair share rate is set as 4, and there is no need to drop packets for f_1 and f_2. At node A_2, the arrival rate $r(A_2)$, which is 10, is bigger than the allocated capacity, which is 5. A_2 allocates its capacity to f_3 and f_4 fairly. Each receives a fair share rate of 2.5. So the switch drops 50% of the packets for both f_3 and f_4.

Weighted HCSFQ. The HCSFQ algorithm can be extended to support flows and flow aggregates with weights. For node v, we use $w(v)$ to represent the weight of the flow or flow aggregate of v. Under max-min fair bandwidth allocation, competing flows or flow aggregates at the bottlenecked link have the same fair share rate $r(v)/w(v)$. There are two changes to the algorithm in order to incorporate the weight. The first change is on the equation to compute the fair rate $\alpha(v)$ when $r(v) > c(v)$. Eq.(1) is changed to

$$c(v) = \sum_{e(v,u)\in E} w(u) \cdot min(\alpha(v), \frac{r(u)}{w(u)}). \tag{6}$$

The second change is on the equation to compute the drop probability. Eq.(3) is changed to

$$\max(0, 1 - \alpha(v.parent) \cdot \frac{w(v)}{r(v)}). \tag{7}$$

4.3 Theoretical Guarantee

We have the following theorem to provide the theoretical guarantees for HCSFQ. The proof of the theorem is in Appendix.

Theorem 1. *Consider a link with a hierarchical fair queueing policy and a flow in the hierarchy. Let w_1, w_2, ..., w_n be the weights of the nodes from the root to the flow. Let α_1, α_2, ..., α_n be the constant normalized fair rate of the nodes from the root to the flow. Let $r_{\alpha_i} = \alpha_i w_i$. If probabilistic dropping is applied at the last layer, then the excess service received by the flow that sends at a rate at no larger than R, is bounded above by*

$$r_{\alpha_n} K(1 + ln\frac{R}{r_{\alpha_n}}) + l_{max} \tag{8}$$

where l_{max} is the maximum packet length.

Consider a parent and its children in the hierarchy. Let the number of children be k. Let $r_{\alpha'}$ be the weighted fair rate of the parent, and $r_\alpha^{(j)}$ be the weighted fair rate of the j-th child. Suppose the inter-arrival time of every packet is at least τ, and

$$r_{\alpha'} \geq \frac{1}{1 - e^{-\tau/K}} \sum_{j=1}^{k} r_\alpha^{(j)}.$$

The the parent node does not drop packets.

Remark. The first conclusion bounds the excess service that can be received by a flow. The second conclusion provides the theoretical condition for only performing probabilistic dropping at the leaf node.

5 Data Plane Design and Implementation

In this section, we describe a data plane design to implement CSFQ and HCSFQ on new-generation programmable switches. Programmable switches enable users to program the multi-stage match-action pipeline in the switch data plane to implement custom features. Users can also access the on-chip memory and implement stateful operations using the register arrays provided by programmable switches. Programmable switches also support a set of primitive actions (e.g., recirculate, bit shift, add and subtract) which make HCSFQ possible. Based on the constructs of programmable switches, we show how to design and implement the rate estimation, the fair rate computation and the flow shaping logic (i.e., Algorithm 1) on programmable switches. Our HCSFQ implementation contains 1952 lines of code in P4 and is compiled to Barefoot Tofino ASIC [29]. The code is open-source and available at `https://github.com/netx-repo/HCSFQ`.

5.1 Single Layer

We first describe how to implement CSFQ, i.e., single-layer HCSFQ, which is used as a building block to implement multi-layer HCSFQ. There are three challenges to implement single-layer HCSFQ on programmable switches: rate estimation, probabilistic drop, and fair rate update. We describe each challenge and its solution as follows.

Rate estimation. The switch needs to estimate two rates: the total arrival rate r, and the accepted rate f. Both rates are estimated with Eq.(4). Because switches have strict timing and resource requirements, an action in a match-action table can only contain a small number of operations in a limited operation set. The equation cannot be directly implemented in the switch data plane due to two reasons. First, the equation involves several multiplication, division and exponentiation operations on floating points. These operations are quite complex and require multiple clock cycles to compute. As such, they are not typically supported by the switch data plane. Second, a rate (r or f) is stored in a register of the on-chip memory. To update the rate, the switch needs to read the rate from the register, uses the equation to calculate the new rate, and then updates the register. A register can only be accessed by its own stage, but the equation includes multiple arithmetic operations, which requires multiple stages to compute.

We leverage the high-precision timestamps available in the data plane, and use a window-based mechanism to estimate the rates. Programmable switches are able to provide high-precision timestamps at the granularity of one nanosecond. To estimate a rate, the switch maintains a pair of registers ($reg.byte$ and $reg.start$). One register ($reg.byte$) stores the total bytes of packets the switch has received in the current window. The other register ($reg.start$) stores the start timestamp of the current window. For each incoming packet, the switch first checks the current timestamp and compares it with *reg.start* to see if the packet belongs to the current window. If so, the switch adds the size of the packet to *reg.byte*; otherwise, the switch clears *reg.byte* and sets *reg.start* to the current timestamp. The switch keeps another register *reg.rate* to store the current rate estimate. When a window is passed, the switch uses *reg.byte* to update *reg.rate*, which can be done with either a direct assignment, or a moving average. Our experiments indicate that using a moving average (implemented with several bit shift and addition operations) works better and avoids oscillation with the control loop that updates the fair share rate and drops packets.

The key benefit of this window-based mechanism is that because the switch can provide nanosecond-granularity timestamps, we can use a small window size to accurately estimate flow rate and capture sudden packet bursts. It is important to note that the rate estimation is local to the switch and only uses timestamps to divide time into windows. So there is no need for time synchronization between switches.

Probabilistic drop. Probabilistic drop is used to regulate the flows to the fair share rate. The switch uses the fair share rate α and the flow arrival rate r to compute the probability to drop packets of the flow (Eq.(3) and line 5 in Algorithm 1). Then the switch checks the condition $max(0, 1-\alpha/r) > rand(0,1)$ to decide whether to drop an incoming packet or not. Similar to rate estimation, the challenge is that switches do not support the division operation to compute the probability. One way to solve the problem is to use a similar window-based mechanism as rate estimation, i.e., divide time into windows with window size δ, and keep counters to allow up to $r\delta$ packets to pass in each window and drop all remaining packets. The drawback of this approach is that it introduces bursty packet drops, which do not work well with congestion control. We want to mimic the behavior of CSFQ to have random packet drops that are uniformly distributed in the packet stream.

We discretize the probability computation to approximate the drop probability with bounded error. We leverage the random number generator provided by the data plane and use multiple stages to realize the discretized computation. Specifically, to check the condition $max(0, 1-\alpha/r) > rand(0,1)$, it is sufficient to check $rand(0,1) > \alpha/r$. We multiply r to both sides of the inequality, and transform the condition to

$$rand(0,r) > \alpha.$$

If the switch can generate a random number between 0 and r, then we can simply compare the generated random number and α to decide whether to drop a packet. However, some switches can only generate a random number in a range of a power of two, i.e., in $[0, 2^n-1]$, where n is a given value at compilation time and cannot be a variable. One possible solution is to use a large value for n at compilation time and use $rand(0, 2^n-1)\%r$ to approximate $rand(0,r)$. But the modulo operation on an arbitrary number may not be supported, and the generated numbers are not uniformly distributed in $[0,r]$. We solve this problem by discretizing the probability computation. We use an integer, instead of a floating point, for the probability. We convert the condition to

$$rand(0, 2^n-1)\cdot r > (2^n-1)\cdot\alpha.$$

While multiplication is not directly supported, we can convert a multiplication operation into several bit shift and addition operations. Since n is small and one stage can do multiple operations, a multiplication can be done in a few stages. This solution introduces errors because the random number is an integer in $[0, 2^n-1]$, instead of a real number in $[0,1]$. However, the error is bounded by $1/2^n$, which reduces exponentially with n. When n is 7, the error introduced by the approximation is bounded by 1/128, which is smaller than 1%.

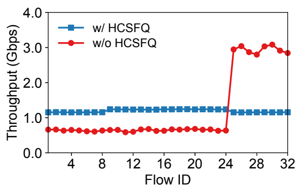
Fair rate update. When the link is congested, the fair share rate is the unique solution to Eq.(1). Because HCSFQ does not maintain per-flow state, it uses $\alpha_{new} = \alpha_{old}c/f$ (Eq.(5)) to approximately compute the fair share rate, where c is the capacity and f is the accepted rate. Like rate estimation and probabilistic drop, Eq.(5) cannot be supported because it contains multiplication and division. What is more challenging is that the fair rate update introduces the following circular dependency to the packet processing.

$$\text{read } \alpha \rightarrow \text{enqueue/drop} \rightarrow \text{update } f \rightarrow \text{update } \alpha$$

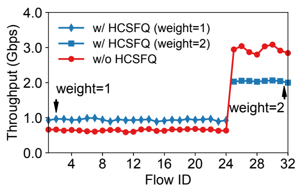
Specifically, the switch needs to read α to compute the drop probability. Then based on whether to enqueue or drop a packet, the switch updates the accepted rate f, which is then used to update α. Because a register can only be accessed by its own stage, the new value of α cannot be used to update the register that stores α in a previous stage.

To address these two problems, we first observe that the update equation $\alpha_{new} = \alpha_{old}c/f$ in HCSFQ is already an approximation, and the correct α is iteratively computed after several updates until f converges to c. As such, we replace the update equation with an additive-increase multiplicative-decrease method, which increases or decreases α each time if f is not equal to c. This ensures that the value for α converges to the correct value. Note that in the original CSFQ, α is also computed iteratively to converge to the correct value.

To address the circular dependency, we leverage packet recirculation available in programmable switches, and let the recirculated packets carry the new value of α to update the register for α in a previous stage. Switches have limited bandwidth for recirculation. We judiciously use packet recirculation to minimize recirculation overhead. We follow the same scheme as CSFQ: update α only when the node is congested or uncongested for a window length of K_c. Given the window size K_c, α is updated by at most $1/K_c$ times per second. As a concrete example, let K_c be 10 μs. Then α is updated by at most 100K times per second, and the amount of recirculation traffic is only a tiny fraction of the switch capacity.



(a) Equal weights.



(b) Different weights.

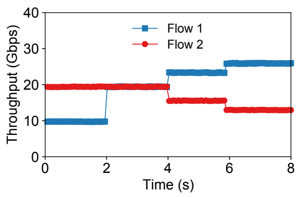
Figure 6: Testbed experiments of fair queueing for UDP. Flow 1–24 send at 2Gbps and Flow 25-32 send at 8Gbps.

5.2 Multiple Layers

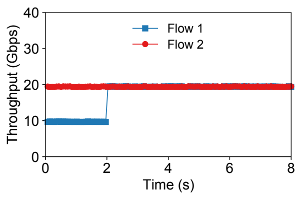
The single-layer design is used as a building block to support multiple layers. As shown in Algorithm 1 and Figure 5, the processing of HCSFQ on a packet is performed layer by layer, from the root to the leaf node. This well matches the multi-stage packet processing pipeline of programmable switches. The layers in HCSFQ can be mapped to the stages in the pipeline, which naturally processes packets sequentially stage by stage. The major difference between HCSFQ and CSFQ is that HCSFQ needs to store more states as it has multiple layers. CSFQ is a single-layer HCSFQ and only maintains the state for three variables, which are the total arrival rate r, the accepted rate f, and the fair share rate α. Each variable use multiple registers as described in §5.1. HCSFQ maintains the state for all interior nodes, each of which includes the three variables. Commodity switches have 10-100 MB on-chip memory [32], which is able to support a large number of interior nodes. For a two-layer HCSFQ for tenant-level and flow-level isolation in multi-tenant datacenters, a switch needs to maintain per-tenant state, but not per-flow state. With 10-100 MB memory, the switch can support millions of tenants. In terms of the number of layers, our prototype supports up to four layers on Barefoot Tofino. There is no theoretical limit on the number of layers given the scalable algorithm design. The constraints for practical implementations mainly come from the restricted hardware primitives to implement the algorithm as we describe in §5.1. These constraints are not fundamental. Newer programmable switches (e.g., Barefoot Tofino 2) have more stages and provide more hardware primitives to support more layers. Despite this, we expect HCSFQ with 2–4 layers should be sufficient to provide hierarchical isolation for many datacenter scenarios (e.g., multi-tenancy).

6 Evaluation

In this section, we provide experimental results to demonstrate the performance of HCSFQ. We first evaluate the performance of single-layer HCSFQ (i.e., CSFQ), and show that it can provide fair queueing (§6.1). We then evaluate the performance of two-layer HCSFQ, and show that it can provide hierarchical fair queueing to enforce tenant-level and flow-level isolation for multi-tenant datacenters (§6.2). Finally, we use simulations to evaluate HCSFQ in a large-scale datacenter environment and compare it with several alternatives (§6.3).

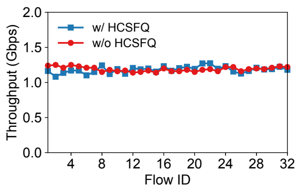


(a) w/o HCSFQ.

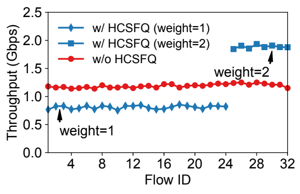


(b) w/ HCSFQ.

Figure 7: Testbed experiments of fair queueing for UDP. Flow 1 is sending at a different rate every 2 seconds. Flow 2 is sending at 20Gbps.



(a) Equal weights.



(b) Different weights.

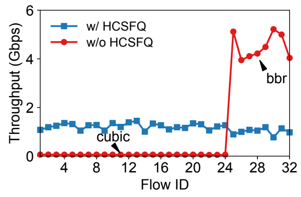
Figure 8: Testbed experiments of fair queueing for TCP.

All testbed experiments are conducted on a hardware testbed with a Barefoot Tofino Wedge 100BF-65X switch. Each server is configured with an 8-core CPU (Intel Xeon E5-2620 @ 2.1GHz), 64GB memory and one 40G NIC (Intel XL710), and runs Ubuntu 16.04.6 LTS with Linux kernel 4.10.0-28-generic. Our switch implementation contains both the edge and core functionalities for HCSFQ. Therefore, our prototype provides hierarchical fair queueing *without* modifications to either the software or hardware of the end hosts. By default, we use TCP Cubic provided by the Linux kernel.

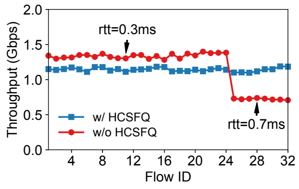
6.1 Fair Queueing Experiments

We first evaluate the capability of HCSFQ to provide fair queueing. Fair queueing requires one-layer HCSFQ. We cover both UDP and TCP traffic with equal or different weights. In the experiments, we use four servers as the senders and one server as the receiver. Each sender sends 8 flows (based on five-tuple), and a total of 32 flows are sent to a receiver. All servers are connected to the switch with 40Gbps links. The bottleneck link is the link between the switch and the receiver.

UDP. If all UDP flows have the same sending rate, they would get similar bandwidth under the tail-drop FIFO queue in the switch. To make the experiment more interesting, we assign different sending rates to the UDP flows. We let 24 flows (Flow 1–24) send at 2Gbps and 8 flows (Flow 25–32) send at 8Gbps. As shown in Figure 6(a), without HCSFQ, Flow 25–32 obtain higher bandwidth than Flow 1–24 because Flow 25–32 have larger sending rates. In comparison, HCSFQ is able to fairly allocate bandwidth to the flows.

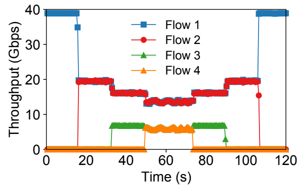


(a) Different TCP algorithms.

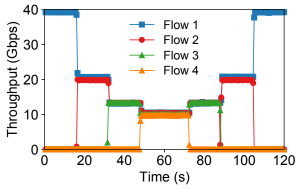


(b) Different RTTs.

Figure 9: Testbed experiments of fair queueing for TCP under different configurations.



(a) Without HCSFQ.



(b) With HCSFQ.

Figure 10: Testbed experiments of UDP convergence. Flow 1 and 2 send at 40Gbps, and Flow 3 and 4 send at 20Gbps.

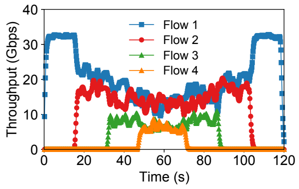
HCSFQ supports weighted fair queueing. We assign weight 1 to Flow 1–24 to and weight 2 to Flow 25–32. As shown in Figure 6(b), without HCSFQ, the result is the same as that with equal weights in Figure 6(a). On the other hand, HCSFQ is able to allocate the bandwidth based on the weights. Flow 25–32 achieve higher throughput than Flow 1–24.

We also evaluate HCSFQ when the UDP flows dynamically change their rates. We let Flow 1 send at a different rate every 2 seconds (10Gbps, 20Gbps, 30Gbps and 40Gbps, respectively) and let Flow 2 keep sending at 20Gbps. Without HCSFQ, when the link is congested (from 4s to 8s), each flow achieves a throughput in proportional to its sending rate. With HCSFQ, two flows get the fair share (20Gbps) when the link is congested.

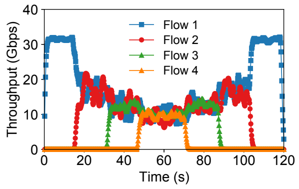
TCP. Figure 8(a) shows the throughput of the flows with and without HCSFQ. Because TCP congestion control provides fair bandwidth allocation, the flows have similar throughput even without HCSFQ. Adding HCSFQ to the switch does not change the bandwidth allocation and thus has a similar result.

However, TCP cannot support weighted fair queueing. To show the benefits of HCSFQ, we let Flow 1–24 have weight 1 and Flow 25–32 have weight 2. Without HCSFQ, the result in Figure 8(b) is similar to that in Figure 8(a). With HCSFQ, the flows get bandwidth in proportional to their weights. The flows with higher weights (Flow 25–32) receive more bandwidth than those with lower weights (Flow 1–24).

Different TCP algorithms. There are many TCP congestion control algorithms. Without in-network enforcement, the flows using aggressive congestion control algorithms would get more bandwidth. In this experiment, we let Flow 1–24 use TCP Cubic (provided by default in Linux) and Flow 25–32

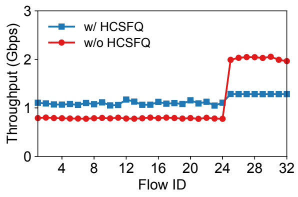


(a) Without HCSFQ.

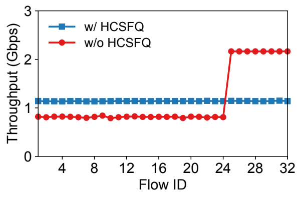


(b) With HCSFQ.

Figure 11: Testbed experiments of TCP convergence. Flow 1 and 2 have 0.3ms RTT, and Flow 3 and 4 have 0.7ms RTT.



(a) Testbed experiment.



(b) Packet-level simulation.

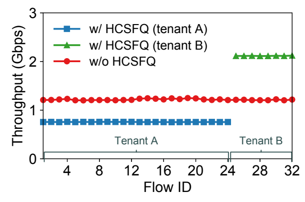
Figure 12: Evaluation result of mixed TCP and UDP traffic.

use TCP BBR. As shown in Figure 9(a), without HCSFQ, because TCP BBR is more aggressive than TCP Cubic, the flows with TCP BBR get almost all the bandwidth. On the other hand, HCSFQ is able to provide fair queueing, regardless of the TCP algorithms they use. We have also tried TCP Reno, which performs similar to TCP Cubic.

Different RTTs. In this experiment, we increase the RTT of Flow 25–32 by 0.4 ms using Linux Traffic Control (Linux tc). The default RTT measured by ping in the testbed, i.e., the RTT of Flow 1–24, is 0.3 ms (mostly host overhead). The TCP throughput is inverse proportional to RTT [33]. In our case, the flows with 0.3 ms RTT (Flow 1–24) should have $0.7/0.3 \approx 2\times$ higher bandwidth than the flows with 0.7 ms RTT (Flow 25–32), which is close to what we see in Figure 9(b). On the other hand, HCSFQ is able to provide fair queueing even when the flows have different RTTs.

Convergence. We let four flows from different clients join and leave a link every 16 seconds to evaluate convergence. Figure 10 shows the UDP result. Flow 1 and 2 send at 40Gbps (using DPDK [34]), and Flow 3 and 4 send 20Gbps. When HCSFQ is enabled, the four flows quickly converge to a similar rate, even though they have different sending rate. Figure 11 shows the TCP result. We set the RTTs of Flow 3 and 4 to 0.7ms using Linux tc, and the RTTs of of Flow 1 and 2 are around 0.3ms by default. With HCSFQ, the four flows quickly converge to a similar rate, regardless of different RTTs.

Mixed UDP and TCP traffic. We evaluate HCSFQ under a mixed workload with both UDP and TCP traffic, and consider the impact of ill-behaved UDP flows on TCP flows. In the experiment, Flow 1–24 are TCP flows, and Flow 25–32 are UDP flows that send at 3.2Gbps. As shown in Figure 12(a), without HCSFQ, because UDP flows are not affected by TCP


(a) Equal weights.

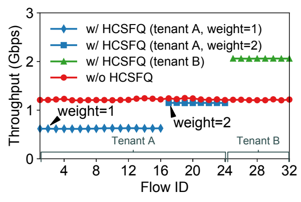
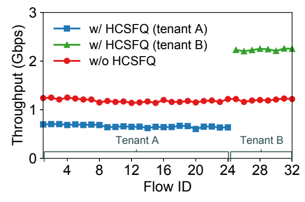

(b) Different weights.

Figure 13: Testbed experiments of hierarchical fair queueing for UDP. Two tenants should have the same *total* throughput.


(a) Equal weights.

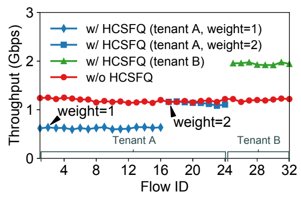
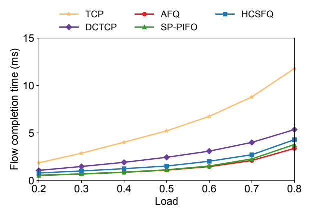

(b) Different weights.

Figure 14: Testbed experiments of hierarchical fair queueing for TCP. Two tenants should have the same *total* throughput.


(a) Average flow completion time for flows less than 100KB.

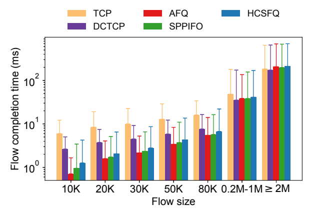
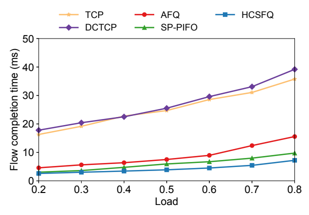

(b) Flow completion time (avg. and 99th) breakdown for 70% load.

Figure 15: Simulation result under the web search workload.


(a) Average flow completion time for flows less than 100KB.

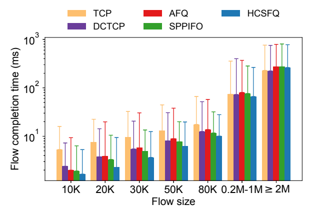

(b) Flow completion time (avg. and 99th) breakdown for 70% load.

Figure 16: Simulation result under the web search workload with injected UDP traffic.

congestion control, Flow 25–32 get 84% higher throughput than their fair share. In comparison, HCSFQ is able to allocate bandwidth fairly between all flows.

Gap between prototype implementation and theoretical algorithm. Although the above experiment demonstrates the effectiveness of HCSFQ on protecting TCP flows from aggressive UDP flows, there is still a small gap from the theoretical upper bound. Figure 12(b) shows the simulation result on the same setup using a packet-level simulator Netbench [35]. In the simulation, the TCP and UDP flows get almost identical throughput with HCSFQ. The reason for the gap between Figure 12(a) and Figure 12(b) is that to realize HCSFQ on a real switch, we make several approximations described in §5. These approximations cause extra jitters for TCP flows, and UDP flows occupy the spare bandwidth caused by the jitters and obtain higher throughput. We believe as programmable switches get more capable, these approximations can be removed to enable more accurate implementation of HCSFQ in the future.

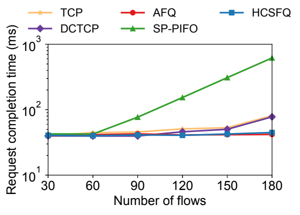
6.2 Hierarchical Fair Queueing Experiments

We now evaluate the capability of HCSFQ to provide hierarchical fair queueing. We show that two-layer HCSFQ can provide tenant-level and flow-level isolation for multi-tenant datacenters. Similar to the previous experiments, we use 4 servers to send a total of 32 flows to a receiver. To evaluate hierarchical fair queueing, we let tenant A contain 24 flows (Flow 1–24) and tenant B contain 8 flows (Flow 25-32).

UDP. We set the sending rates of all 32 UDP flows to 8 Gbps. As shown in Figure 13(a), without HCSFQ, the flows have similar throughput. Because tenant A has three times as many flows as tenant B, the total throughput of A is three times as that of B. With HCSFQ, two tenants get the same total throughput. Because A has more flows, each flow in A has lower throughput than that in B.

To evaluate weighted hierarchical fair queueing, we assign different weights to tenant A's flows. We let Flow 1–8 have weight 2 and Flow 9–24 have weight 1. We assign the same weight to tenant A and B. As shown in Figure 13(b), the result without HCSFQ is the same as it in Figure 13(a). All flows receive the same bandwidth, regardless of tenants and weights. With HCSFQ, because the two tenants have the same weight, the bandwidth allocation to the two tenants stays the same. In tenant A, a flow with weight 2 has double throughput as a flow with weight 1. In tenant B, all flows have the same weight, and thus they have the same throughput.

TCP. TCP congestion control does not recognize tenants. Figure 14(a) shows the throughput of 32 TCP flows. Similar to the UDP experiment, without HCSFQ, every flow receives the same amount of bandwidth, and tenant A has higher total throughput. With HCSFQ, the bandwidth is allocated equally to the two tenants, and each flow in A has lower throughput than each flow in B. We also assign weights to the TCP flows as the UDP experiment, and the result is in Figure 14(b). Similarly, with HCSFQ, Flow 1–8 in tenant A have lower throughput than Flow 9–24, because Flow 1–8 lower higher weight. The flows in tenant B have the same throughput because we do not change their weights.


(a) Total request.

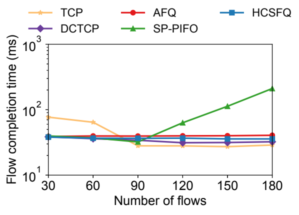

(b) Individual flows.

Figure 17: Simulation result under the incast scenario.

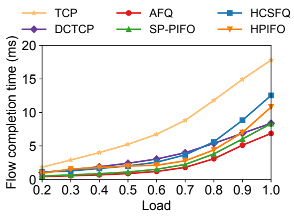
6.3 Large-Scale Simulation

We use simulations to evaluate HCSFQ in a large-scale datacenter environment. The simulations are conducted with a packet-level simulator Netbench [35]. Following the setting in SP-PIFO [12], we use a leaf-spine topology with 144 servers, 9 leaf switches and 4 spine switches, and set the access and leaf-spine links to 1Gbps and 4Gbps, respectively. We compare HCSFQ with TCP, DCTCP, and two state-of-the-art approaches AFQ (32 queues) [11] and SP-PIFO (32 queues) [12]. As in [11, 12], we enable ECN marking and use DCTCP as the transport layer for HCSFQ, AFQ and SP-PIFO.

Web search workload. We generate traffic based on the web search workload [36]. Figure 15(a) shows the flow completion time (FCT) for small flows less than 100KB, and Figure 15(b) shows the flow completion time breakdown when the network is at 70% utilization. HCSFQ achieves up to 60% lower FCT than vanilla TCP and DCTCP. AFQ and SP-PIFO are 15% better than HCSFQ on FCT because HCSFQ enforces fairness by packet dropping and cannot provide guarantee for sensitive packets which can be a drawback for datacenter workload. However, the gap is small and does not grow as the traffic load gets larger. The result demonstrates that HCSFQ is compatible with DCTCP, and can provide significant improvement under a representative datacenter topology and workload as the smaller flows can finish faster with a fair share rate.

Web search workload with injected UDP traffic. To evaluate performance isolation, we inject additional ill-behaved UDP flows to the web server workload. The UDP flows are evenly distributed in the topology and occupy about half of the total bandwidth of the network. Figure 16 shows that TCP and DCTCP perform significantly worse than others, because they do not have performance isolation between TCP and UDP flows. HCSFQ performs better than AFQ and SP-PIFO, because AFQ and SP-PIFO map different flows to a small number of queues and aggressive UDP traffic overloads the queues shared by multiple TCP and UDP flows, while HCSFQ drops excessive UDP packets before they enter the queues.

Incast. This experiment evaluates HCSFQ in an incase scenario where a receiver requests for a 4.5MB file distributed over N (=30–180) sender nodes. We follow the common practice to use a small RTO_{min} (200μs) for all schemes [37]. As


(a) Average flow completion time for flows less than 100KB (Tenant 1).

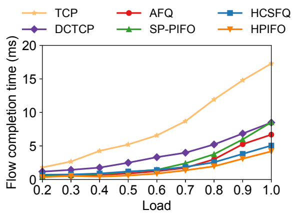

(b) Average flow completion time for flows less than 100KB (Tenant 2).

Figure 18: Simulation result under the web search workload with two tenants. Tenant 1 sends five times as many flows as tenant 2, and should have higher FCT than tenant 2.

shown in Figure 17(a), when the number of flows grows, HCSFQ achieves a lower request completion time compared with SP-PIFO, TCP and DCTCP, and is close to AFQ. SP-PIFO does not handle the incast traffic pattern well, because there are many packets arriving at the same time with similar ranks saturating some queues and getting dropped. Figure 17(b) shows that HCSFQ achieves low average completion times for individual flows as the number of flows changes.

Web search workload with two tenants. This experiment evaluates hierarchical fair queueing with two tenants. Tenant 1 sends five times as many flows as tenant 2, and the flow size and arriving time follow the web search workload [36]. As shown in Figure 18, HCSFQ can provide tenant-level fairness, so that since tenant 1 has more flows, the average flow completion time of tenant 1 is higher than that of tenant 2. We also implement a hierarchical version of PIFO (HPIFO) as an upper bound for comparison. Note that although HPIFO delivers the best result, it needs to maintain three queues (one in the first layer and two in the second layer) for two tenants. It cannot be implemented on today's switches and it is hard to support many tenants due to the need of hierarchical queues. Other approaches do not distinguish between tenants, and the average flow completion times of the two tenants are similar.

Scalability with many tenants. We show the scalability of HCSFQ on supporting many tenants and flows. When there are many tenants and flows, the share of each tenant/flow is small and the bias from rate estimation and rate update in each step will accumulate. In this experiment, we examine 50 tenants. Half of the tenants (tenant 1-25) have one VM in each server, and the other half (tenant 26-50) have two VMs in each server. Each VM has a long-lasting TCP flow with another VM of the same tenant in another rack. We set the bandwidth of access links and leaf-spine links to 10Gbps and 40Gbps respectively in order to accommodate more tenants and flows than previous experiments. Figure 19 shows that TCP, DCTCP, AFQ and SP-PIFO do not provide tenant-level fairness, and the tenants with more flows have higher total throughput. In comparison, HCSFQ provides fair bandwidth allocation between tenants, regardless of the number of flows each tenant has.

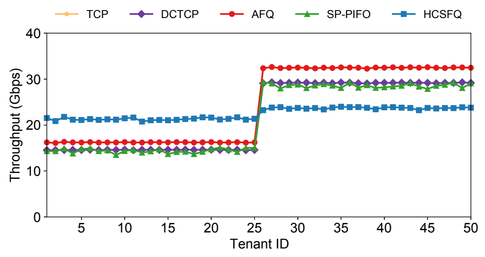


Figure 19: Throughput of different tenants. Each tenant of Tenants 1-25 has one VM in each server, while each tenant of Tenants 26-50 has two VM in each server. Each tenant is sending pairwise TCP traffic between its VMs.

7 Related Work

Fair queueing. There is a long history of work on fair queueing. The original proposal from Nagle [1] introduces the idea of using separate FIFO queues for flows to achieve fair bandwidth allocation. The bit-by-bit round robin (BR) algorithm [2, 3] computes a bid number to estimate the departure time for each packet, and transmits the packet with the lowest bid number with a priority queue. To avoid expensive priority queues, several algorithms, such as SFQ [4] and DRR [5], propose to map flows to a small number of FIFO queues, which do not work well when the number of flows are far larger than the number of queues. Another approach is probabilistic packet dropping, which maintains per-flow state to estimate drop probability, such as FRED [6], RED-PD [7] and AFD [8]. CSFQ [13] is distinct from these algorithms in that it does not require per-flow state, per-flow queues or an expensive priority queue. Hierarchical fair queueing adds a hierarchy to fair queueing, which require not only per-flow state, but also a hierarchy of queues [9, 10, 15]. HCSFQ eliminates both requirements, making hierarchical fair queueing feasible to be implemented in high-speed hardware switches.

Network isolation in multi-tenant cloud. Prior work has proposed techniques to provide performance guarantees and share bandwidth between multiple tenants [14, 16–28, 38, 39]. However, existing works either can only enforce hierarchical fairness at end hosts, or can not be efficiently implemented in today's hardware. For example, BwE [39] is a WAN bandwidth allocation mechanism which enforces hierarchical fair allocation at end hosts. FairCloud [14] proposes to apply CSFQ for network isolation in datacenters, but it does not have a hardware implementation for CSFQ and does not support hierarchical fair queueing. HCSFQ is to the best of our knowledge, the *first* solution to provide hierarchical fair queueing on commodity switches with small switch memory footprint and a single FIFO queue.

Programmable switches. Programmable switches have triggered many innovations in recent years [32, 40–60]. Programmable packet scheduling is the most relevant to HCSFQ. UPS [61] shows that Least Slack Time First (LSTF) provides a good approximation for many scheduling algorithms in practice. PIFO [10] provides a hardware design to realize the abstraction of a push-in first-out (PIFO) queue. It relies on a tree of PIFO queues to implement hierarchical fair queueing. AFQ [11] approximates fair queueing by using a few queues to emulate many queues. It stores per-flow counters in a count-min sketch, and does not support hierarchical fair queueing. SP-PIFO [12] uses several strict priority queues to emulate a PIFO queue, which can support fair queueing, not hierarchical fair queueing. Compared to them, we show how to leverage programmable switches to support fair queueing without per-flow state based on CSFQ, and present a new algorithm HCSFQ to support hierarchical fair queueing.

8 Conclusion

We present HCSFQ, a scalable algorithm for hierarchical fair queueing. Hierarchical fair queueing is a long standing problem in networking. Instead of relying on a hierarchy of queues with complex queue management, HCSFQ only keeps the state for the interior nodes and uses only one queue to achieve hierarchical fair queueing. This dramatically simplifies the design, and makes the design possible to be implemented in high-speed switches. Indeed, we have built a prototype for HCSFQ on programmable switches. Our prototype shows that HCSFQ works well with both UDP and TCP without any changes to either the hardware (e.g., NICs) or software (e.g., TCP/IP stack) of the end hosts.

To the best of our knowledge, HCSFQ is the first solution that has been demonstrated to provide hierarchical fair queueing on hardware switches at line rate. HCSFQ is not only theoretically interesting, but also has important practical implications. Network isolation is critical to multi-tenant clouds, which have a natural two-layer hierarchy. This hierarchy naturally requires the datacenter network to first allocate the bandwidth to the tenants, and then allocate each tenant's bandwidth between the tenant's flows. HCSFQ provides the first solution to enable this two-layer isolation in datacenter networks. Our prototype shows that this can be done without any changes to either the hardware (e.g., NICs) or software (e.g., TCP/IP stack) of the end hosts, and it works well with both UDP and TCP. We believe HCSFQ is a promising solution for network isolation in multi-tenant datacenters.

Acknowledgments

We sincerely thank our shepherd Dongsu Han and the anonymous reviewers for their valuable feedback on earlier versions of this paper. Xin Jin (xinjinpku@pku.edu.cn) is the corresponding author. This work is supported in part by NSF grants CCF-1652257, CNS-1813487 and CCF-1918757, and a Google Faculty Research Award.

References

[1] J. Nagle, “On packet switches with infinite storage,” *IEEE Transactions on Communications*, April 1987.

[2] A. Demers, S. Keshav, and S. Shenker, “Analysis and simulation of a fair queueing algorithm,” *SIGCOMM CCR*, August 1989.

[3] S. Keshav, “On the efficient implementation of fair queueing,” *Internetworking: Research and Experience*, September 1991.

[4] P. E. McKenney, “Stochastic fairness queueing.,” in *IEEE INFOCOM*, June 1990.

[5] M. Shreedhar and G. Varghese, “Efficient fair queueing using deficit round robin,” in *ACM SIGCOMM*, October 1995.

[6] D. Lin and R. Morris, “Dynamics of random early detection,” in *ACM SIGCOMM*, October 1997.

[7] R. Mahajan, S. Floyd, and D. Wetherall, “Controlling high-bandwidth flows at the congested router,” in *IEEE ICNP*, November 2001.

[8] R. Pan, L. Breslau, B. Prabhakar, and S. Shenker, “Approximate fairness through differential dropping,” *SIGCOMM CCR*, April 2003.

[9] J. C. Bennett and H. Zhang, “Hierarchical packet fair queueing algorithms,” in *ACM SIGCOMM*.

[10] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown, “Programmable packet scheduling at line rate,” in *ACM SIGCOMM*, August 2016.

[11] N. K. Sharma, M. Liu, K. Atreya, and A. Krishnamurthy, “Approximating fair queueing on reconfigurable switches,” in *USENIX NSDI*, April 2018.

[12] A. G. Alcoz, A. Dietmüller, and L. Vanbever, “SP-PIFO: Approximating push-in first-out behaviors using strict-priority queues,” in *USENIX NSDI*, February 2020.

[13] I. Stoica, S. Shenker, and H. Zhang, “Core-stateless fair queueing: Achieving approximately fair bandwidth allocations in high speed networks,” in *ACM SIGCOMM*, October 1998.

[14] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica, “Faircloud: Sharing the network in cloud computing,” in *ACM SIGCOMM*, August 2012.

[15] S. Floyd and V. Jacobson, “Link-sharing and resource management models for packet networks,” *IEEE/ACM Transactions on Networking*, August 1995.

[16] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, “Towards predictable datacenter networks,” in *ACM SIGCOMM*, August 2011.

[17] H. Ballani, K. Jang, T. Karagiannis, C. Kim, D. Gunawardena, and G. O’Shea, “Chatty tenants and the cloud network sharing problem,” in *USENIX NSDI*, April 2013.

[18] 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 *ACM CoNEXT*, November 2010.

[19] D. Xie, N. Ding, Y. C. Hu, and R. Kompella, “The only constant is change: Incorporating time-varying network reservations in data centers,” in *ACM SIGCOMM*, August 2012.

[20] S. Angel, H. Ballani, T. Karagiannis, G. O’Shea, and E. Thereska, “End-to-end performance isolation through virtual datacenters,” in *USENIX OSDI*, October 2014.

[21] J. Lee, Y. Turner, M. Lee, L. Popa, S. Banerjee, J.-M. Kang, and P. Sharma, “Application-driven bandwidth guarantees in datacenters,” in *ACM SIGCOMM*, August 2014.

[22] H. Rodrigues, J. R. Santos, Y. Turner, P. Soares, and D. O. Guedes, “Gatekeeper: Supporting bandwidth guarantees for multi-tenant datacenter networks.,” in *Workshop on I/O Virtualization*, June 2011.

[23] K. Jang, J. Sherry, H. Ballani, and T. Moncaster, “Silo: Predictable message latency in the cloud,” in *ACM SIGCOMM*, August 2015.

[24] A. Shieh, S. Kandula, A. G. Greenberg, C. Kim, and B. Saha, “Sharing the data center network,” in *USENIX NSDI*.

[25] V. T. Lam, S. Radhakrishnan, R. Pan, A. Vahdat, and G. Varghese, “NetShare and stochastic NetShare: Predictable bandwidth allocation for data centers,” *SIGCOMM CCR*, June 2012.

[26] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, C. Kim, and A. Greenberg, “EyeQ: Practical network performance isolation at the edge,” in *USENIX NSDI*, April 2013.

[27] L. Popa, P. Yalagandula, S. Banerjee, J. C. Mogul, Y. Turner, and J. R. Santos, “ElasticSwitch: Practical work-conserving bandwidth guarantees for cloud computing,” in *ACM SIGCOMM*, August 2013.

[28] M. Chowdhury, Z. Liu, A. Ghodsi, and I. Stoica, "HUG: Multi-resource fairness for correlated and elastic demands," in *USENIX NSDI*, March 2016.

[29] "Barefoot Tofino." https://www.barefootnetworks.com/technology/#tofino.

[30] "Broadcom Ethernet Switches and Switch Fabric Devices."

[31] "Cavium XPliant," 2018. https://www.cavium.com/.

[32] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching ASICs," in *ACM SIGCOMM*, August 2017.

[33] M. Mathis, J. Semke, J. Mahdavi, and T. Ott, "The macroscopic behavior of the TCP congestion avoidance algorithm," *SIGCOMM CCR*, July 1997.

[34] "Intel data plane development kit (dpdk)," 2018. http://dpdk.org/.

[35] "Netbench." http://github.com/ndal-eth/.

[36] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "pfabric: minimal near-optimal datacenter transport," in *ACM SIGCOMM*, 2013.

[37] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller, "Safe and effective fine-grained tcp retransmissions for datacenter communication," *SIGCOMM CCR*, August 2009.

[38] P. Kumar, N. Dukkipati, N. Lewis, Y. Cui, Y. Wang, C. Li, V. Valancius, J. Adriaens, S. Gribble, N. Foster, and A. Vahdat, "PicNIC: Predictable virtualized NIC," in *ACM SIGCOMM*, August 2019.

[39] A. Kumar, S. Jain, U. Naik, A. Raghuraman, N. Kasinadhuni, E. C. Zermeno, C. S. Gunn, J. Ai, B. Carlin, M. Amarandei-Stavila, *et al.*, "Bwe: Flexible, hierarchical bandwidth allocation for wan distributed computing," in *ACM SIGCOMM*, August 2015.

[40] N. K. Sharma, A. Kaufmann, T. E. Anderson, A. Krishnamurthy, J. Nelson, and S. Peter, "Evaluating the power of flexible packet processing for network resource allocation.," in *USENIX NSDI*, March 2017.

[41] K.-F. Hsu, R. Beckett, A. Chen, J. Rexford, P. Tammana, and D. Walker, "Contra: A programmable system for performance-aware routing," in *USENIX NSDI*, February 2020.

[42] "In-band Network Telemetry (INT) Dataplane Specification." https://github.com/p4lang/p4-applications/blob/master/docs/INT.pdf.

[43] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger, "Sonata: Query-driven streaming network telemetry," in *ACM SIGCOMM*, August 2018.

[44] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim, "Language-directed hardware design for network performance monitoring," in *ACM SIGCOMM*, August 2017.

[45] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, "Hula: Scalable load balancing using programmable data planes," in *ACM SOSR*, March 2016.

[46] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "NetCache: Balancing key-value stores with fast in-network caching," in *ACM SOSP*, October 2017.

[47] Z. Liu, Z. Bai, Z. Liu, X. Li, C. Kim, V. Braverman, X. Jin, and I. Stoica, "DistCache: Provable load balancing for large-scale storage systems with distributed caching," in *USENIX FAST*, February 2019.

[48] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya, "IncBricks: Toward in-network computation with an in-network cache," in *ACM ASPLOS*, April 2017.

[49] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica, "NetChain: Scale-free sub-RTT coordination," in *USENIX NSDI*, April 2018.

[50] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé, "NetPaxos: Consensus at network speed," in *ACM SOSR*, June 2015.

[51] H. T. Dang, M. Canini, F. Pedone, and R. Soulé, "Paxos made switch-y," *SIGCOMM CCR*, April 2016.

[52] D. R. K. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy, "Designing distributed systems using approximate synchrony in data center networks," in *USENIX NSDI*, May 2015.

[53] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. Ports, "Just say NO to Paxos overhead: Replacing consensus with network ordering," in *USENIX OSDI*, November 2016.

[54] H. Zhu, Z. Bai, J. Li, E. Michael, D. Ports, I. Stoica, and X. Jin, "Harmonia: Near-linear scalability for replicated storage with in-network conflict detection," in *Proceedings of the VLDB Endowment*, November 2019.

[55] J. Li, E. Michael, and D. R. K. Ports, "Eris: Coordination-free consistent transactions using in-network concurrency control," in *ACM SOSP*, October 2017.

[56] A. Lerner, R. Hussein, P. Cudre-Mauroux, and U. eXascale Infolab, "The case for network accelerated query processing.," in *CIDR*, January 2019.

[57] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis, "In-network computation is a dumb idea whose time has come," in *ACM SIGCOMM HotNets Workshop*, November 2017.

[58] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. R. Ports, and P. Richtárik, "Scaling distributed machine learning with in-network aggregation," *arXiv preprint arXiv:1903.06701*, February 2019.

[59] N. Ivkin, Z. Yu, V. Braverman, and X. Jin, "Qpipe: Quantiles sketch fully in the data plane," in *ACM CoNEXT*, December 2019.

[60] Z. Yu, Y. Zhang, V. Braverman, M. Chowdhury, and X. Jin, "Netlock: Fast, centralized lock management using programmable switches," in *ACM SIGCOMM*, August 2020.

[61] R. Mittal, R. Agarwal, S. Ratnasamy, and S. Shenker, "Universal packet scheduling," in *USENIX NSDI*, March 2016.

A Proof of Theorem 1

Proof. The first conclusion is directly derived from the guarantee of CSFQ [13].

For the second conclusion, we consider a model with a parent and k children. We add a script $'$ to represent the notations related to the parent, e.g., r'_i is the estimated arrival rate of the i-th packets at the parent. We add a script (j) to represent the notations related to the j-th child, e.g., $r_i^{(j)}$ is the estimated arrival rate of the i-th packets at the j-th child. Suppose the time episode is universal for all children. Suppose that $r_0^{(j)} = r'_0 = 0$ for $j = 1, \dots, k$.

Suppose the inter-arrival time $T_i \geq \tau$ for all i. Suppose

$$r_{\alpha'} \geq \frac{1}{1 - e^{-\tau/K}} \sum_{j=1}^{k} r_\alpha^{(j)}.$$

Then we will show that the parent node $r_{\alpha'}$ does not drop packets. To this end, we only need to prove that

$$r'_i \leq r_{\alpha'}, \quad \forall i. \tag{9}$$

After the first drop, the package length is $h_i = h_i^{(1)} + \cdots + h_i^{(k)}$, where

$$h_i^{(j)} = \begin{cases} \ell_i^{(j)} & r_i^{(j)} \leq r_\alpha^{(j)}, \\ \ell_i^{(j)} \frac{r_\alpha^{(j)}}{r_i^{(j)}} & r_i^{(j)} > r_\alpha^{(j)}. \end{cases}$$

And by definition,

$$r'_i = (1 - e^{-T_i/K}) \frac{h_i}{T_i} + e^{-T_i/K} r'_{i-1}, \quad 1 \leq i \leq n.$$

We now recursively prove Eq. (9).

(i) First let $i = 1$.

We will use the following inequality to prove Eq. (9):

$$(1 - e^{-T_1/K}) \frac{h_1^{(j)}}{T_1} \leq r_\alpha^{(j)}, \quad \forall j. \tag{10}$$

On the one hand, if Eq. (10) is true, we have

$$r'_1 = (1 - e^{-T_1/K}) \frac{\sum_{j=1}^{k} h_1^j}{T_1} \leq \sum_{j=1}^{k} r_\alpha^j \leq r_{\alpha'},$$

which implies Eq. (9) for $i = 1$.

On the other hand, recall $r_1^{(j)} = (1 - e^{-T_1/K}) \frac{\ell_1^{(j)}}{T_1}$, we then prove Eq. (10) as following:

1. If $r_1^{(j)} < r_\alpha^{(j)}$, then $h_1^{(j)} = \ell_1^{(j)}$, thus

$$(1 - e^{-T_1/K}) \frac{h_1^{(j)}}{T} = (1 - e^{-T_1/K}) \frac{\ell_1^{(j)}}{T} = r_1^{(j)} \leq r_\alpha^{(j)}.$$

2. If $r_1^{(j)} \geq r_\alpha$, then $h_1^{(j)} = \ell_1^{(j)} \frac{r_\alpha^{(j)}}{r_1^{(j)}}$, thus

$$(1 - e^{-T_1/K}) \frac{h_1^{(j)}}{T_1} = (1 - e^{-T_1/K}) \frac{\ell_1^{(j)}}{T_1} \frac{r_\alpha^{(j)}}{r_1^{(j)}} = r_\alpha^{(j)}.$$

Thus Eq. (10) holds.

(ii) Now suppose that $r'_{i-1} \leq r_{\alpha'}$.

We will use the following inequality to prove our claim:

$$(1 - e^{-T_i/K}) \frac{h_i^{(j)}}{T_i} \leq r_\alpha^{(j)}, \quad \forall j. \tag{11}$$

On the one hand, if Eq. (11) is true, we have

$$\begin{aligned} r'_i =& (1 - e^{-T_i/K}) \frac{\sum_{j=1}^{k} h_i^{(j)}}{T_i} + e^{-T_i/K} r'_{i-1} \\ \leq& \sum_{i=1}^{k} r_\alpha + e^{-a/K} r'_\alpha \leq r'_\alpha, \end{aligned}$$

which implies Eq. (9) for i.

On the other hand, recall

$$r_i^{(j)} = (1 - e^{-T_i/K}) \frac{\ell_i^{(j)}}{T_i} + e^{-T_i/K} r_{i-1}^{(j)},$$

we then prove Eq. (11) as following:

1. If $r_i^{(j)} < r_\alpha^{(j)}$, then $h_i^{(j)} = \ell_i^{(j)}$, thus

$$\begin{aligned} (1 - e^{-T_i/K}) \frac{h_i^{(j)}}{T_i} =& (1 - e^{-T_i/K}) \frac{\ell_i^{(j)}}{T_i} \\ =& r_i^{(j)} - e^{-T_i/K} r_{i-1}^{(j)} \\ \leq& r_i^{(j)} \leq r_\alpha^{(j)}. \end{aligned}$$

2. If $r_i^{(j)} \geq r_\alpha^{(j)}$, then $h_i^{(j)} = \ell_i^{(j)} \frac{r_\alpha^{(j)}}{r_i}^{(j)}$, thus

$$\begin{aligned} (1 - e^{-T_i/K}) \frac{h_i^{(j)}}{T_i} =& (1 - e^{-T_i/K}) \frac{\ell_i^{(j)}}{T_i} \frac{r_\alpha^{(j)}}{r_i^{(j)}} \\ =& (r_i^{(j)} - e^{-T_i/K} r_{i-1}^{(j)}) \frac{r_\alpha^{(j)}}{r_i^{(j)}} \\ \leq& r_i^{(j)} \frac{r_\alpha^{(j)}}{r_i^{(j)}} = r_\alpha^{(j)}. \end{aligned}$$

Thus Eq. (10) holds. By (i) and (ii) and mathematical induction our proof is finished. $\square$

Breaking the Transience-Equilibrium Nexus: A New Approach to Datacenter Packet Transport

Shiyu Liu[1], Ahmad Ghalayini[1], Mohammad Alizadeh[2], Balaji Prabhakar[1], Mendel Rosenblum[1], and Anirudh Sivaraman[3]

[1]Stanford University, [2]MIT, [3]NYU

Abstract

Recent datacenter transport protocols rely heavily on rich congestion signals from the network, impeding their deployment in environments such as the public cloud. In this paper, we explain this trend by showing that, without rich congestion signals, there is a strong tradeoff between a packet transport's equilibrium and transience performance. We then propose a simple approach to resolve this tension without complicating the transport protocol and without rich congestion signals from the network. Our approach factors the transport into two separate components for equilibrium and transient handling. For equilibrium handling, we continue to use existing congestion control protocols. For transients, we develop a new underlay algorithm, On-Ramp, which intercepts and holds any protocol's packets at the network edge during transient overload. On-Ramp detects transient overloads using accurate measurements of one-way delay, made possible in software by a recently developed time-synchronization algorithm.

On the Google Cloud Platform, On-Ramp improves the 99th percentile request completion time (RCT) of incast traffic of CUBIC by 2.8× and BBR by 5.6×. In a bare-metal cloud (CloudLab), On-Ramp improves the RCT of CUBIC by 4.1×. In ns-3 simulations, which model more efficient NIC-based implementations of On-Ramp, On-Ramp improves RCTs of DCQCN, TIMELY, DCTCP and HPCC to varying degrees depending on the workload. In all three environments, On-Ramp also improves the flow completion time of non-incast background traffic. In an evaluation at Facebook, On-Ramp significantly reduces the latency of computing traffic while ensuring the throughput of storage traffic is not affected.

1 Introduction

Datacenter packet transport has been an active area of research within the networking community for over a decade. The primary goals of datacenter transport protocols are to achieve high throughput and low latency and to effectively deal with bursty traffic, especially incast [51]. To achieve these goals, the research community has pursued two broad lines of work. The first is a series of congestion control algorithms that have relied on progressively richer forms of congestion signals from the network. These signals run the gamut from single-bit ECN marking [13] to multi-bit signals [53] and all the way to queue size and link utilization information [44]. The second line of work has focused on packet scheduling mechanisms that proactively prevent congestion from occurring in the first place [21, 34], or explicitly optimize for objectives like flow completion times [16, 17, 20, 49]. These schemes often require more elaborate network and application support, such as in-switch priority queues and application hints about flow sizes or deadlines.

Taking a step back from recent developments, we ask: *is it possible for a congestion control algorithm to achieve good behavior without rich congestion signaling or packet scheduling support from the network?* This question is not merely of academic interest; it has significant practical implications. In many environments, there is no way for the network to export rich signaling information or perform sophisticated packet scheduling. A particularly important example is the environment in which public cloud customers find themselves today. Cloud customers have access to the edge of the network, whether it is within a VM or in a bare metal server or potentially within SmartNICs in the future. However, they do not have access to the network infrastructure.

Motivated by the above question, we show that the trend towards rich congestion signals in state-of-the-art schemes is rooted in an inherent tension between the two main functions of a datacenter transport protocol: (i) converging quickly to a fair and stable equilibrium point as large flows arrive and depart, and (ii) handling transients of the incast-type effectively. Specifically, we consider two widely-deployed algorithms (TIMELY [47] and DCQCN [53]) and show that a parameter setting that works well in equilibrium performs poorly in transience and vice versa. On the other hand, protocols such as HPCC [44] which use richer congestion signaling from the network (e.g., queue size information and link utilization from INT [41]) can improve performance in both equilibrium and transience, as discussed in [44, §2.3 and §5.2].

Informed by these results, we ask a second question: *can congestion control be modularized into two simpler components, one each to deal with equilibrium and transient concerns?* This paper answers the question by proposing a new approach to congestion control that breaks the nexus between transience and equilibrium behavior. It delegates transient handling to a protocol, called *On-Ramp*, which is tailor-made for transients, leaving the congestion control algorithm to deal with equilibrium behavior.

On-Ramp (OR) can be coupled with any datacenter congestion control protocol to improve its performance during transients; indeed, implementing On-Ramp requires only end-host modifications and we have combined On-Ramp with CUBIC [33], BBR [19], DCQCN [53], TIMELY [47], DCTCP [13] and HPCC [44]. The core idea behind On-Ramp is extremely simple: when the congestion control protocol at a sender decides to transmit a packet, the sender temporarily holds back the packet if the sender-to-receiver one-way delay of the most-recently acknowledged packet exceeds a threshold. Thus, On-Ramp lets congestion control algorithms do what they are good at—determining transmission rates or window sizes in dynamic settings to improve metrics such as throughput and fairness—while giving them a leg up with functionality they struggle with: transience.

As a very useful by-product, On-Ramp also enhances the performance of existing congestion control algorithms in equilibrium by making them more robust to the choice of algorithm parameters. This is because of a phenomenon that we term *state-space compaction*. The state space of a network is the size of the queues within the network at any time. It spans a large dynamic range of queues sizes from transient (high queue) to equilibrium (low, but non-zero) to unstable (zero queues and link underflows). By compacting the state space, On-Ramp reduces the dynamic range of the state space observed by the congestion control algorithm and keeps it in a tight band around the desired operating point. This attenuates the large and frequent congestion signals from the network which, in turn, can cause an overreaction by the congestion control algorithm.

Key to making On-Ramp practical is an accurate measurement of one-way delay, which requires synchronized clocks between the sender and receiver. Since the one-way delays in a datacenter can be a few tens of microseconds or lower, this implies that the sender and receiver clocks must be synchronized to within a few microseconds. Such high-accuracy clock synchronization has traditionally required hardware-intensive protocols like Precision Time Protocol (PTP) [37]. But a recently developed system, Huygens [30], showed that it is possible to achieve nanosecond-level clock synchronization without special hardware or dedicated priorities. On-Ramp leverages the Huygens algorithm, making it easier to deploy.

We evaluated On-Ramp in three different environments: the public cloud, a CloudLab cluster (bare-metal cloud) and ns-3 simulations. We find:

1. **Performance improvements.** On Google Cloud, On-Ramp improves the 99th percentile request completion time (RCT) of incast traffic of CUBIC by $2.8\times$ and BBR by $5.6\times$. In CloudLab, On-Ramp improves the tail RCT of CUBIC by $4.1\times$. In ns-3 simulations, which model more efficient NIC-based implementations, On-Ramp improves RCTs to varying degrees depending on the workload under DCQCN, TIMELY, DCTCP and HPCC. In all three environments, On-Ramp also improves the flow completion time of non-incast background traffic.
2. **CPU overhead of On-Ramp.** When running at 40% network load on a 10 Gbps NIC and an 8-core CPU, the total CPU utilization is 15.1% without On-Ramp, and 18.7% with On-Ramp. If On-Ramp is implemented in the NIC, this overhead can be eliminated.

2 Transience–Equilibrium Tension

Congestion control algorithms execute one of the following update equations:

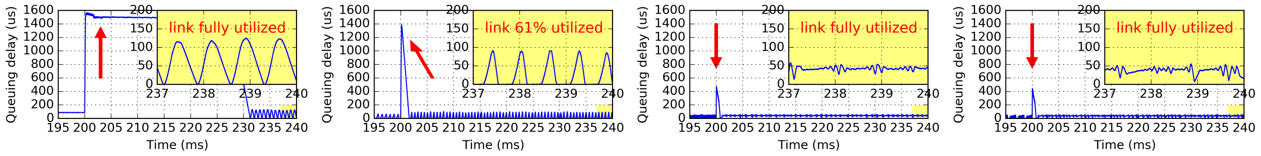
$$\begin{aligned} W(next) &= f(W(now),\ congestion\ signals,\ K), \text{ or} \\ R(next) &= g(R(now),\ congestion\ signals,\ K). \end{aligned} \tag{1}$$

That is, based on congestion signals received from the network, the algorithm updates the window size W or the transmission rate R. This is taking place constantly and iteratively, as flows arrive or depart or the path bandwidth changes (e.g., due to prioritization). K here is the gain in the control loop, which is typically very carefully chosen to provide stability in equilibrium and quick reaction to congestion in transience.[1] However, in the high bandwidth and small-buffered environment of data centers, we shall see that it is quite hard to pick the gain K so as to get great performance in *both* transience and equilibrium: a high value of K gives the responsiveness needed to react quickly to congestion but suffers from bad performance and instability in equilibrium when there are lags in the control loop; conversely, a low value of K can provide good performance in equilibrium but make the source sluggish during transience. We refer to this as the *transience-equilibrium tension* and we shall see that industrial-grade and commercially-deployed algorithms like TIMELY and DCQCN suffer from the transience-equilibrium tension.

With elaborate and frequent congestion signals from the network (e.g., queue depths, link utilization and flow rate, sent on a per-packet basis using in-band telemetry [41]), the congestion control algorithm can be improved simultaneously in transience and equilibrium (e.g., HPCC [44]). Unfortunately, such elaborate signals are not available in virtual environments such as the public clouds.

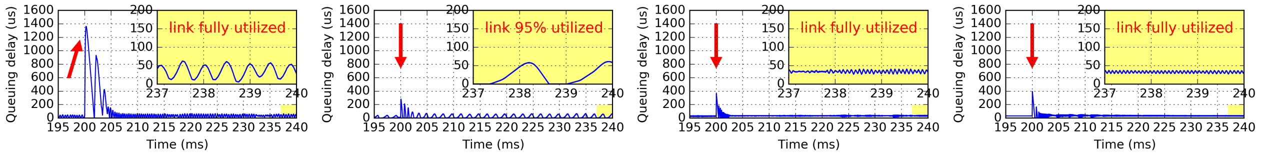
An alternative is to bake in two different modes of congestion handling within the same algorithm, e.g., slow start for transients and congestion avoidance for equilibrium. However,

[1] Indeed, an extensive literature in congestion control theory is devoted to the careful choice of the gain parameters, e.g. [14, 15, 24, 27, 40, 45, 52].



(a) β=0.2, no OR **(b)** β=0.8, no OR **(c)** β=0.2, with OR **(d)** β=0.8, with OR

Figure 1: 12 100G servers sending traffic to one receiver using TIMELY. 2 of them start at t=0, the other 10 start at t=200ms. From ns-3 runs using the setup in §5.1.1. The red arrow points to transience, and the yellow box is a zoom-in of equilibrium. OR threshold T=30μs.



(a) T_i=55μs, T_d=50μs, no OR **(b)** T_i=300μs, T_d=4μs, no OR **(c)** T_i=55μs, T_d=50μs, with OR **(d)** T_i=300μs, T_d=4μs, with OR

Figure 2: DCQCN study. OR threshold T=30μs.

this approach is fragile because it requires defining precise conditions to trigger switches between modes. Further, when a congestion control algorithm in equilibrium encounters severe congestion and drastically cuts its window or rate, it must make a difficult choice between remembering and forgetting its previous equilibrium state (rate/window). This choice is difficult because it depends on the duration of congestion, which is hard to predict. If the congestion is transient, such as an incast, the algorithm must remember its previous state so as to rapidly restore the old equilibrium without under utilization, once the incast ends. On the other hand, if the congestion is sustained, such as the simultaneous arrival of many long-lived flows, the algorithm must forget its previous state so that it can rapidly find a new equilibrium.

In this context, On-Ramp makes two key contributions: (i) when the one-way delay (OWD) on the path exceeds a given threshold, T, On-Ramp can quickly and forcefully react by pausing transmissions at the source, reducing congestion; and (ii) On-Ramp also reduces the sensitivity to the choice of the gain parameter K by ensuring good transient and equilibrium performance over a wide range of values of K.

We shall show that On-Ramp achieves the above by *compacting the state space* in the network. By that we mean that On-Ramp maintains network state variables such as queuing delays around their desired operating points, preventing large excursions (high or low) which occur during transience or in equilibrium when the value of K is high. Essentially, state space compacting *attenuates* (but does not eliminate) the congestion signals, preventing an overreaction at a source with high gain K while providing the throttling necessary for a source with low gain K.

Let us now illustrate the above points by considering the following scenario: 12 100G servers sending traffic to one receiver; all machines are connected to a single switch. 2 servers start sending at time 0; the other 10 start at 200 ms.

Figures 1a and 1b show the queuing delays at the switch during transience (t=200ms) and equilibrium (t=237-240ms) when TIMELY is used with its gain parameter $\beta = 0.2$ and $\beta = 0.8$, respectively. Note that $\beta = 0.8$ is the value recommended in [47,54], and $\beta = 0.2$ is the largest value to achieve full utilization in equilibrium.

At $\beta = 0.2$, TIMELY performs well in equilibrium: the queue stays above zero and the link rate is maintained close to 100G. However, it performs poorly in transience: it takes a long time to react to the congestion caused by the newly arriving 10 flows—the queuing delay converges very slowly. At $\beta = 0.8$, it reacts much more quickly to transience, but due to aggressive congestion control, the queue underflows during equilibrium and leads to a link utilization of just 61%.

The trend is similar for DCQCN. The rate-increase timer T_i and rate-decrease timer T_d are varied to change its control gains. Note that the values T_i = 55 μs and T_d = 50 μs are the settings recommended in [53] and T_i = 300 μs and T_d = 4 μs are the default settings recommended by a vendor of network hardware (cf. [44]). At less aggressive gains (Fig. 2a), DCQCN performs well in equilibrium but reacts slowly during transience. Aggressive gain settings (Fig. 2b) give the opposite behavior.

Figures 1c and 2c respectively consider the scenario where the low (equilibrium-friendly) gain parameters for TIMELY and DCQCN are used in conjunction with On-Ramp. We observe that both algorithms react very quickly to transient congestion and converge smoothly to a stable equilibrium with full link utilization. Indeed, the equilibrium performance is actually improved by On-Ramp: the oscillation of queues during equilibrium is reduced! Figures 1d and 2d consider the high gain parameter scenario. We see that On-Ramp helps here as well by preventing severe queue undershoots and providing a high link utilization.

In conclusion, On-Ramp helps to cope with severe transient congestion by lowering queuing delays rapidly; conversely, it also prevents queues from underflowing, hence keeping a high link utilization. It achieves this by compacting the state space to a region around the desired queue size. As a useful by-product of this, it also reduces the sensitivity of the congestion control algorithm to gain parameters. It is critical to note here that On-Ramp *does not* perform the window or rate updates at Equation 1. This is left to the congestion control algorithm.

One might wonder if a more sophisticated delay-based protocol could make better use of precise one-way delay measurements. While an interesting possibility, we did not pursue this route for two reasons. First, On-Ramp allows us to decouple transient and long-term congestion management, and handle each at its own appropriate timescale without burdening a single protocol with both. Second, On-Ramp also composes very naturally with existing congestion control algorithms. It lets congestion control algorithms do what they are good at—improving long-term metrics such as throughput and latency, while taking care of transients for them. This factorized approach has allowed us to combine On-Ramp with several existing congestion control algorithms [13, 19, 33, 44, 47, 53].

3 On-Ramp Design

On-Ramp is a simple end-to-end flow control algorithm, sitting as a shim between the congestion control algorithm and the network (see Figure 3). On-Ramp aims to bring down the path queuing delays as quickly as possible by pausing the flow at the sender's end of the network when the measured OWD (which we denote as O) exceeds a threshold T. For a congestion control algorithm that does not control queuing delays on its own (e.g., TCP CUBIC), On-Ramp adds this functionality. For a congestion control algorithm that does control queuing delays on its own (e.g., DCTCP), On-Ramp works like a safeguard, for example, by reducing queue spikes during transience.

We first present a simple version of the On-Ramp algorithm that is intuitive but has queue oscillations and the possibility of under-utilization in the presence of feedback delay. We update On-Ramp by amending the rule for pausing, resulting in the final version of the algorithm.

3.1 Strawman Proposal for On-Ramp

As shown in Figure 3, On-Ramp is implemented underneath the congestion control protocol (CC) between the sender S and receiver R. It consists of two parts:

(i) Receiver side: Upon receiving a packet, the receiver sends an OR-ACK to the sender, which contains (1) the flow indicator (i.e., the 5-tuple representing the flow), (2) the sequence number of the received packet, and (3) the time at which the packet was received.

(ii) Sender side: The sender maintains two data structures for each flow: (1) a queue of outstanding packets belonging to that flow waiting to be transmitted; and (2) a value called $t_{NextPkt}$ representing when the next packet from the flow will be transmitted. Initially, $t_{NextPkt}$ is set to 0 and, upon the receipt of an OR-ACK, it is updated as follows:

$$t_{NextPkt} \leftarrow \begin{cases} t_{Now} + O - T, & \text{if } O > T \\ t_{NextPkt}, & \text{else.} \end{cases} \quad (2)$$

Here t_{Now} is the current system time. The flow will be paused until $t_{NextPkt}$ if that is larger than t_{Now}. If an OR-ACK is dropped, the sender will not be able to measure OWD O at

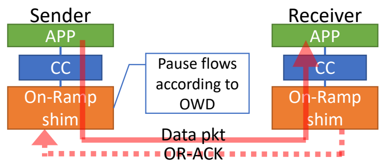


Figure 3: The On-Ramp Underlay.

this time, so $t_{NextPkt}$ will not be changed. The sender dequeues packets of the active (non-paused) flows in round-robin order.

The strawman proposal is simple and intuitive: upon receiving an OR-ACK with an OWD value of O exceeding threshold T, pause for $O - T$. The goal is to drain the queue such that the OWD after the pause is under T. This reasoning would have been correct *if there were no delay* in getting acks from the receiver. In the presence of feedback delay, however, the sender will actually pause for a significantly longer time than necessary.

To understand what is going on, suppose that the sender receives an ack with OWD O exceeding T for the first time at time t, and it immediately pauses for duration $O - T$. Notice that it will take at least one additional round-trip-time (RTT) after t for the sender to see the impact of this pause on the OWD values carried in acks. In particular, acks received for packets that were transmitted before pausing are likely to also carry OWD values exceeding T. Hence a sender using the strawman design will actually pause for at least the next RTT, even if the OWD exceeds T by a small amount. By the time it resumes sending traffic, the queue will have significantly undershot T, which risks under-utilization.

3.2 The Final Version of On-Ramp

To fix the above problem, we propose a simple mechanism to compensate for the sender's feedback delay in receiving the OWD signal. The key is to observe that it is possible the sender was paused when the green packet (see Figure 4) was in flight and before the sender received its ack. The update equation (2) doesn't take these previous pauses into account, and therefore it overestimates how much additional time it needs to pause for the OWD to drop down to T.

One approach to correct for previous pauses would be to subtract the total time the sender was paused while the green packet was in flight from the OWD value O obtained in the ack. This approach assumes that if the sender was paused for some duration P, then the current value of OWD is no longer O but rather $O - P$. However, this ignores the contribution of other senders to the OWD. The reduction in OWD due to a pause of duration P is *at most* P, but it may be less if other senders transmit in that time.

To account for other senders, On-Ramp estimates the relationship between the actions it takes (i.e., its own pause duration) and the effect of these actions (i.e., reduction in OWD). To this end, it dynamically measures a parameter β_m, which equals the change in OWD per unit of On-Ramp pause time; in other words, the *gain* of the On-Ramp control mechanism. Refer to Figure 4, and let O_B and O_G be the OWDs

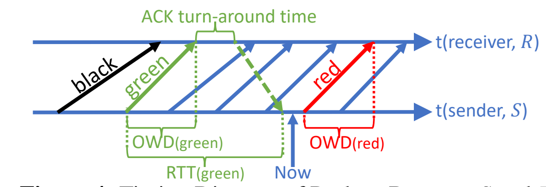
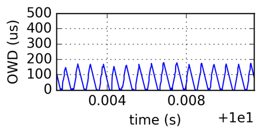
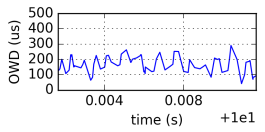


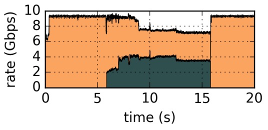
Figure 4: Timing Diagram of Packets Between S and R



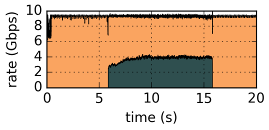
(a) OWD with Strawman OR



(b) OWD with Final OR



(c) Flow rates, Strawman OR



(d) Flow rates, Final OR

Figure 5: Two long-lived CUBIC flows sharing a link

of the black and green packets, where the black packet is the one which has been acked immediately prior to the green packet being acked. Let P_{BG} be the total time On-Ramp had paused the sender in between the times it transmitted these two packets. If $P_{BG} > 0$, then $\beta_m = (O_B - O_G)/P_{BG}$ captures the effect of the sender pausing on the reduction in the OWD. We threshold β_m to be between 0 and 1. A value of β_m close to 1 indicates that the reduction in OWD is roughly the same as the sender's pause time. This would occur, for example, if there is little traffic from other senders, or if the senders are synchronized and all pause together. On the other hand, β_m near 0 indicates that On-Ramp must pause for a long amount of time to reduce OWD.

On-Ramp obtains one such β_m measurement for each new packet that it transmits for which $P_{BG} > 0$. It computes a moving average of these values with each new measurement:

$$\beta \leftarrow (1-g)\cdot\beta + g\cdot\beta_m \qquad (\text{if } P_{BG} > 0) \tag{3}$$

Here g is the EWMA gain. Finally, upon receiving an OR-ACK, the sender replaces the first line in Equation (2) with

$$\begin{aligned} t_{NextPkt} &\leftarrow t_{Now} + O - T - \beta\cdot P_{LastPktRTT} \\ &\quad (\text{if } O - \beta\cdot P_{LastPktRTT} > T) \end{aligned} \tag{4}$$

Here, $P_{LastPktRTT}$ is the total time pause was asserted during the time period spanned by the RTT of the most recently acked packet (the green packet).

Figure 5 demonstrates the effectiveness of lag compensation. It shows the OWDs and the flow rates when two CUBIC flows share a bottleneck link in a bare-metal environment (Cloudlab [26]). The second flow is on during 6–16 seconds. The threshold T is set to 50μs. It's clear that the strawman On-Ramp leads to significant queue undershoot under T and causes under-utilization, and the final On-Ramp fixes this problem. Figure 24 in Appendix shows the case of 12 flows, where final On-Ramp removes most fluctuations in queue lengths and achieves better fair sharing among all flows.

Parameter Selection. The threshold T should clearly be higher than the minimal OWD on the path, plus some headroom to tolerate errors in the OWD measurement. §5.1.4 describes how to pick T in practice. We choose the EWMA gain $g = 1/16$ and find that the end-to-end performance of On-Ramp is relatively insensitive to g, shown in §7.4.

3.3 Importance of Accurate One-way Delay

To measure OWD accurately, On-Ramp uses Huygens [30], a recently developed system for highly-accurate clock synchronization. The Huygens algorithm uses a random probe mesh among all the clocks; the mesh is formed by each clock probing typically 10 other clocks. The clocks exchange probes and acks on this mesh. We note that probes and acks are UDP packets *not* using higher priorities in switches. The send and receive timestamps of each probe and ack are processed through a combination of local and central algorithms every 2–4 secs. The local algorithm performs filtering using coded probes and support vector machines to estimate the discrepancies between two clocks. These techniques make Huygens robust to network queuing delays, random jitter, and timestamp noise. Then, the central algorithm, dubbed "network effect" in the paper, determines errors in the accuracy of clock sync using the transitivity property: the sum of the clock offsets A-B, B-C, and C-A is zero; else, errors exist. By looking at clock offset surpluses over *loops* of the probe mesh, Huygens pins down clock sync errors and provides corrections which can be applied offline or online.

Since the probe mesh is set up end-to-end at the hosts, there is no need for special hardware support (in contrast to other high-accuracy algorithms like PTP [37], DTP [43], or DPTP [39]). With NIC hardware timestamps, [30] reports a 99th percentile synchronization accuracy of 20–40 nanoseconds even under network loads of 80%. The probe mesh makes Huygens robust to high loads and link or node failures. Further, the local-central processing distributes effort across all nodes, making the algorithm scalable to 1000s of nodes. In the present paper, we use Huygens with CPU (software) timestamps because software timestamps are universally available in both VMs in public clouds and bare-metal machines. In this case, we see a median accuracy of a few hundred nanoseconds and 99th percentile accuracy of 2–3 microseconds under high network loads such as 80% in a single data center.

To see the importance of accurate one-way delay, we compare three signals that On-Ramp could use to measure path congestion: (i) OWD measured with accurately synchronized clocks using Huygens, (ii) round-trip time (RTT), and (iii) OWD measured with less accurately synchronized clocks using NTP [46]. Referring to Figure 4, we evaluate how well each of these signals, measured for the green packet, correlates with the OWD of the red packet. If the correlation is high, then the fate of the green packet is a good predictor of the congestion to be experienced by the red and immediately succeeding packets. As Figure 6a and 6d show, the OWD of the green packet measured using Huygens is highly corre-

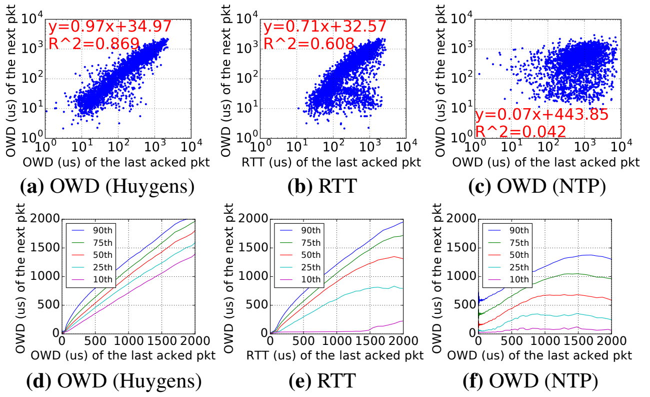


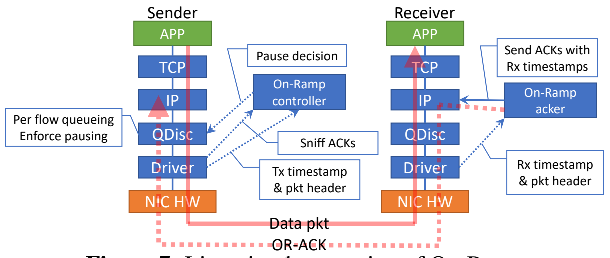
Figure 6: The OWD of the next packet vs. the OWD (Huygens), RTT, and OWD (NTP) of the *last-acked packet*. (a)–(c) are samples plotted in log scale. (d)–(f) are percentiles plotted in linear scale. The test is conducted on 100 machines in a bare metal cloud with *WebSearch* [13] traffic and 40% network load.

lated with the OWD of the red packet. However, neither the RTT nor the OWD measured using NTP correlates well with the actual OWD experienced by the red packet. This makes On-Ramp much less effective with the latter two signals.

4 Implementation

In this section, we describe the Linux implementation of On-Ramp using kernel modules, which allows us to install On-Ramp by just loading a few kernel modules rather than reinstalling (or worse still, recompiling and then reinstalling) the whole kernel. We will also comment on the benefits—both implementation and performance—of implementing On-Ramp in Smart NICs in the future.

4.1 Linux Kernel Modules



Sender
Receiver
APP
TCP
IP
QDisc
Driver
NIC HW
Pause decision
On-Ramp controller
Sniff ACKs
Tx timestamp & pkt header
Per flow queueing Enforce pausing
Send ACKs with Rx timestamps
On-Ramp acker
Rx timestamp & pkt header
Data pkt
OR-ACK

Figure 7: Linux implementation of On-Ramp

Figure 7 shows our On-Ramp implementation. It consists of four parts, described below. For each of them, we mention whether it involves changes to the sender, receiver, or both.

1. **On-Ramp controller.** This module runs at the sender and calculates OWDs and makes decisions to pause or not on a per-flow basis.
2. **NIC driver.** The NIC driver at the receiver is modified to timestamp packets (before GRO). The NIC driver at the sender is modified to timestamp packets, sniff OR-ACKs and forward them to the On-Ramp controller for implementing the On-Ramp algorithm. The NIC driver could be either a physical or a virtual NIC's driver.
3. **QDisc.** We modify the fair queueing (FQ) Qdisc [25] at the sender to queue packets into per-flow queues and exert pause on a per-flow basis.
4. **On-Ramp acker.** This module at the receiver sends OR-ACKs. These are UDP packets that use the same Ethernet priority as the received data packets, so they don't require any priority queues in Ethernet switches. We avoid piggybacking receive timestamps onto TCP ACKs, because they may be delayed by the TCP stack, and modifying the TCP stack requires recompiling the kernel.

Note that On-Ramp does not modify the existing data packets, and OR-ACKs are standard UDP packets.

There are three important details regarding the implementation that pertain to three critical aspects of On-Ramp: (i) the accuracy of measuring the OWD, (ii) the granularity of control (exerting pause), and (iii) the behavior after a pause ends. Accordingly, the On-Ramp implementation may vary depending on the deployment scenario, e.g., public cloud (Google Cloud Platform), bare-metal cloud (CloudLab), or bare-metal cloud with SmartNICs. We expand on these details below.

1. **Timestamp collection.** To compute OWD, we need to collect both sender and receiver timestamps. These timestamps are taken within the NIC driver and based on the system clock on both the sender and receiver sides. We choose the NIC driver because it is close to the wire and therefore minimizes additional software stack latency being added to the OWDs. This is important because stack latencies can be quite variable and confound the accurate detection of one-way delays in the *network* which are caused by congestion. Even though this makes On-Ramp's implementation NIC-driver-specific, the patch is only 20 lines of code and is quite easy to add to the NIC driver.[2] In bare-metal machines, if the NICs support hardware timestamping (e.g., through PHC [1]), NIC timestamps can also be captured, yielding less noisy OWD measurements, which, in turn, can lead to better control.

2. **The effect of generic send offload (GSO).** On the sender side, when GSO is enabled, the data segments handled by QDisc and the driver are GSO segments, which can be up to 64 KB (~43 packets). This limits the granularity of control by On-Ramp, as well as the accuracy of capturing transmit timestamps. §5 shows that On-Ramp already gives satisfactory performances with the default setting of GSO enabled and the max GSO size of 64KB. §7.2 shows that reducing max GSO size will further improve On-Ramp's performance, but increase CPU overhead, so there is a tradeoff here.

3. **Behavior after a pause ends.** One might wonder whether a burst of packets will be sent into the network after a pause ends, causing spikes in the queuing delay. This does not happen in practice. (1) For window-based CCs like CUBIC and DCTCP, although packets are also queued in the On-Ramp module, thanks to the TCP small queues patch [2] in Linux, the total number of bytes queued in the TCP stack and On-

[2]In a given public cloud, the vNIC implementations are the same, making the addition of the patch a one-time effort for each public cloud.

Ramp modules are limited. Furthermore, because On-Ramp is based on the FQ Qdisc, it dequeues in round-robin order across flows, smoothing out traffic after a pause ends. (2) For rate-based CCs like BBR, DCQCN and TIMELY, On-Ramp exerts pause by letting the rate pacer[3] hold back the next packet until $t_{Now} \geq \max(t_{NextTx}, t_{NextPkt})$, where t_{NextTx} is the time of sending the next packet determined by CC algorithm; $t_{NextPkt}$ is the value maintained by On-Ramp for each flow (see §3.1). When a pause ends, the rate pacer will resume releasing packets into the network according to the rate determined by CC, therefore the transmission will not be burstier.

Network and CPU overhead. The network overhead of On-Ramp comes from OR-ACKs; typically, 1 OR-ACK is sent per about 10 MTU-sized packets. Therefore, at 40% load on a 10 Gbps network and 78 Bytes per OR-ACK, the bandwidth consumed by OR-ACKs is about 21 Mbps, or 0.2% of the line rate. Huygens provides the clock synchronization service for On-Ramp. Its probe mesh adds negligible network overhead, which is about 3 Mbps, or 0.03% of the line rate.

For a typical scenario in §5, running 40% load of *Web-Search* traffic plus 2% load of incast on a 10 Gbps NIC and an 8-core Intel Xeon D-1548 CPU, the total CPU utilization is 15.1% without On-Ramp, and 18.7% with On-Ramp. If On-Ramp is implemented in the NIC as described later, this overhead can be eliminated. The CPU usage of Huygens is only around 0.5%.

We consider On-Ramp's overhead under higher network speeds. Table 1 shows overheads when two servers send iPerf flows to a third server simultaneously. Each server has a 25 Gbps NIC and a 10-core Intel Xeon E5-2640v4 CPU. We use the default GSO settings: enabled, max GSO size = 64KB. In this experiment, we get the same throughput with and without On-Ramp, both saturating the 25G link at the receiver. Most of the CPU overhead of On-Ramp is at the receiver, caused mainly by parsing, timestamping, and sending OR-ACKs. At the senders, On-Ramp operates at the granularity of GSO segments, so the overhead is small. Note that the On-Ramp implementation has not yet been optimized for CPU overhead.

	No On-Ramp[4]	With On-Ramp
Sender	1.01%	1.04%
Receiver	5.50%	6.99%

Table 1: The CPU usage in an iPerf experiment with 25G NIC

4.2 NIC Implementation

With the advent of programmable SmartNICs [3,7], On-Ramp can ideally be offloaded to the NIC in the future, conserving host CPU cycles. Moreover, a SmartNIC implementation can further improve On-Ramp's performance due to (1) shorter ack turn-around times at the receiver and (2) the exertion of pauses on MTU-sized packets, rather than GSO segments. §5.2.3 uses ns-3 [8] to emulate On-Ramp's performance using a NIC implementation.

[3] BBR uses FQ Qdisc to pace packets, which is compatible with the Linux kernel module implementation of On-Ramp. DCQCN's rate pacing is inside NICs, TIMELY's rate pacing can be in software or NICs, and we use ns-3 simulation to study On-Ramp's performance on top of them, as in §4.2.

[4] Since On-Ramp is built based on FQ QDisc, for a fair comparison of CPU overhead, we use FQ without On-Ramp as a baseline in this experiment. We note that FQ_CoDel [9] (the default QDisc in many modern Linux distributions such as Ubuntu 18.04) incurs a higher CPU overhead than FQ QDisc (when On-Ramp is not used): sender 1.41%, receiver 5.68%.

5 Evaluation

We evaluate the effectiveness of On-Ramp in a variety of environments and under different workloads and congestion control algorithms. In terms of performance, we consider:

1. Application-level performance measures: (i) request completion times for incast traffic, and (ii) flow completion times for non-incast background traffic.
2. Network-level performance measures: (i) number of packet drops and (ii) number of TCP timeouts.

5.1 Evaluation Setup

5.1.1 Evaluation environments

We consider three environments: public clouds, bare-metal clouds, and ns-3 [8] simulations. In the clouds we use On-Ramp's Linux implementation with different CC algorithms. The ns-3 simulations help us understand On-Ramp's performance in RDMA networks with different CC algorithms such as DCQCN, TIMELY, DCTCP, and HPCC.

VMs in Google Cloud. We use 50 VMs of type `n1-standard-4` [6]. Each VM has 4 vCPUs, 15 GB memory, and 10 Gbps network bandwidth. The OS is Ubuntu 18.04 LTS with Linux kernel version 5.0.

Bare-metal cloud in CloudLab. CloudLab [26] is an open testbed for research on cloud computing. We use the `m510` cluster [10] in CloudLab, which has 270 servers in 6 racks, 6 top-of-the-rack (ToR) switches, and 1 spine switch. The bandwidth is 10 Gbps between each server and ToR, and 4×40 Gbps between each ToR and the spine switch. Each ToR switch has a 9 MB shared buffer. Each server has an 8-core Intel Xeon CPU, 64GB memory, and a Mellanox ConnectX-3 10 Gbps NIC. For the On-Ramp evaluation, we rented 100 servers in this cluster (randomly chosen from these 6 racks by Cloudlab), and installed Ubuntu 18.04 LTS with Linux kernel version 4.15 on each server.

ns-3. We implement On-Ramp in ns-3 based on the open-source ns-3 simulator of HPCC [4,44]. We also use the same simulation setup as in [44, §5.1]. There are 320 servers in 20 racks, 20 aggregation switches and 16 core switches. The network topology is a 3-stage FatTree [12], consisting of ToR, aggregation and core switches. Each server has a 100 Gbps link connected to the ToR switch. All links between core, aggregation and ToR switches are 400 Gbps. Each link has 1 μs delay.[5] Each switch has a 32 MB shared buffer. Because

[5] Consisting of the link propagation delay and the packet processing delay in the corresponding switch.

the ns-3 simulations consider RDMA flows, we have priority flow control (PFC) [38] in effect. Note that neither the Google Cloud nor the CloudLab experiments has PFC; hence, buffers overflow and result in packet drops.

5.1.2 Traffic loads

We have two categories of traffic: incast type traffic [51] where requests generate bursts of small equal-sized flows simultaneously, and background traffic consisting of flows of varying sizes.

The incast traffic has a fanout of 40, where each of the 40 flows is either 2KB [13] or 500KB [44], so the total request size is 80KB and 20MB, respectively. The 2KB-sized flows are common in query-type scenarios while the 500KB-sized flows occur in query- and storage-type settings. The average load due to incast traffic is 2% or 20%.

For the background traffic, we use three data center workloads, namely: (1) *WebSearch* [13]: the web search traffic measured in Microsoft production clusters; (2) *FB_Hadoop* [50]: the traffic measured in Hadoop clusters at Facebook; (3) *GoogleSearchRPC* [5,49]: the RPC traffic generated by search applications at Google. Figure 8 shows the distribution of flow sizes. We adjust the average interval between adjacent flows to make the average traffic load to be 40%, 60% or 80%.

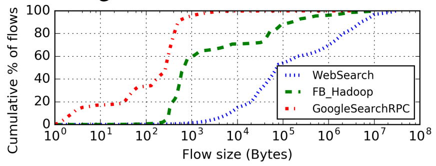


Figure 8: Distribution of flow sizes in the background traffic

5.1.3 Clock synchronization and packet timestamping

On-Ramp needs clock synchronization across servers to measure OWD. For bare-metal cloud and VMs in the public cloud, we use Huygens [30] as the clock synchronization algorithm. We find that the standard deviation of clock offsets after synchronization is around 200 ns in Google cloud, 100 ns in CloudLab, and the 99th percentile is less than 3 μs in both cases. In ns-3, by default the clocks are perfectly synchronized. However, to mimic clock inaccuracy in the real world even under good clock synchronization, we add a random offset to each server's clock according to a Gaussian distribution with a standard deviation of 200 ns.

For VMs in Google Cloud, the packet timestamps are taken inside the VMs with system clocks. In CloudLab, although the ConnectX-3 NIC supports hardware timestamping, we use software timestamps provided by the system clocks so that we can compare a public cloud and a bare-metal cloud by making the CloudLab setup as close as possible to a bare-metal cloud.

5.1.4 Selection of On-Ramp parameters

The EWMA gain g is set to $1/16$ (§3.2). The threshold T should be higher than the minimal OWD on the path, plus some headroom to tolerate errors in OWD measurements.

In Google Cloud, (i) as reported in [23], the VM-VM minimal RTT for TCP traffic is typically 25 μs, so the minimal OWD is less than that; (ii) the inaccuracy of Tx and Rx timestamps can be around 50 μs due to GSO in VM on Tx side and the merging of Rx segments in the hypervisor; (iii) the high percentile clock sync inaccuracy under Huygens is less than 3 μs. Taken together, we pick $T = 150\mu s$ to be safe.

In Cloudlab, following similar steps, we pick $T = 50\mu s$, because the minimal OWD is smaller, and there is no VM hypervisor involved in Cloudlab evaluations.

In ns-3, the minimal OWD is up to 6 μs inside the network. It emulates a NIC implementation of On-Ramp so the timestamps are accurate. To be safe, we pick $T = 6 + 10 = 16\mu s$.

§7.4 shows that the end-to-end performance of On-Ramp is only mildly sensitive to the value of T and g.

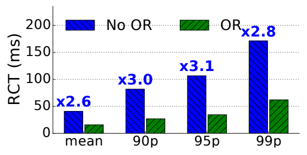
5.2 On-Ramp Performance

5.2.1 Google Cloud Platform (GCP)

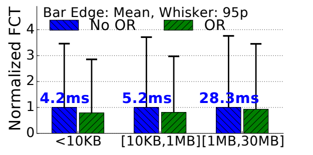
We first consider the **following basic scenario** in GCP: *WebSearch* traffic at 40% load, and an incast load at 2% with a fanout of 40 where each of the 40 flows is 2KB (on each server, the average interval between two consecutive incast requests is 3.2 ms), and CUBIC congestion control.

Incast RCT. As can be seen in Figure 9a, the mean, 90th, 95th and 99th percentile RCTs of an incast request are reduced by 2.6×, 3.0×, 3.1× and 2.8×, respectively.

Background traffic FCT. We group background traffic flows into three buckets by size: small (≤10KB), medium (10KB–1MB), and large (1MB–30MB) flows. On-Ramp improves the mean FCT of the *WebSearch* background traffic flows by 21%, 19%, and 7% compared to the baseline for the flows of small, medium and large sizes respectively. The 95th percentile FCT shows similar improvements. This means that time-critical short flows get lower FCTs and the throughput of long flows does not degrade. Thus, On-Ramp *does not adversely affect* the congestion control of the long flows; indeed, it even improves it mildly by reducing packet drops and timeouts, so the long flows avoid unnecessary window cuts in CUBIC algorithm. See Figure 9b.[6] Finally, the percentage of packets retransmitted reduces from 0.0693% to 0.0314%.



(a) Incast RCT



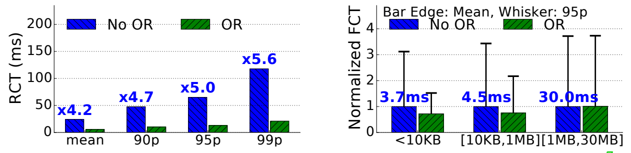
(b) FCT of Background Traffic, normalized by mean FCT without OR[7]

Figure 9: Cloud VM, CUBIC, *WebSearch* at 40% load + Incast at 2% load (fanout=40, size of each flow=2KB).

[6]Note that, for this and all following experiments in GCP and CloudLab, the baseline "No OR" already includes the benefits of per-flow queueing and round-robin dequeueing, because the default QDisc (FQ_CoDel [9] for CUBIC and FQ [25] for BBR) in Ubuntu 18.04 provides this functionality. Therefore, the performance gain from "no OR" to "OR" purely comes from the On-Ramp algorithm.

Varying congestion control algorithms. We generalize the basic scenario by swapping out CUBIC for BBR [19]. BBR is an out-of-box alternative to CUBIC in VMs of public clouds. Figure 10a shows the mean and tail RCTs of an incast request are reduced by On-Ramp by 4.2× – 5.6×. On-Ramp also reduces the mean FCT of the *WebSearch* background traffic flows by 28% and 25% for the small and medium sizes respectively, and tail FCT by 51% and 37%, while maintaining the same performance as the baseline for the large flows, as shown in Figure 10b. In this scenario, the fraction of packets retransmitted reduces by 21×, from 0.0105% to 0.0005%.

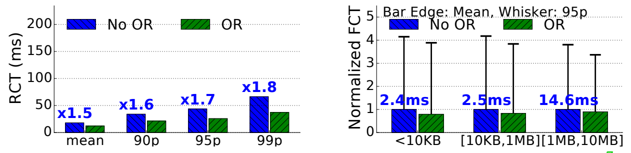
Note that BBR gives smaller RCTs of incast requests and FCTs of short flows than CUBIC because it controls delays. On-Ramp is able to further improve BBR's performance under the incast traffic.



(a) Incast RCT **(b)** FCT of Background Traffic[7]

Figure 10: Cloud VM, **BBR**, *WebSearch* at 40% load + Incast at 2% load (fanout=40, size of each flow=2KB).

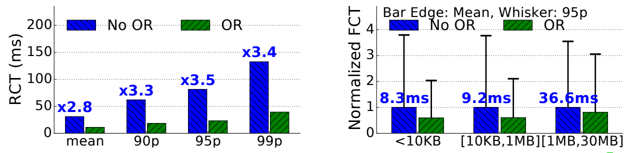
Varying background traffic pattern. Next, we consider the *FB_Hadoop* traffic at 40% load with an incast load as above and the CUBIC algorithm. Figure 11 shows the results.



(a) Incast RCT **(b)** FCT of Background Traffic[7]

Figure 11: Cloud VM, CUBIC, **FB_Hadoop** at 40% load + Incast at 2% load (fanout=40, size of each flow= 2KB).

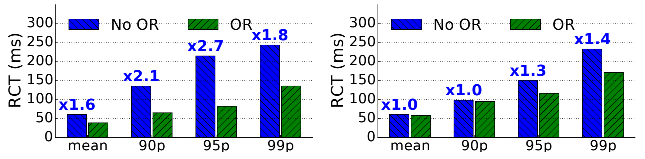
Varying the load level of background traffic. To test the robustness of On-Ramp under high load, we increase the background traffic to 80% load in the basic scenario, leaving everything else fixed. The results are in Figure 12. The performance gains achieved by On-Ramp are larger under higher loads. Figure 25 in Appendix shows the results at 60% load.



(a) Incast RCT **(b)** FCT of Background Traffic[7]

Figure 12: Cloud VM, CUBIC, *WebSearch* at **80% load** + Incast at 2% load (fanout=40, size of each flow 2KB).

Varying the pattern and load level of incast traffic. Finally, in order to understand the effect of On-Ramp on RCT of larger incast requests (modeling storage-type traffic), we increased the size of incast flows from 2KB to 500KB in the basic scenario at incast loads of 20% and 2%. The findings are shown in Figure 13. Also, the total number of timeouts is reduced by 21× and 13× in the cases of 20% and 2% incast load respectively.



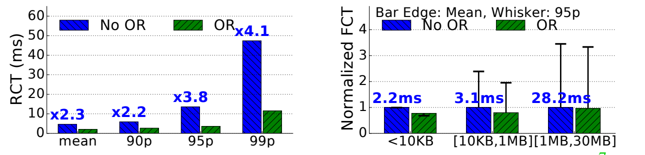
(a) Incast RCT (20% Incast Load) **(b)** Incast RCT (2% Incast Load)

Figure 13: Cloud VM, CUBIC, *WebSearch* at 40% load + Incast (fanout=40, **size of each flow 500KB**).

This scenario highlights an interesting aspect of On-Ramp, namely, that the potential one RTT delay in obtaining OWD measurement when a new flow starts can be avoided by using the OWD measurements from previous flows. Specifically, under 2% load, the average interval between two consecutive incast requests is 800 ms, which is which is about one order of magnitude higher than the average RCT for requests of size 20MB (500KB × 40). Therefore, the gap between two request responses is large, and each request starts by getting new OWD measurements. When the incast load is 20%, the average inter-request interval (80 ms) is comparable to the average RCT. Therefore, a new request is able to leverage the OWD measurement of the previous one and help On-Ramp detect and throttle the incast episodes.[8]

5.2.2 CloudLab

For the evaluation on CloudLab, we consider the basic scenario described in the GCP evaluation except that the *WebSearch* traffic is at 60% load. As can be seen in Figure 14a, we observe similar improvements as GCP: the mean and tail RCTs of an incast request improves by 2.3× – 4.1×. The mean FCTs of the *WebSearch* background traffic flows are improved by 23%, 20%, 4% across flows of small, medium and large sizes respectively, as shown in Figure 14b.



(a) Incast RCT **(b)** FCT of Background Traffic[7]

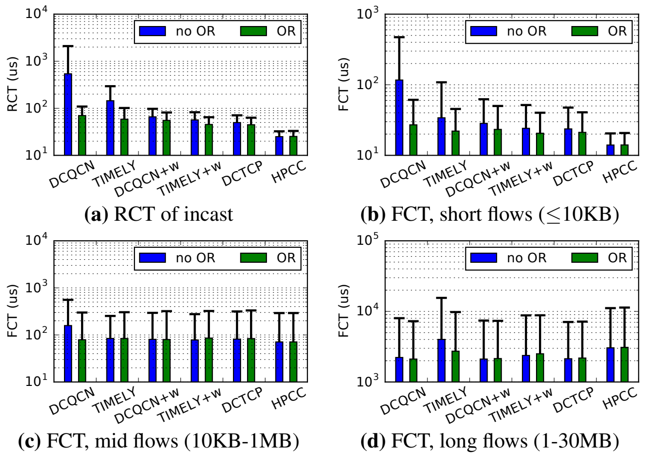
Figure 14: Bare-metal, CUBIC, *WebSearch* at 60% load + Incast at 2% load (fanout=40, size of each flow=2KB).

5.2.3 Large-scale ns-3 simulations

To understand the performance of On-Ramp when combined with recently developed congestion control schemes which use detailed network congestion information, we use ns-3 simulations. We use *WebSearch* traffic at 60% load, plus incast with a fanout of 40 and flow sizes of 2KB at 2% load. As

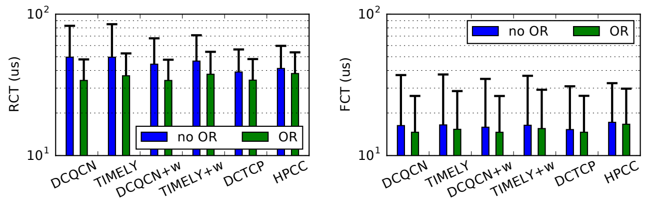
[7]Fig. 10b, 11b, 12b, 14b, 19b has the same normalization as Fig. 9b.

[8]Note that when the incast flow sizes are 2KB and the load is 2%, the average inter-request interval is 3.2 ms, 250 times smaller than when the flow sizes are 500KB. Therefore, in this case, an On-Ramp sender-side module *does* receive frequent-enough OWD measurements even at low load.



(a) RCT of incast **(b)** FCT, short flows (≤10KB) **(c)** FCT, mid flows (10KB-1MB) **(d)** FCT, long flows (1-30MB)

Figure 15: ns-3, *WebSearch* of 60% load + incast of 2% load. Bars: mean, whiskers: 95th percentile. *Y-axis in log.*



(a) RCT of incast **(b)** FCT, GoogleSearchRPC flows

Figure 16: ns-3, **GoogleSearchRPC** of 60% load + incast of 2% load. Bars: mean, whiskers: 95th percentile. *Y-axis in log.*

a reminder, to mimic realistic deployments, PFC is in effect and each server's clock is jittered by an additional random offset according to a Gaussian distribution with a standard deviation of 200 ns, so OWD measurement is not precise. The congestion control algorithms used are DCQCN, TIMELY, DCTCP and HPCC. Following [44], a sending window is added to DCQCN and TIMELY to limit the bytes-in-flight. These algorithms are called DCQCN+w and TIMELY+w.

As seen in Figure 15, the mean and tail RCT of incast traffic is reduced significantly for DCQCN, TIMELY, DCQCN+w, TIMELY+w and DCTCP. The performance of HPCC is not significantly improved because it utilizes recent and detailed congestion information from the network elements and is already highly performant. Further, the FCT of the *WebSearch* flows is improved across all categories, including the large flows (1MB-30MB). Again, the extent of improvement is algorithm-specific.

Then, we change the background traffic to *FB_Hadoop*, leaving the other settings the same. The improvement given by On-Ramp is similar to the above. See Appendix Figure 26.

Next, we consider the *GoogleSearchRPC* workload. Since this traffic has mostly (>99.85%) small flows (≤10KB), we simply consider the mean and high percentile FCT across all flows rather than categorizing by flow size, as shown in Figure 16. This workload is challenging for most congestion control algorithms because nearly 80% of bytes are due to flows under 10 KB in size—too short for congestion control algorithms to address.[9] However, just by more efficiently controlling the transient events caused by the remaining 20% of bytes from the relatively long flows, On-Ramp improves the performance of all algorithms, including HPCC.

6 Evaluation in Facebook's Network

We have also evaluated On-Ramp at Facebook, where On-Ramp was used to throttle large, high-bandwidth storage file transfers so that they don't eat up all the switch buffers, causing severe packet drops for latency-sensitive compute application traffic. This scenario is canonical in data centers where multiple types of traffic with different objectives share the same network fabric. Our goal is to use On-Ramp to ensure storage traffic gets the bandwidth it needs while not affecting the latency of the compute applications.

Environment. Two racks inside a Facebook production cluster are used in the evaluation. As a typical setup, in the first rack, 15 machines work as application clients and 12 work as storage clients. In the second rack, 30 machines work as application servers and 6 work as storage servers. The two ToR switches are connected to 3 spine switches. The link bandwidth is 100 Gbps for each storage server, 25 Gbps for all other machines, and 100 Gbps between each ToR and each spine switch. Huygens runs on all machines to synchronize their clocks.

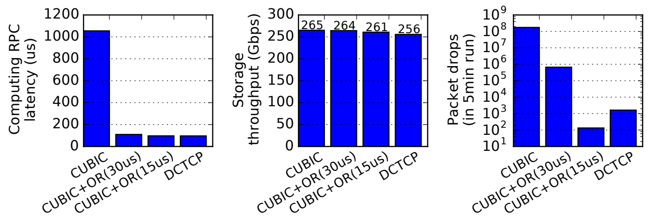
Traffic loads. Two types of traffic are run simultaneously: (1) Computing traffic: They are latency-sensitive short flows carrying RPCs generated by computing jobs. They run between the application clients and servers. (2) Storage traffic: Each pair of storage clients read files from one storage server via NVMe-over-TCP [11], which generates throughput-sensitive long flows consisting of 16–128KB bursts. When 12 or more storage clients are reading, the total amount of storage traffic requested will be $25 \times 12 = 300$ Gbps or more, enough to saturate the uplinks from the second ToR switch to the spine switches. Severe congestion happens at this load.

Results. Figure 17 shows the results when 12 storage clients are reading from storage servers.[10] When using the default CC CUBIC, the latency of computing RPCs is severely hurt by the storage traffic, and numerous packets are dropped. Deploying On-Ramp with CUBIC reduces the latency of computing RPCs by 10× while maintaining the throughput of storage traffic. The number of packets dropped is reduced by about 260×. Here we pick On-Ramp threshold T=30μs following the guideline in §3.2. The more aggressive setting of T=15μs reduces the packet drops even more while only marginally reducing the storage throughput, as shown in the figure.

On-Ramp's performance was compared with DCTCP as well, and we found that, under saturation loading of storage traffic (300 Gbps), both achieve a similar performance in

[9]DCQCN, TIMELY and HPCC have no slow start phase. Following [44], DCTCP's slow start phase is removed for fair comparisons.

[10]See Appendix §10.3 for results with 0–12 storage clients.



(a) Latency **(b)** Throughput **(c)** Packet drops

Figure 17: On-Ramp's performance in a Facebook cluster

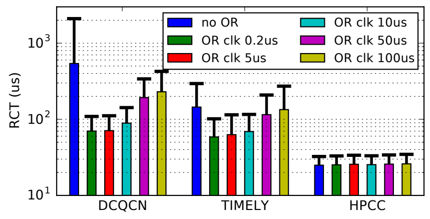


Figure 18: RCT of incast traffic under different levels of clock inaccuracy. Bars: mean, whiskers: 95th percentile.

terms of compute application latency,[11] but DCTCP achieves a slightly lower storage throughput compared to On-Ramp + CUBIC. A drawback of DCTCP is that it needs ECN marking at all the switches which is not only operationally burdensome, but challenging when many non-DCTCP flows (e.g., flows whose source or destination is an external server) share switches with the DCTCP flows. Being purely edge-based, On-Ramp sidesteps these burdens and challenges.

7 On-Ramp Deep Dive

7.1 The Accuracy of OWD Signals

To study the effect of the accuracy of OWD signals on On-Ramp's end-to-end performance, we repeat the ns-3 scenario described in §5.2.3 and consider DCQCN, TIMELY and HPCC. We add a constant random Gaussian offset to each clock of standard deviation σ_{clk}, and vary σ_{clk} to model different levels of clock inaccuracy.

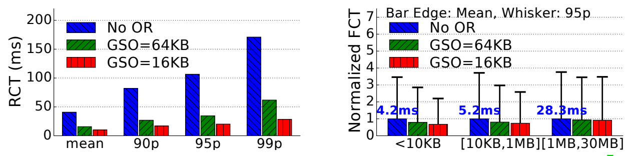
Recall that $0.2\mu s$ is the default σ_{clk} we use throughout our ns-3 evaluation, with a corresponding threshold $T = 16\mu s$. Here, we increase σ_{clk} up to $100\mu s$ and, following the guidelines in §3.2, we change the threshold T according to the formula $T = 2\sigma_{clk} + minOWD$ ($minOWD = 6\mu s$ in ns-3). Essentially, inaccurate clocks lead to inaccurate measurements of OWD which become confounded with path congestion. Choosing a value of T as per the formula above allows for inaccurate clocks. In practice, Huygens reports an estimation of clock inaccuracy via the network effect [30], so we can adapt the threshold T according to it using this formula.

Figure 18 shows the RCT of incast requests under different σ_{clk}. We observe that for DCQCN and TIMELY, as σ_{clk} increases, the mean and tail incast RCT also increases. The performance degradation becomes more significant when σ_{clk} becomes comparable to the OWDs under congestion (roughly $20-100\mu s$). Since HPCC maintains very small queues (due to its bandwidth headroom), it operates well under the threshold T of queuing delay needed to trigger On-Ramp. Hence, On-Ramp doesn't affect its performance.

7.2 The Granularity of Control

As discussed in §4.1, GSO affects the granularity of control by On-Ramp. We study its effect by reducing the max GSO size from the default value of 64 KB to 16 KB. As shown in Figure 19, the mean and 90th, 95th, 99th percentile of incast traffic RCT are *further* reduced by 36%, 37%, 41%, 54%, respectively. The FCT of short and mid-sized flows in the *WebSearch* traffic are further reduced by 8-14% (mean) and 13-22% (95th percentile). The throughput of long flows is well-maintained. However, reducing max GSO size adds more CPU overhead to the sender, so we let the user decide on it.

Remark. This experiment explains the significant performance improvement of On-Ramp in ns-3 when compared to the Google Cloud and CloudLab implementations. In ns-3, we are effectively simulating a NIC implementation where On-Ramp has per-packet control and highly accurate time synchronization, which leads to better performance.



(a) Incast RCT **(b)** FCT of Background Traffic[7]

Figure 19: Cloud VM, CUBIC, *WebSearch* at 40% load + Incast at 2% load (fanout=40, each flow 2KB).

7.3 Co-existence

A public cloud user is unaware of other users and the amount of traffic (not controlled by On-Ramp) they insert into the network. Hence, it is possible that the effect of On-Ramp can be blunted when other traffic is present. In fact, if non-On-Ramp traffic shares links with On-Ramp traffic, the latter may unilaterally do worse because On-Ramp may pause transmission when congestion due to the non-On-Ramp traffic increases. The results in §5.2.1 show that this dire situation may not happen: a cloud user can achieve better performance by enabling On-Ramp in their own VM cluster even though there *may* be non-On-Ramp traffic on their paths. In this section, we revisit this question in the controlled environment of CloudLab.

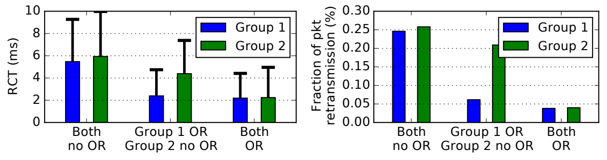
We consider the scenario in §5.2.2 and divide the 100 servers in CloudLab randomly into two groups with 50 servers each. The same workload as in §5.2.2 is run inside each group, but we don't run cross-group traffic. This models 2 users renting servers in a cloud environment unbeknownst to each other. We evaluate the performance in the following cases: (i) both groups do not use On-Ramp, (ii) Group 1 uses On-Ramp but not Group 2, and (iii) both groups use On-Ramp.

Figure 20 summarizes the results. Interestingly, when Group 1 uses On-Ramp and Group 2 does not, *both groups* do

[11]Indeed, both DCTCP and On-Ramp + CUBIC achieve an RPC latency under saturation loading nearly equal to the case when there is no storage traffic, which is the best possible. See Appendix §10.3 for details.

better than when neither uses On-Ramp. Essentially, during periods of congestion, On-Ramp enables Group 1 senders to transmit their traffic at moments when Group 2 traffic is at low load. Conversely, the improvement in Group 2's performance is due to a reduction in overall congestion.

When Group 2 also uses On-Ramp, the improvement in Group 1's performance is only slight. Thus, Group 1 obtains almost the same benefit from using On-Ramp whether or not Group 2 uses it; this is desirable for incremental deployment.

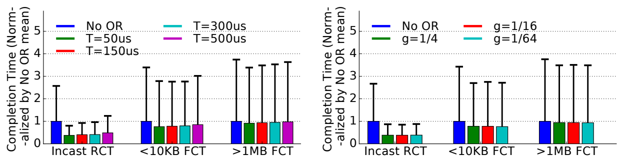


(a) RCT of incast **(b)** Pkt retransmission

Figure 20: Co-existence of traffic with and without OR.

7.4 On-Ramp Parameters T and g

To explore the sensitivity of On-Ramp on the threshold T and EWMA gain g, we run the same basic evaluation scenario described in 5.2.1 in GCP, but now vary T between 50μs and 500μs, and g between $\frac{1}{4}$ and $\frac{1}{64}$. Recall that for GCP, the default parameter values are $T = 150\mu s$ and $g = \frac{1}{16}$. Figure 21a shows that the performance of On-Ramp worsens noticeably as T increases beyond 300 μs. Figure 21b shows that On-Ramp's performance is relatively insensitive to the value of g in the range between $\frac{1}{4}$ and $\frac{1}{64}$.



(a) Threshold T **(b)** EWMA gain g

Figure 21: Changing T and g. RCT and FCT normalized by the mean values of No OR. Bars: mean, whiskers: 95th percentile.

8 Related Work

Congestion control (CC). CC algorithms can be broadly categorized into two groups: (i) those which need no network assistance, e.g., drop-based schemes (TCP NewReno [28], CUBIC [33]) or delay-based schemes (TCP Vegas [18], TIMELY [47], and Swift [42]); and (ii) those which rely on network assistance, e.g., use ECN signals (DCTCP [13], DCQCN [53]), leverage in-band network telemetry (HPCC [44]), or rely on the network's ability to perform some functions like scheduling or trimming packets [16,17,29,31,34,47,49]. On-Ramp is complementary to CC. It is meant to be deployed underneath any CC algorithm, providing a fast and accurate response to transient congestion purely from the edge of the network. Swift [42] is a recent algorithm that uses RTT measurements and carefully chosen delay targets with support for fractional congestion windows to obtain good performance across a wide range of deployment scenarios. Swift couples the handling of equilibrium (on ACK) and transient (on timeout). This coupling is likely to lead to the equilibrium-transient tension mentioned in §2. By contrast, On-Ramp explicitly decouples the two, providing more robust performance.

In-network pause. Schemes such as Priority-based Flow Control (PFC) [38] have been widely deployed to eliminate packet drops in switches. However, PFC causes several safety and performance challenges including PFC deadlocks and congestion spreading [32,36,48,53]. By pausing flows at the edge, On-Ramp avoids these challenges.

Congestion Control (CC) in cloud environments. Previous works like AC/DC TCP [35] and Virtualized Congestion Control (VCC) [22] give cloud admins control over the CC of the users' VMs by translating between the target CC and the VM's CC. Their architectures are similar to On-Ramp: they also operate as a shim layer between the VM applications and the physical network, intercepting packets without requiring network infrastructure changes. However both AC/DC TCP and VCC rely on ECN support from the network infrastructure to implement the target CC (DCTCP) and need to be implemented by the cloud provider within the hypervisor. On-Ramp makes no assumptions of the underlying network infrastructure and can be implemented by cloud users within their VMs.

Flow scheduling in data centers. On-Ramp performs a form of flow scheduling because it pauses packets at the edge for a short period of time. Previous work in this space like pHost [29], NDP [34], and Homa [49] propose algorithms with varying levels of network support to enable flow scheduling within the network and improve end-to-end performance. On-Ramp requires no network support and is done purely at the edge. It can therefore be readily deployed, especially by users of public cloud environments.

9 Conclusion and Future Work

Datacenter packet transport over the last decade has relied increasingly on network support (e.g., ECN marking, queue size information), making it hard to deploy in environments such as the public cloud. We show empirically that the move towards increasingly rich network support is rooted in a tension between equilibrium and transience performance. Motivated by these results, we take a step back and modularize congestion control into two separate components, one responsible for equilibrium and the other for transients. We leave equilibrium handling to existing congestion control algorithms and design a new underlay scheme, On-Ramp, for transient handling. On-Ramp uses one-way delay measurements enabled by synchronized clocks to hold back packets transmitted by any congestion control algorithm at the edge of a network during transient congestion. Intellectually, On-Ramp contains two ideas that are of independent interest. First is the use of synchronized clocks to improve network performance. Second is the factoring of datacenter congestion control—traditionally a single control loop—into two separate control loops, one each for transience and equilibrium. We hope this paper is the beginning of a more in-depth investigation of both ideas.

Acknowledgments

We thank Vaibhavkumar Patel and Manoj Wadekar at Facebook for their tremendous work and support in conducting the Facebook evaluations. We thank the sponsors of the Platform Lab at Stanford University for generously supporting this research. Ahmad Ghalayini is supported by the Caroline and Fabian Pease Stanford Graduate Fellowship.

References

[1] PTP hardware clock infrastructure for Linux. https://www.kernel.org/doc/Documentation/ptp/ptp.txt, 2011. [Online; accessed 16-Sept-2020].

[2] TCP small queues. https://lwn.net/Articles/507065/, 2012. [Online; accessed 16-Sept-2020].

[3] Mellanox BlueField SmartNIC 25Gb/s Dual Port Ethernet Network Adapter. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_BlueField_Smart_NIC.pdf, 2019. [Online; accessed 16-Sept-2020].

[4] GitHub - alibaba-edu/High-Precision-Congestion-Control. https://github.com/alibaba-edu/High-Precision-Congestion-Control, 2020. [Online; accessed 25-January-2020].

[5] HomaSimulation/Google_SearchRPC.txt at omnet_simulations - PlatformLab/HomaSimulation - GitHub. https://github.com/PlatformLab/HomaSimulation/blob/omnet_simulations/RpcTransportDesign/OMNeT%2B%2BSimulation/homatransport/sizeDistributions/Google_SearchRPC.txt, 2020. [Online; accessed 16-Sept-2020].

[6] Machine types | Compute Engine Documentation | Google Cloud. https://cloud.google.com/compute/docs/machine-types, 2020. [Online; accessed 25-January-2020].

[7] Netronome Agilio SmartNICs. https://www.netronome.com/products/smartnic/overview/, 2020. [Online; accessed 16-Sept-2020].

[8] ns-3 Network Simulator. https://www.nsnam.org/, 2020. [Online; accessed 16-Sept-2020].

[9] tc-fq_codel. http://man7.org/linux/man-pages/man8/tc-fq_codel.8.html, 2020. [Online; accessed 16-Sept-2020].

[10] The Cloud Lab Manual. Chapter 11: Hardware. http://docs.cloudlab.us/hardware.html, 2020. [Online; accessed 16-Sept-2020].

[11] NVMe-oF Specification. https://nvmexpress.org/developers/nvme-of-specification/, 2021. [Online; accessed 08-Feb-2021].

[12] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A Scalable, Commodity Data Center Network Architecture. *SIGCOMM Comput. Commun. Rev.*, 38(4):63–74, August 2008.

[13] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 63–74, New York, NY, USA, 2010. ACM.

[14] Mohammad Alizadeh, Adel Javanmard, and Balaji Prabhakar. Analysis of DCTCP: Stability, Convergence, and Fairness. In *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '11, page 73–84, New York, NY, USA, 2011. ACM.

[15] Mohammad Alizadeh, Abdul Kabbani, Berk Atikoglu, and Balaji Prabhakar. Stability Analysis of QCN: The Averaging Principle. In *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '11, page 49–60, New York, NY, USA, 2011. ACM.

[16] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pFabric: Minimal near-Optimal Datacenter Transport. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 435–446, New York, NY, USA, 2013. ACM.

[17] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Information-Agnostic Flow Scheduling for Commodity Data Centers. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, page 455–468, USA, 2015. USENIX Association.

[18] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *Proceedings of the Conference on Communications Architectures, Protocols and Applications*, SIGCOMM '94, pages 24–35, New York, NY, USA, 1994. ACM.

[19] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-based Congestion Control. *Commun. ACM*, 60(2):58–66, January 2017.

[20] Li Chen, Kai Chen, Wei Bai, and Mohammad Alizadeh. Scheduling mix-flows in commodity datacenters with karuna. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 174–187, 2016.

[21] Inho Cho, Keon Jang, and Dongsu Han. Credit-scheduled delay-bounded congestion control for datacenters. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 239–252, 2017.

[22] Bryce Cronkite-Ratcliff, Aran Bergman, Shay Vargaftik, Madhusudhan Ravi, Nick McKeown, Ittai Abraham, and Isaac Keslassy. Virtualized Congestion Control. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 230–243, New York, NY, USA, 2016. ACM.

[23] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, and et al. Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI'18, page 373–387, USA, 2018. USENIX Association.

[24] Nandita Dukkipati, Masayoshi Kobayashi, Rui Zhang-Shen, and Nick McKeown. Processor Sharing Flows in the Internet. In *Proceedings of the 13th International Conference on Quality of Service*, IWQoS'05, page 271–285, Berlin, Heidelberg, 2005. Springer-Verlag.

[25] Eric Dumazet. pkt_sched: fq: Fair Queue packet scheduler, 2013. [Online; accessed 25-June-2020].

[26] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, and et al. The Design and Operation of Cloudlab. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 1–14, USA, 2019. USENIX Association.

[27] S. Floyd. RFC3649: HighSpeed TCP for Large Congestion Windows, 2003.

[28] S. Floyd and T. Henderson. RFC2582: The NewReno Modification to TCP's Fast Recovery Algorithm, 1999.

[29] Peter X. Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. PHost: Distributed near-Optimal Datacenter Transport over Commodity Network Fabric. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15, pages 1–12, New York, NY, USA, 2015. ACM.

[30] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. Exploiting a Natural Network Effect for Scalable, Fine-grained Clock Synchronization. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI'18, pages 81–94, Berkeley, CA, USA, 2018. USENIX Association.

[31] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. Queues Don't Matter When You Can JUMP Them! In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, page 1–14, USA, 2015. USENIX Association.

[32] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over Commodity Ethernet at Scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 202–215, New York, NY, USA, 2016. ACM.

[33] Sangtae Ha, Injong Rhee, and Lisong Xu. CUBIC: A New TCP-Friendly High-Speed TCP Variant. *SIGOPS Oper. Syst. Rev.*, 42(5):64–74, July 2008.

[34] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. Re-Architecting Datacenter Networks and Stacks for Low Latency and High Performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 29–42, New York, NY, USA, 2017. ACM.

[35] Keqiang He, Eric Rozner, Kanak Agarwal, Yu (Jason) Gu, Wes Felter, John Carter, and Aditya Akella. AC/DC TCP: Virtual Congestion Control Enforcement for Datacenter Networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 244–257, New York, NY, USA, 2016. ACM.

[36] Shuihai Hu, Yibo Zhu, Peng Cheng, Chuanxiong Guo, Kun Tan, Jitendra Padhye, and Kai Chen. Deadlocks in Datacenter Networks: Why Do They Form, and How to Avoid Them. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, HotNets '16, pages 92–98, New York, NY, USA, 2016. ACM.

[37] IEEE. IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. *IEEE Std 1588-2008 (Revision of IEEE Std 1588-2002)*, pages 1–300, July 2008.

[38] IEEE. IEEE Standard for Local and metropolitan area networks–Media Access Control (MAC) Bridges and Virtual Bridged Local Area Networks–Amendment 17: Priority-based Flow Control. *IEEE Std 802.1Qbb-2011*

(Amendment to IEEE Std 802.1Q-2011 as amended by IEEE Std 802.1Qbe-2011 and IEEE Std 802.1Qbc-2011), pages 1–40, Sep. 2011.

[39] Pravein Govindan Kannan, Raj Joshi, and Mun Choon Chan. Precise Time-Synchronization in the Data-Plane Using Programmable Switching ASICs. In *Proceedings of the 2019 ACM Symposium on SDN Research*, SOSR '19, page 8–20, New York, NY, USA, 2019. ACM.

[40] Dina Katabi, Mark Handley, and Charlie Rohrs. Congestion Control for High Bandwidth-Delay Product Networks. *SIGCOMM Comput. Commun. Rev.*, 32(4):89–102, August 2002.

[41] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. Inband network telemetry via programmable dataplanes. In *SIGCOMM '15*. ACM, 2015.

[42] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 514–528, New York, NY, USA, 2020. ACM.

[43] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. Globally Synchronized Time via Datacenter Networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 454–467, New York, NY, USA, 2016. ACM.

[44] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. HPCC: High Precision Congestion Control. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, pages 44–58, New York, NY, USA, 2019. ACM.

[45] Lisong Xu, K. Harfoush, and Injong Rhee. Binary increase congestion control (BIC) for fast long-distance networks. In *IEEE INFOCOM 2004*, volume 4, pages 2514–2524 vol.4, 2004.

[46] D. L. Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on Communications*, 39(10):1482–1493, Oct 1991.

[47] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. TIMELY: RTT-based Congestion Control for the Datacenter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 537–550, New York, NY, USA, 2015. ACM.

[48] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting Network Support for RDMA. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 313–326, New York, NY, USA, 2018. ACM.

[49] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 221–235, New York, NY, USA, 2018. ACM.

[50] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the Social Network's (Datacenter) Network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 123–137, New York, NY, USA, 2015. ACM.

[51] Vijay Vasudevan, Amar Phanishayee, Hiral Shah, Elie Krevat, David G. Andersen, Gregory R. Ganger, Garth A. Gibson, and Brian Mueller. Safe and Effective Fine-Grained TCP Retransmissions for Datacenter Communication. *SIGCOMM Comput. Commun. Rev.*, 39(4):303–314, August 2009.

[52] D. X. Wei, C. Jin, S. H. Low, and S. Hegde. FAST TCP: Motivation, Architecture, Algorithms, Performance. *IEEE/ACM Transactions on Networking*, 14(6):1246–1259, 2006.

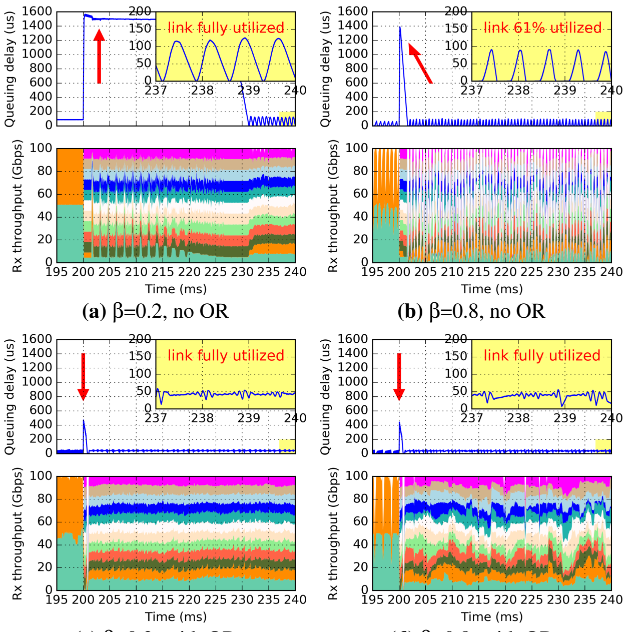
[53] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion Control for Large-Scale RDMA Deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 523–536, New York, NY, USA, 2015. ACM.

[54] Yibo Zhu, Monia Ghobadi, Vishal Misra, and Jitendra Padhye. ECN or Delay: Lessons Learnt from Analysis of DCQCN and TIMELY. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, page 313–327, New York, NY, USA, 2016. ACM.

10 Appendix

10.1 Supplemental Material for §2 and §3

Figures 22 and 23 extend the results shown in Figures 1 and 2 of §2, by displaying the throughputs received by each flow as a stack. When the gain parameter is low (Figs. 22a and 23a), both TIMELY and DCQCN suffer from a long convergence time in transience, during which the flow throughputs are unstable and unfair. Under a high gain parameter, both algorithms under-utilize the link during equilibrium; see Figs. 22b and 23b. When On-Ramp is enabled, both CCs have shorter time in transience and smoother, fairer flow throughputs in equilibrium.



(a) β=0.2, no OR **(b)** β=0.8, no OR **(c)** β=0.2, with OR **(d)** β=0.8, with OR

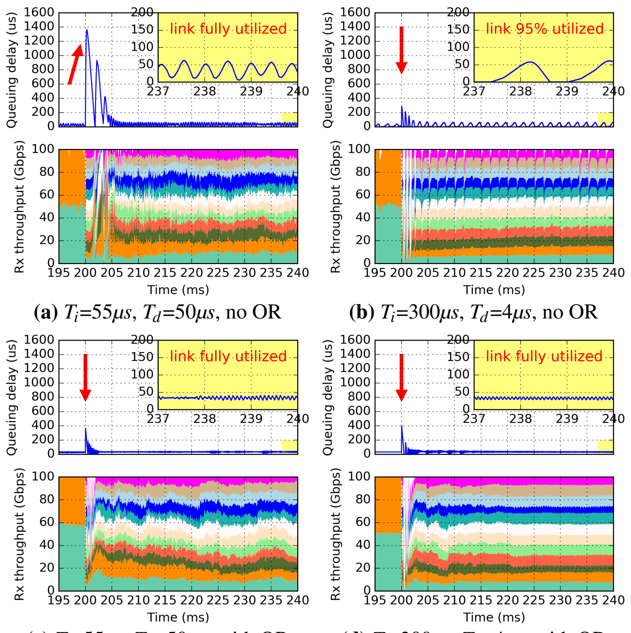
Figure 22: TIMELY study, the red arrow points to transience, and the yellow box is a zoom-in of equilibrium. OR T=30μs.

Figure 24 is a follow-up of Figure 5 in §3.2, which shows the OWDs and the flow rates when 12 CUBIC flows share a bottleneck link in a bare-metal environment (Cloudlab). The threshold T is also 50μs. When using strawman On-Ramp, similar to the case of 2 flows, the queue also suffers from significant undershooting. When using the final version of On-Ramp, most fluctuations in queue lengths are removed, and it achieves better fair sharing among all flows.

10.2 Supplemental Evaluation

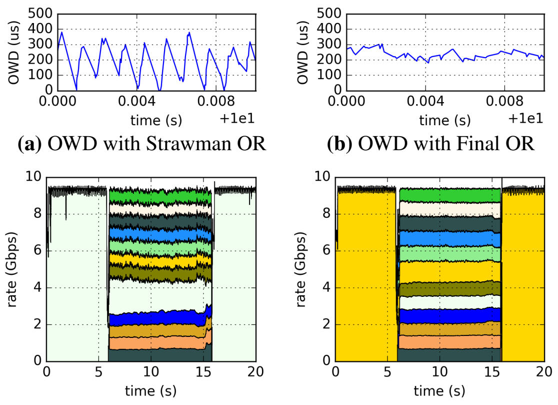
Figure 25 corresponds to the evaluation scenario referred to in §5.2.1 of the main text, where the background *WebSearch* load is 60% for the base scenario in GCP.

Figure 26 corresponds to the ns-3 evaluation in §5.2.3. Here, we run *FB_Hadoop* traffic at 60% load, plus incast with



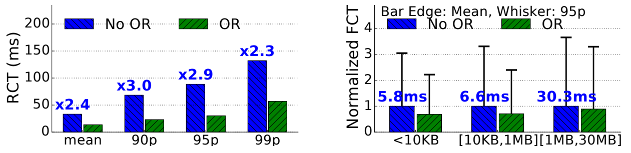
(a) T_i=55μs, T_d=50μs, no OR **(b)** T_i=300μs, T_d=4μs, no OR **(c)** T_i=55μs, T_d=50μs, with OR **(d)** T_i=300μs, T_d=4μs, with OR

Figure 23: DCQCN study. OR T=30μs.



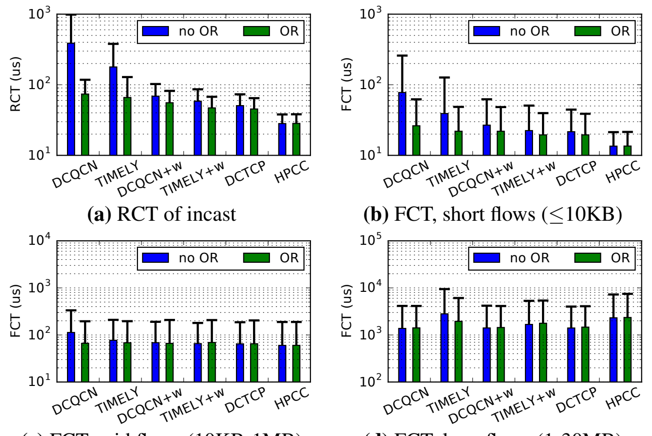
(a) OWD with Strawman OR **(b)** OWD with Final OR **(c)** Flow rates, Strawman OR **(d)** Flow rates, Final OR

Figure 24: 12 long-lived CUBIC flows sharing a link



(a) Incast RCT **(b)** FCT of Background Traffic, normalized by mean FCT without OR

Figure 25: Cloud VM, CUBIC, *WebSearch* at **60% load** + Incast at 2% load (fanout=40, each flow 2KB).



(a) RCT of incast **(b)** FCT, short flows (≤10KB)

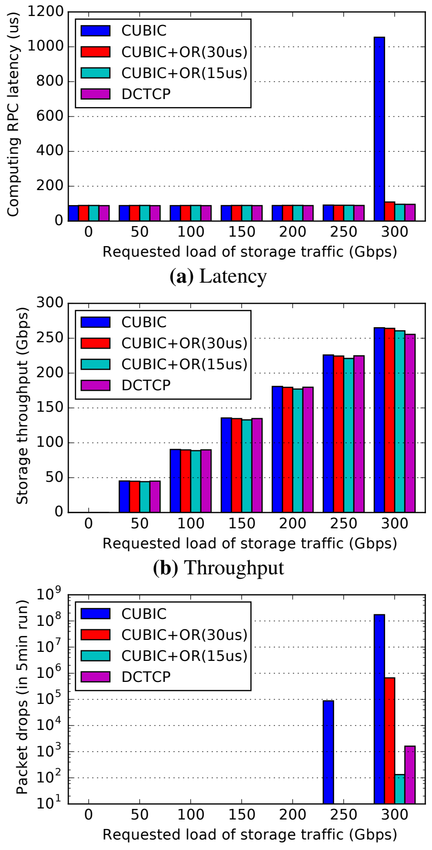
(c) FCT, mid flows (10KB-1MB) **(d)** FCT, long flows (1-30MB)

Figure 26: ns-3, **FBHadoop** of 60% load + incast of 2% load. Bars: mean, whiskers: 95th percentile. *Y-axis in log.*

a fanout of 40 and flow sizes of 2KB at 2% load. The findings are similar to the experiment with *WebSearch* workload (Figure 15). With On-Ramp, the RCT of incast requests and FCT of short flows in background traffic are significantly reduced, while the throughput of long flows is well-maintained (even improved e.g. in TIMELY). Again, the extent of improvement is algorithm-specific.

10.3 Supplemental Results: Facebook

Figure 27 corresponds to the evaluation in Facebook described in §6. The number of storage clients reading from storage servers is set to be 0, 2, 4, ..., 12, so the requested load of storage traffic is 0, 50, 100, ..., 300 Gbps respectively. When the requested load is less than or equal to 250 Gbps, the network is not congested yet, CUBIC, CUBIC + On-Ramp and DCTCP give similar performances. When the requested load hits 300 Gbps, the computing RPC latency shoots up dramatically under CUBIC, meaning it is severely hurt by the storage traffic. Using CUBIC + On-Ramp or DCTCP brings the latency down to the level similar to the non-congested case, while keeping the throughput of storage traffic well-maintained, as we discussed in §6.



(a) Latency

(b) Throughput

(c) Packet drops

Figure 27: On-Ramp's performance in a Facebook cluster

Running BGP in Data Centers at Scale

Anubhavnidhi Abhashkumar[♯*†], Kausik Subramanian[♯*], Alexey Andreyev[◇], Hyojeong Kim[◇], Nanda Kishore Salem[◇], Jingyi Yang[◇], Petr Lapukhov[◇], Aditya Akella[♯], Hongyi Zeng[◇]

University of Wisconsin - Madison[♯], Facebook[◇]

Abstract

Border Gateway Protocol (BGP) forms the foundation for routing in the Internet. More recently, BGP has made serious inroads into data centers on account of its scalability, extensive policy control, and proven track record of running the Internet for a few decades. Data center operators are known to use BGP for routing, often in different ways. Yet, because data center requirements are very different from the Internet, it is not straightforward to use BGP to achieve effective data center routing.

In this paper, we present Facebook's BGP-based data center routing design and how it marries data center's stringent requirements with BGP's functionality. We present the design's significant artifacts, including the BGP Autonomous System Number (ASN) allocation, route summarization, and our sophisticated BGP policy set. We demonstrate how this design provides us with flexible control over routing and keeps the network reliable. We also describe our in-house BGP software implementation, and its testing and deployment pipelines. These allow us to treat BGP like any other software component, enabling fast incremental updates. Finally, we share our operational experience in running BGP and specifically shed light on critical incidents over two years across our data center fleet. We describe how those influenced our current and ongoing routing design and operation.

1 Introduction

Historically, many data center networks implemented simple tree topologies using Layer-2 spanning tree protocol [5, 11]. Such designs, albeit simple, had operational risks due to broadcast storms and provided limited scalability due to redundant port blocking. While centralized software-defined network (SDN) designs have been adopted in wide-area networks [28, 29] for enhanced routing capabilities like traffic engineering, a centralized routing controller has additional scaling challenges for modern data centers comprising thousands of switches, as a single software controller cannot react quickly to link and node failures. Thus, as data centers grew, one possible design was to evolve into a fully routed Layer-3 network, which requires a distributed routing protocol.

[*]Work done while at Facebook. Authors contributed equally to this work.
[†]Currently works at ByteDance.

Border Gateway Protocol (BGP) is a Layer-3 protocol which was originally designed to interconnect autonomous Internet service providers (ISPs) in the global Internet. BGP has supported the Internet's unfettered growth for over 25 years. BGP is highly scalable, and supports large topologies and prefix scale compared to intra-domain protocols like OSPF and ISIS. BGP's support for hop-by-hop policy application based on communities makes it an ideal choice for implementing flexible routing policies. Additionally, BGP sessions run on top of TCP, a transport layer protocol that is used by many other network services. Such explicit peering sessions are easy to navigate and troubleshoot. Finally, BGP has the support of multiple mainstream vendors, and network engineers are familiar with BGP operation and configuration. Those reasons, among others, make BGP an attractive choice for data center routing.

BGP being a viable routing solution in the data center (DC) networks has been well known in the industry [11]. However, the details of a practical implementation of such a design have not been presented by any large-scale operator before. This paper presents a first-of-its-kind study that elucidates the details of the scalable design, software implementation, and operations. Based on our experience at Facebook, we show that BGP can form a robust routing substrate but it needs tight co-design across the data center topology, configuration, switch software, and DC-wide operational pipeline.

Data center network designers seek to provide reliable connectivity while supporting flexible and efficient operations. To accomplish that, we go beyond using BGP as a mere routing protocol. We start from the principles of configuration *uniformity* and operational *simplicity*, and create a baseline connectivity configuration (§2). Here, we group neighboring devices at the same level in the data center as a peer group and apply the same configurations on them. In addition, we employ a uniform AS numbering scheme that is reused across different data center fabrics, simplifying ASN management across data centers. We use hierarchical route summarization on all levels of the topology to scale to our data center sizes while ensuring forwarding tables in hardware are small. Our policy configuration (§3) is tightly integrated with our baseline connectivity configuration. Our policies ensure reliable communication using route propagation scopes and predefined backup paths for failures. They also allow us to maintain the network by seamlessly diverting traffic from problematic/faulty devices in a graceful fashion. Finally, they ensure services remain reachable even when an instance of the service gets added, removed, or migrated.

While BGP's capabilities make it an attractive choice for routing, past research has shown that BGP in the Internet suffers from convergence issues [33, 37], routing instabilities [32], and frequent misconfigurations [21, 36]. Since we control all routers in the data center, we have flexibility to tailor BGP to the data center which wouldn't be possible to achieve in the Internet. We show how we tackled common issues faced in the Internet by fine-tuning and optimizing BGP in the data center (§4). For instance, our routing design and predefined backup path policies ensure that under common link/switch failures, switches have alternate routing paths in the forwarding table and do not send out fabric-wide re-advertisements, thus avoiding BGP convergence issues.

To support the growing scale and evolving routing requirements, our switch-level BGP agent needs periodic updates to add new features, optimization, and bug fixes. To optimize this process, i.e., ensure fast frequent changes to the network infrastructure to support good route processing performance, we implemented an in-house BGP agent (§5). We keep the codebase simple and implement only the necessary protocol features required in our data center, but we do not deviate from the BGP RFCs [6–8]. The agent is multi-threaded to leverage multi-core CPU performance of modern switches, and leverages optimizations like batch processing and policy caches to improve policy execution performance.

To minimize impact on production traffic while achieving high release velocity for the BGP agent, we built our own testing and incremental deployment framework, consisting of unit testing, emulation and canary testing (§6.1). We use a multi-phase deployment pipeline to push changes to agent (§6.2). We find that our multi-phase BGP agent pushes ran for 52% of the time in a 12 month duration, highlighting the dynamic nature of the BGP agent in our data center.

In spite of our tight co-design, simplicity, and testing frameworks, network outages are unavoidable. On the operational side, we discuss some of the significant BGP-related network outages known as SEVs [38] that occurred over two years (§6.3)—these outages were either caused by incorrect policy configurations, bugs in the BGP agent software, or interoperability issues between different agent versions during the deployment of the new agent. Using our operational experience, we discuss current directions we are pursuing in extending policy verification and emulation testing to improve our operational framework, and changing our routing design to support weighted load-balancing to address load imbalances under maintenance/failures.

Contributions.

- We present our novel BGP routing design for the data center which leverages BGP to achieve reliable connectivity along with operational efficiency.
- We describe the routing policies used in our data center to enforce reliability, maintainability, scalability, and service reachability.
- We show how our data center routing design and policies overcome common problems faced by BGP in Internet.
- We present our BGP operational experience, including the benefits of our in-house BGP implementation and challenges of pushing BGP upgrades at high release velocity.

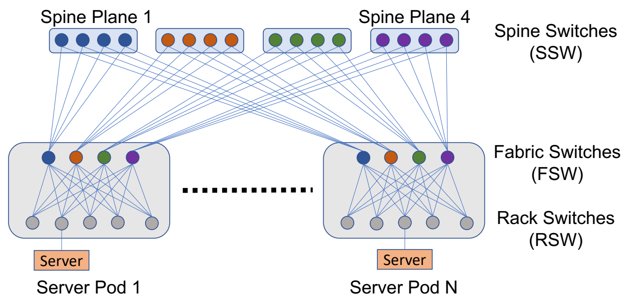


Figure 1: Data Center Fabric Architecture

2 Routing Design

Our original motivation in devising a routing design for our data center was to *build* our network quickly while keeping the routing design *scalable*. We sought to create a network that would provide high availability for our services. However, we expected failures to happen - hence, our routing design aimed to minimize the blast radius of those.

In the beginning, BGP was a better choice for our needs compared to a centralized SDN routing solution for a few reasons. First, we would have needed to build the SDN routing stack from scratch with particular consideration for scalability and reliability, thus, hindering our deployment pace. Simultaneously, BGP has been demonstrated to work well at scale; thus, we could rely on a BGP implementation running on third-party vendor devices. As our network evolved, we gradually transitioned to our custom hardware [18] and in-house BGP agent implementation. This transition would have been challenging to achieve without using a standardized routing solution. With BGP, both types of devices were able to cooperate in the same network seamlessly.

At the time, BGP was a better choice for us compared to the Interior Gateway protocols (IGP) like Open Shortest Path First (OSPF) [39] or Intermediate System to Intermediate System (ISIS) [25]. The scalability of IGPs at scale was unclear, and the IGPs did not provide the flexibility to control route propagation, making it harder to manage failure domains.

We used BGP as the sole protocol and did not pursue a hybrid BGP-IGP routing design as maintaining multiple protocols would add to the complexity of the routing solution. Our routing design builds on the eBGP (External BGP) peering model: Each switch is a BGP speaker and the neighboring BGP speakers are in different autonomous systems (AS). In this section, we provide an overview of our BGP-based routing design catered for our scalable data center fabric topology.

2.1 Topology Design

Application requirements evolve constantly, and our data center design must be capable of scaling out and handling additional demand in a seamless fashion. To this end, we adopt a modular data center fabric topology design [4], which is a collection of *server pods* interconnected by multiple parallel *spine planes*. We illustrate our topology in Figure 1.

A server pod is the smallest unit of deployment, and it has the following properties: (1) each pod can contain up to 48 server racks, and thus, up to 48 rack switches (RSWs), (2) each pod is serviced by up to 16 fabric switches (FSWs), and (3) each rack switch connects to all FSWs in a pod.

Multiple spine planes interconnect the pods. Each plane has multiple spine switches (SSW) connecting to FSWs using uniform high-bandwidth links (FSW-SSW). The number of spine planes corresponds to the number of FSWs in one pod. Each spine plane provides a set of disjoint end-to-end paths between a collection of server pods. This modular design enables us to scale server capacity and network bandwidth as needed—we can increase compute capacity by adding new server pods, while inter-pod bandwidth scales by adding new SSWs on planes.

2.2 Routing Design Principles

We employ two guiding design principles in our DC-wide BGP-based routing design: *uniformity* and *simplicity*. We realize these principles by tightly integrating routing design and configuration with the above topology design.

We strive to minimize the BGP feature set and establish repeatable configuration patterns and behaviors throughout the network. Our BGP configuration is homogeneous within each network tier (RSW, FSW, SSW). The devices serving in the same tier have the same configuration and policies, except for the originated prefixes and peer addresses.

We generate the network topology data and configuration which includes port-maps, IP addressing, BGP, and routing policy configurations for our switches irrespective of the underlying switch platforms. The abstract generic configurations are then translated into the target platform's configuration syntax by our automation software. This ensures that we can easily adapt to changing hardware capabilities in the data center. The details of our configuration management and platform-specific syntax generation can be found in Robotron [44].

2.3 BGP Peering & Load-Sharing

Peering. For uniformity and simplicity in configuration and operations, we treat the whole set of the BGP peers of the same adjacent tier (RSW/FSW/SSW) on a network switch as an *atomic group*, called peer group. Each data center switch connects to groups of devices on each adjacent tier. For example, a FSW aggregates a set of RSWs and has uplinks to multiple SSWs—this makes two distinct peer groups. All BGP peering sessions between adjoining device tiers—for example RSW↔FSW and FSW↔SSW—utilize the same protocol features, timers, and other parameters. Thus, all peers within a group operate in a uniform fashion.

We apply BGP configuration and routing policies on a peer group level. Individual BGP peer sessions belong to a peer group and do not have any additional configuration information beside the neighbor specification. This grouping helps us to simplify configuration and streamline processing of routing updates, as all peers in the same group have identical policies.

For peering, we use direct single-hop eBGP sessions with BGP NEXT_HOP attribute, set to the remote end of the point-to-point subnet. This makes the link usable for BGP routing purposes as soon as it is up. If there are multiple parallel links between the devices, we treat them as individual point-to-point Layer-3 subnets with corresponding BGP sessions. This design allows us to clearly associate BGP sessions with the corresponding network interfaces and simplifies RIB (routing information base) and FIB (forwarding information base) navigation, manipulation, and troubleshooting.

Load-Sharing. To support load-sharing of traffic along multiple paths in the data center, we use BGP with Equal Cost Multipath (ECMP) feature. Each switch forwards traffic equally among paths with equivalent attributes according to BGP best path selection and routing policy in effect. With the presence of multiple paths of equal cost, the vast majority of the switch FIB programming involves removing next hops (when failure occurs) or adding them back (when switch/link comes back up) in the existing ECMP groups. Updating ECMP groups in the FIB is a lightweight and simple operation.

We do not currently use weighted load-balancing inside our data centers for various reasons. Our fabric topology is highly symmetric with wide ECMP groups. We provision the bandwidth uniformly to maximize flexibility of dynamic service placement in the data center. Coupled with the design of our failure domains, this ensures sufficient capacity for services under most common failure scenarios. Moreover, WCMP [48] requires more hardware resources due to the replication of next-hops to perform weighted load-balancing. This does not align well with our goal of minimizing the FIB size requirements in hardware.

2.4 AS Numbering

Following the design principles of uniformity and simplicity, we design a uniform AS numbering scheme for the topology building blocks, such as server pods and spine planes. Our AS numbering scheme is canonical, i.e., the same AS numbers can be reused across data centers in the same fashion. For example, each SSW in the first spine plane in each data center would have the same AS number (e.g., AS 65001). Similarly, the RSWs and FSWs in every server pod of every data center share the same AS numbering structure. To accomplish this goal, we leverage BGP confederations [7]. A confederation

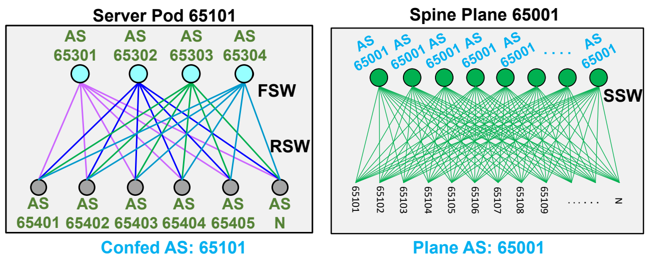


Figure 2: BGP Confederation and AS Numbering scheme for server pods and spine planes in the data center.

divides an AS into multiple sub-ASes such that the sub-ASes and internal paths between them are not visible to the BGP peers outside the confederation.

The uniformity facilitated by our use of confederations and the reusable ASNs (as opposed to a flat routing space) establishes well-structured AS_PATHs for policies and automation. This also helps operators to reason about a routing path easily by inspecting a given AS_PATH during troubleshooting. Inside the data center, we utilize the basic two-octet Private Use AS Numbers, which are sufficient for our design.

Server Pod. To create a reusable ASN structure for server pods—the most numerous building blocks inside our data center network—we implement each server pod as a BGP Confederation. Inside the pod, we allocate unique internal confederation-member ASNs for each FSW and each RSW. We then peer between the devices in a fashion similar to eBGP. The structure of these internal sub-AS numbers repeats within each pod. We assign a unique private AS number per pod (Pod ASN) within a data center as a Confederation Identifier ASN, which is how the pod presents itself to the data center spine and servers. The numbering pattern of unique pod Confederation Identifier ASNs repeats across different data centers. In Figure 2, in each pod, RSWs are numbered from ASN 65401 to N, FSWs are numbered from ASN 65301 to ASN 65304, and server pods are numbered as Confederation Identifier ASN 65101, 65102 and so on.

Spine Plane. Each spine plane in the data center fabric has its own unique (within the data center) private ASN assigned to all SSWs in it. In Figure 2, in the first spine plane, all SSWs are numbered ASN 65001. Similarly, all SSWs in the next spine plane would be numbered ASN 65002. This simplicity is possible because each SSW device operates independently from the others, serving as a member of the ECMP groups for the paths between pods. As no two SSWs directly peer with each other, they can use the same AS number. Reuse of ASNs acts as a *loop breaking mechanism*, ensuring that no route will traverse through multiple SSWs. The unique per-plane ASNs also aid us in simple identification of the operationally available planes for paths visible on rack switches.

2.5 Route Summarization

There are two principal categories of IP routes in our data centers: infrastructure and production. Infrastructure prefixes facilitate network device connectivity, management, and diagnostics. They carry relatively low traffic. In the event of a device or link failure, their reachability may be non-critical or can be supported by stretched paths. Production prefixes carry high-volume live traffic of our applications and must have continuous reachability in all partial failure scenarios, with optimal routing and sufficient capacity of all involved network paths and ECMP groups.

There are many routes in our data centers. To minimize the FIB size requirements in hardware and ensure lightweight control plane processing, we use hierarchical route summarization on all levels of the network topology. For production routes, we design IP addressing schemes which closely reflect the multi-level hierarchy. The RSWs aggregate the IP addresses of their servers and the FSWs aggregate the routes of their RSWs. For infrastructure routes, we have the following aggregates. Each device aggregates the IP addresses of all its interfaces, i.e. per-device aggregate. FSWs aggregate per-device RSW/FSW infrastructure routes into per-pod aggregates. And SSWs aggregate per-device SSW infrastructure routes into per-spine aggregates.

Depending on the route type and associated reachability criteria, switches advertise prefixes into BGP either unconditionally, or upon meeting the requirement of the minimal number of more-specific prefixes. The more-specific prefixes have a more limited propagation scope, while the coarser aggregates propagate farther on the network. For example, rack prefixes circulate only within their local pod, while pod-level aggregates propagate to the other pods and racks within the local data center fabric.

Hence, despite the sheer scale of our data center fabrics, our structured uniform route summarization ensures that the sizes of routing tables on switches are in low thousands of routes. Without route summarization, each router would have over hundred thousand routes, each route corresponding to the switches' interfaces and server racks. Our approach has many benefits: it allows us to use inexpensive commodity switch ASICs at the data center scale, enables fast and efficient transmission of routing updates, speeds up convergence (§5), and speeds up programming forwarding hardware.

3 Routing Policies

A key feature of BGP is the availability of well-defined attributes that influence the best path selection. Together with the ability to intercept route advertisements and admission at any hop and session, it allows us to control route propagation in the network with high precision. In this section, we review the use cases for routing policies in our data centers. We also describe how we configure the policies in BGP, while

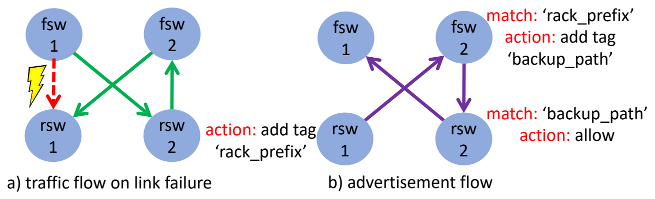


Figure 3: Example of predefined backup path policy.

realizing our principles of uniformity and simplicity.

3.1 Policy Goals

The Internet comprises multiple ASes owned by different ISPs. ISPs coordinate with each other to ensure routing objectives across the Internet. The routing policies mainly pertain to peering based on business relationships (customer-peer-provider) among different ISPs. However, since all the routers in our data centers are controlled by us, we do not have to worry about peering based on business relationships. Our data center routing design uses routing policies to ensure reliability, maintainability, scalability, and service reachability. We summarize these policy goals in Table 1.

Goal	Description
Reliability	Enforce route propagation scopes, predefine backup paths for failure
Maintainability	Isolate and remediate problematic nodes without disrupting traffic
Scalability	Enforce route summarization, avoid backup path explosion
Service reachability	Avoid service disruptions when instances of services are added, removed or migrated

Table 1: Policy goals

We use BGP Communities/tags to categorize prefixes into different types. We attach a particular *route type* community during prefix origination at the network device. This type *tag* persists with the prefix as it propagates. We perform matching on these communities to implement all our BGP policies in a uniform scalable fashion. We demonstrate how we use them with the examples in this section.

Reliability. Our routing policies allow us to safeguard the data center network stability. The BGP speakers only accept or advertise the routes they are supposed to exchange with their peers according to our overall data center routing design.

The BGP policies match on tags to enforce the intended route propagation scope. For example, in Fig. 3b, routes tagged with $rack\_prefix$ only propagate within the pod (i.e., not to the SSW layer).

Using BGP policies, we establish deterministic backup paths for different route types. This uniformly-applied procedure ensures the traffic will take predictable backup paths in the event of failures. We use backup path policies to protect FSW-RSW link failures. Consider the example in Fig. 3. We use tags to implement the backup policy, as shown in Fig. 3b. When $rsw1$ originates a route, it adds a $rack\_prefix$ tag. The $fsw2$ matches on that tag, adds another tag $backup\_path$, and forwards the route to $rsw2$. $rsw2$ ensures routes tagged with $backup\_path$ are advertised to $fsw1$. When $fsw1$ detects the tag $backup\_path$, it installs the backup route and adds the tag $completed\_backup\_path$ (not shown in figure) which stops any unnecessary continued backup route propagation. In Fig. 3a, when the fsw1-rsw1 link fails, fsw1 will not send a new advertisement to its SSWs to signal the loss of connectivity to rsw1. Instead, BGP will reconverge to use the backup path ($fsw1 \rightarrow rsw2 \rightarrow fsw2 \rightarrow rsw1$) to reroute traffic through another RSW within the pod. And due to route summarization at the FSW (§2.5), these failures within a pod will not be visible to the SSWs and hence the routers outside the pod.

Backup paths are computed and distributed automatically as a part of BGP routing convergence. They are readily available when link failure happens. Typically, an FSW has multiple backup paths, of the same AS path length, to each RSW. When the direct fsw-rsw link fails, all of the backup paths will be used for ECMP.

In our network, each device has inbound (import) and outbound (export) match-action rules. Routes get advertised between two neighboring BGP speakers (X and Y) if they are allowed at both ends of the BGP session, i.e., they need to match an outbound rule of device X and an inbound rule of its neighboring device Y. This logic protects against routing misconfigurations on the peer. Additionally, on each device, routes that do not match on any of its rules are dropped to prevent unintended network behaviors.

Maintainability. In a data center, many events occur every hour and we expect things to fail. We see events like rack removal or addition, link flap or transceiver failure, network device reboot or software crash, software or configuration push failure, etc. Additionally, network devices are undergoing routine software upgrades and other maintenance operations. To avoid disruption of production traffic, we gracefully *drain* the device before maintenance—production traffic gets diverted from the device without incurring losses. For this, we define multiple distinct operational states for a network device. The state affects the route propagation logic through the device, as shown in Table 2. We change the routing policy configuration of a device based on its operational state. These configurations implement the logic specified in Table 2.

To gracefully take a device or a group of devices out of service (DRAINED) or put it back in service (LIVE), we apply policies corresponding to the current state on the peer groups. This initiates the new mode of operation across all affected BGP peers. Previous works [23] have used a multi-stage draining to gracefully drain traffic without disruptions. We also implement a multi-stage drain process with an interim WARM state. In the WARM state, we change the BGP policies to de-prioritize routes traversing through the device about to be drained. We also adjust the local and/or remote

State	Description
LIVE	The device is operating in active mode and carries full production traffic load.
DRAINED	The device is operating in passive mode. It doesn't carry any production traffic. Only the traffic to/from infrastructure/diagnostic prefixes may be allowed. Transiting infrastructure prefixes are lowered in priority.
WARM	The device is in process of changing states. It maintains full local RIB and FIB ready to support the production traffic, but adjusts route propagation and signals to avoid attracting live traffic.

Table 2: Operational states of a network switch

ECMP groups and ensure that network links do not become overloaded during the transition from LIVE to DRAINED state and vice-versa. Once BGP converges, all production traffic is rerouted to/from the device, and we can change the state of the network device again into the final state.

In the DRAINED state, BGP policies allow us to propagate only selected prefixes through the devices, and change route priorities. For example, this feature allows us to maintain reachability to the infrastructure (e.g., the switch's management plane) and advertise diagnostic prefixes through the devices under maintenance, while keeping the production traffic away from such devices.

Drain/undrain is a frequently used operation in data center maintenance. On average, we perform 242 drain and undrain operations daily. These operations take on average 36*s* to complete. The multi-stage state change ensures that there are no transient drops during this process.

Scalability. The routing policies allow us to implement and enforce our hierarchical route summarization design (§2.5). For example, in our network, our policy in FSW summarizes rack-level prefixes into pod-specific aggregates. They advertise these aggregates to the SSW tier. These policies also control propagation scopes for different route aggregation levels and minimize the routing table sizes in our switches.

The predefined backup paths also aid in scalability. These paths ensure our reaction to failures are deterministic and avoid triggering large-scale advertisements during failures which can cause BGP convergence problems.

To reduce policy processing overhead, we design all our policies to first apply rules which accept or deny the most number of prefixes. For example, in a drained state (Table 2), the FSW's outbound policy toward SSW first rejects routes marked to (i) avoid propagation to SSWs, or (ii) carry any production traffic. After that, it matches and lowers the priority of infrastructure routes before sending them to SSW. This design ensures we minimize the policy processing overhead on routes that will be dropped.

Service Reachability. One important goal of the data center network design is providing service reachability. A service should remain reachable even when an instance of the service gets added, removed, or migrated. As one of the mechanisms for providing service reachability in the network, we use Virtual IP addresses (VIPs). A particular service (e.g., DNS) may advertise a single VIP (serviced by multiple instances). In turn, anycast routing will provide reachability to one of the instances for traffic destined to the VIP.

To support flexible instance placement without compromising uniformity and simplicity, we create a VIP injector service in the form of a software library integrated with a service instance. The injector establishes a BGP session with the RSW and announces a route to signal the VIP reachability. When the service instance gets terminated, the injector sends a route withdrawal for the VIP. The routing policy on the RSW relays VIP routes to FSW after performing safety checks, such as ensuring that the injected VIP prefix conforms to the design intent. FSW's inbound policy from RSWs tags and sets different priorities for different VIP routes. This method allows for network-wide VIP priorities for active/backup applications.

By directly injecting VIP routes from services, we do not need to make changes to the network when creating/destroying service instances or adjusting active/backup service behaviors. That is, we do not need to change RSW configurations to start/stop advertising the VIPs or change VIP instance priorities. Our services integrate the injector library into their code (§5) and fully control when and how they want to update their VIPs.

3.2 Policy Configuration

For scalability and uniformity reasons, our policies primarily operate on BGP Communities and AS_PATH regular expression matches, and not on specific IP prefixes. To implement policy actions, we may accept or deny a route advertisement, or modify BGP attributes to manipulate the route's priority for best-path selection. We configure our routing policies on the BGP *peer group* level—therefore, any policy change is simultaneously applied to all peers in the group. Our reusable ASN design (§2.4) also allows us to use the same policies across our multiple data center fabrics.

The number of policy rules that exist between tiers of our data center network are relatively lightweight: 3-31 inbound rules (average 11 per session) and 1-23 outbound rules (average 12 per session). The majority of outbound policies tag routes to specify their propagation scope, and the majority of inbound policies perform admission control based on the route propagation tags and adjust LOCAL_PREF or AS_PATH length to influence route preference.

For the most numerous device role in our fleet, RSW, we keep the policy logic at the necessary minimum to reduce the need for periodic changes. To compensate for this, the FSWs in the pods have larger policies that offload some processing logic from the RSWs.

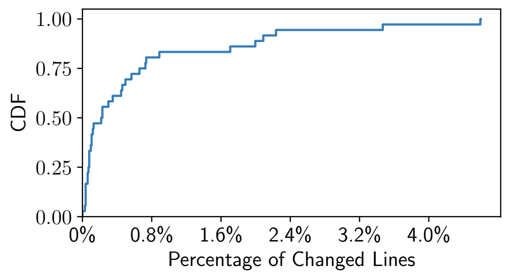


Figure 4: Network Policy Churn

For the commonly used BGP communities and other prefix attributes we maintain structured naming and numbering standards, suitable both for humans and automation tools. For the purposes of this paper, we elide the low-level details of our policy language syntax, objects, and rules.

3.3 Policy Churn

We maintain a global set of abstract policy templates and use them to generate individual switch configurations via an automated pipeline [44]. The routing policy used in our data center is fairly stable—we have made 40 commits to the routing policy templates over a period of three years. We show the cumulative distribution function (CDF) of the number of lines of changes made to the routing policy templates in Figure 4. We observe that most changes to the policy are incremental—80% of commits change less than 2% of policy lines. However, small changes to policy can have drastic service impacts, therefore they are always peer-reviewed and tested before production deployment (§6.2).

4 BGP in DCs versus the Internet

Multiple papers have studied issues with BGP convergence [33, 37], routing instabilities [32] and misconfigurations [21, 36], in the context of the Internet. This section summarizes these issues and describes how we address them in the data center context.

4.1 BGP Convergence

BGP convergence at the Internet-scale is a well-studied problem. Empirically, BGP can take minutes to converge. Labovitz et al. [33] proposed an upper bound on BGP convergence. During convergence, BGP attempts to explore all possible paths in a monotonically increasing order (in terms of AS_PATH length)—a behavior known as the path-hunting problem [2]. In the worst case, BGP convergence can require O(n!) messages, where n is the number of routers in the network. Using *MinRouteAdvertisementInterval* (MRAI) timer—minimum time between advertisements from a BGP peer for a particular prefix—BGP convergence can take O(n) x MRAI seconds. As mentioned in §3.1, our data centers experience many drain/undrain operations daily. These operations will cause BGP to reconverge, and this makes convergence a frequent event in our data centers.

To alleviate the BGP path-hunting problem, we define route propagation scopes and limit the set of backup paths that a BGP process needs to explore. For example, rack prefixes circulate only within a fabric pod; thus, an announcement or withdrawal of a rack prefix should only trigger a pod's reconvergence. To prevent slow convergence during network failures, we employ BGP policies that limit the AS_PATH that a prefix may carry, thus curbing the path-hunting problem.

Our topology design with broad path diversity (§2) and our predefined backup path policies (§3.1) ensure we only trigger fabric-wide re-advertisements when a particular router has lost all connections to its peers. Such events require tens to hundreds of links to fail, which is very unlikely. Thus, BGP convergence delays are infrequent in our data center. Since we want the network to converge as quickly as possible, we set the MRAI timer to 0. This could lead to increased advertisements (as each router would advertise any changes immediately), but our route propagation scopes ensure these advertisements do not affect the entire network.

4.2 Routing Instability

Routing instability is the rapid change of network reachability and topology information caused by pathological BGP updates. These pathological BGP updates lead to increasing CPU and memory utilization on routers, which can result in processing delays for legitimate updates, or router crashes; these can lead to delay in convergence or packet drops. Labovitz et al. [32] show that a significant fraction of routing updates on the Internet was pathological and do not reflect real network changes. With fine-grained control over the routing design, BGP configuration, and software implementation, we ensure that these pathological cases do not manifest in the data center. We describe the common pathological cases of routing instabilities and the solution in our data center to mitigate these cases in Table 3.

The most frequent pathological BGP message pattern reported by Labovitz et al. was WWDup. WWDup is a repeated transmission of BGP withdrawals for a prefix, which is unreachable. The cause of WWDup was stateless BGP implementation: a BGP router does not store any state regarding information advertised to its peers. The router would send a withdrawal to all its peers, irrespective of whether it had sent the same message. Internet-scale routers deal with millions of routes, so it was not practical to store each prefix's state for each peer. In data centers, BGP works at a much smaller scale (tens of thousands of prefixes) and typically has more memory resources. Thus, we can maintain the state of advertisements sent to each peer and check if a particular update needs sending. This feature eliminates pathological BGP withdrawals. Another class of pathological routing messages is AADup: a

Update Type	Description	DC Solution
WWDup	Repeated BGP withdrawals for unreachable prefixes	Store advertisement state in routers to suppress duplicate withdrawals
AADup	Implicit route withdrawal replaced by a duplicate of the same route	Store advertisement state in routers to suppress duplicate announcements
AADiff	Route is replaced by an alternate route	Fixed set of LOCAL_PREF values to avoid pathological metric changes
TUp/TDown	Prefix reachability oscillation	Monitor failures and automatically drain traffic from faulty devices

Table 3: Pathological BGP Updates found in the Internet by Labovitz et al. [32] and how we fix those in the data center

route is implicitly withdrawn and replaced by a duplicate. We stop AADups with our stateful BGP implementation as well.

The other types of BGP messages causing routing instabilities are AADiff (an alternate route replacing the old one) and TUp/TDown (prefix reachability oscillation). AADiffs happen due to MED (multi-exit discriminator) or LOCAL_PREF (local preference) oscillations in configurations that map these values dynamically from the IGP metric. As a result, when internal topology changes, BGP will announce advertisements to its peers with new MED/LOCAL_PREF values, even though the inter-domain BGP paths are unaffected. Hot-potato BGP routing [46] is a similar type of routing instability where the internal IGP cost affects the BGP best path decision. We use a fixed set of LOCAL_PREF values. Thus, any change in LOCAL_PREF indicates a legitimate update in the routing preference. We do not use MED. TUp and TDown come from the actual *oscillating* hardware failures. Our monitoring tools detect such failures and automatically reroute traffic from malfunctioning components to restore stability.

4.3 BGP Misconfigurations

Mahajan et al. [36] analyzed BGP misconfigurations in the Internet. They found that those affected up to 1% of the global prefixes each day. The misconfigurations increase the BGP control plane overhead with generation of pathological route updates. They can also lead to disruption of connectivity. The two types of BGP misconfigurations were the following. First, the origin misconfiguration is when a BGP router injects an incorrect prefix to the global BGP table. Second, the export misconfiguration is when an AS_PATH violates the routing policy for an ISP. The former can happen in the data center. For example, imagine a router advertising more specific /64 prefixes instead of the aggregated /56 prefix. A router could also inject a prefix from a different pod's address space, hijacking the traffic. The latter is also possible in the data center. A router may incorrectly advertise a prefix outside the prefix's intended propagation scope due to a bug in the routing policy. However, in practice, they are rare in our data center, as all our route advertisement configurations are automatically generated and verified. Since we have visibility and control over the data center, we can detect these issues with monitoring/auditing tools and promptly fix them. We further discuss the causes of misconfigurations reported by Mahajan et al. and demonstrate how we can avoid these in our architecture.

Incorrect BGP Attributes. One of the leading causes for incorrect prefix injection is a router advertising prefixes assuming that they will get filtered upstream. For reliability (§3), we add filters on both ends of the BGP session to ensure incorrect prefixes get filtered at either end. Errors can also happen due to wrong BGP communities, address typos, and inaccurate summarization statements. We use a centralized framework [44] to generate the configuration for individual routers from templates. Thus, we can catch errors from a single source, instead of dealing with separate routers.

Interactions with Other Protocols. A typical pattern is to use IGPs such as OSPF for intra-domain routing and configure redistribution to advertise the IGP routes into BGP for inter-domain routing. Configuring redistribution can end up announcing unintended routes. However, that is not a problem with a single-protocol design that we have.

Configuration Update Issues. Mahajan et al also observed cases when upon BGP restart, unexpected prefixes got advertised due to misconfigurations. For instance, in one scenario, configuration changes were not committed to persistent storage, and a router restarted using the old configuration. In our implementation, we ensure BGP does not advertise prefixes until after processing all configuration constructs. Each router has a configuration database, and we use transactions to update it consistently. We can afford slower upgrade mechanisms in the data center due to increased redundancy; routers in the Internet cannot be unavailable for long periods of time.

Thus, our BGP-based routing design tailored for the data center, that realizes the high-level DC-oriented goals of uniformity and simplicity, is able to overcome BGP problems common in the Internet.

5 Software Implementation

Like any other software, our BGP agent needs updates to add new features/optimizations, apply bug fixes, be compatible with other services, etc. Extending a third-party BGP implementation (by network vendors or open source [22, 30]) is not trivial and can add substantial complexity. Additionally, they have long development cycles for upstreaming or releasing their updates, and this affects our pace of innovation. To overcome those challenges, we develop an in-house BGP agent in C++ to run on our FBOSS [18] switches. In this section, we

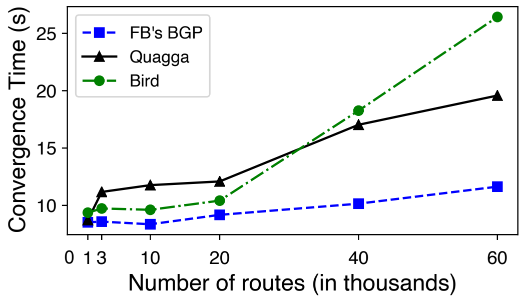


Figure 5: FB's BGP vs Quagga vs Bird (convergence time)

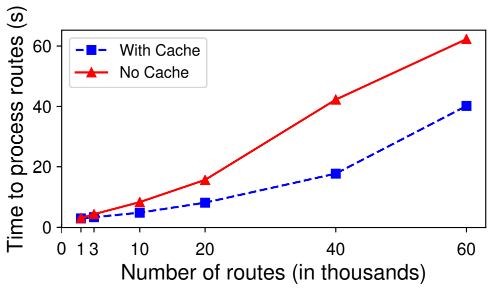


Figure 6: Impact of Policy Cache

present the main attributes of our agent.

Limited Feature Set. There are dozens of RFCs related to BGP features and extensions, especially to support routing for the Internet. Third-party implementations have support for many of these features and extensions. This increases the size of the agent codebase and its complexity due to interactions between various features. A large and complex codebase makes it harder for engineers to debug an issue and find a root cause, extend the codebase to add new features, or to refactor code to improve software quality. Therefore, the implementation of our BGP agent contains only the necessary protocol features required in our data center, but it does not deviate from the BGP RFCs [6–8]. Additionally, we only implement a small subset of matches and actions to implement our routing policies. We summarize the limited protocol features and match-action fields in Appendix A.

Multi-threading. Many BGP implementations are single-threaded (e.g., Quagga [30] and Bird [22]). Modern switches contain server-grade multi-core CPUs which allow us to run the BGP control plane at the scale of our data center. Our implementation employs multiple system threads, such as the peer thread and RIB thread, to leverage the multi-core CPU. The peer thread maintains the BGP state machine for each peer and handles parsing, serializing, sending, and receiving BGP messages over TCP sockets. The RIB thread maintains Loc-RIB (the main routing table), calculates the best path and multipaths for each route, and installs them to the switch hardware. To further maximize parallelism in the context of each system thread, we employ lightweight application threads `folly::fibers` [3]. These have low context-switching cost and execute small modular tasks in a cooperative manner. The fiber design is ideal for the peer thread as BGP session management is I/O intensive. To ensure lock-free property between system threads, we use message queues between fiber threads, running on the same or different systems threads.

To evaluate our BGP agent's performance, we compare it against two popular open source BGP stacks: Quagga [30] and Bird [22]. We run them on a single FSW device that is receiving both IPv4 and IPv6 routes from 24 SSWs. We compare their initial convergence time; this represents the time period between starting the BGP process to network convergence; this includes time for session establishment, and receiving and processing all route advertisements. In Fig. 5, we show the average over 5 runs. We observe that our BGP agent constantly outperforms other software and provides a speedup as high as 1.7X (Quagga) and 2.3X (Bird).

Policy. To improve policy execution performance, we added a few optimizations again building on our uniform design. Most of the peering sessions, from a device's point of view, are either towards uplink or downlink devices sharing the same inbound/outbound policies. Here, we made two observations: (1) prefixes learned from the same peer usually share the same BGP attributes, and (2) when routes are sent to the same type of peers (uplink or downlink peers), the same policy is applied for each peer separately. Peer groups help to avoid repetition in configuration, however, policies are still executed for routes sent/received from each peer separately. To leverage (1), we implemented *batching* in policy execution, where a set of prefixes and their shared BGP attributes are given as input to the policy engine. The policy engine performs the operation of *matching* the given BGP attributes and the prefixes sharing those attributes, and returning the accepted prefixes and their modified BGP attributes, based on the policy *action*. To avoid re-computations of (2), we introduced a policy cache, implemented in the form of an LRU (least recently used) cache containing <policy name, prefix, input BGP attributes, output BGP attributes> tuples. Once we apply the policy for routes to a peer and store that result in the policy cache, other peers in the same tier sharing the same policy can use the cached result and avoid re-execution of the policy. To show its impact, we run an experiment with and without the cache. We run them on a single FSW device that is sending IPv6 routes to 24 SSWs. We compare their time to process all route advertisements, which includes the time to apply outbound policy for each peer. In Fig. 6, we show the average over 5 runs. We observe that policy cache improves the time to process all routes by 1.2-2.4X.

Service Reachability. For flexible service reachability (§3), we want a service to inject routes for virtual IP addresses (VIPs) corresponding to the service directly to the RSW. However, current vendor BGP implementations commonly do not allow multiple peering sessions from the same peer address, which meant we would have to run a single injector service on every server and the applications on the server will need to interact with the injector to inject routes to the RSW. This becomes operationally difficult since application owners do not have visibility to the injection process. There also exists a failure dependency as (i) applications need to monitor the health of the injector service to use it, and (ii) the injector needs to withdraw routes if the application fails. Instead, our BGP agent can support multiple sessions from the same peer address. Applications running on a server can directly initiate a BGP peer session with the BGP agent on the RSW and inject VIPs for service reachability. Thus, we do not have to maintain the cumbersome injector service to workaround the vendor BGP implementation constraint, and we also remove the application-injector dependency.

Instrumentation. Traditionally, operators used network management tools (e.g. SNMP [27], NETCONF [20], etc) to collect network statistics, like link load and packet loss ratio, to monitor the health of the network. These tools can also collect routing tables and a limited set of BGP peer events. However, extending these tools to collect new types of data—such as BGP convergence time, the number of application peers, etc—is not trivial. It requires modifications and standardization of the network management protocols. Facebook uses an in-house monitoring system called ODS [9,18]. Using a Thrift [1] management interface, operators can customize the type of statistics they want to monitor. Next, ODS collects these statistics into an event store. Finally, operators both manually and through an automated alerting tool, query and analyze the data to monitor their system. By integrating our BGP agent with this monitoring framework, we treat BGP like any other software. This allows us to collect fine-granular information on BGP's internal operation state, e.g. the number of peers established, the number of sent/received prefixes per peer, and other BGP statistics mentioned above. We monitor these data to detect and troubleshoot network outages (§6.3).

6 Testing and Deployment

The two main components we routinely test and update are configurations and the BGP agent implementation. These updates introduce new BGP features and optimizations, fix security issues, change BGP routing policies for improving reliability and efficiency. However, frequent updates to the control plane lead to increased risk of network outages in production due to new bugs or performance regressions. We want to ensure smooth network operations, avoid outages in the data center, and catch regressions as early as possible. Therefore, we developed continuous testing and deployment pipelines for quick and frequent rollouts to production.

6.1 Testing

Our testing pipeline comprises three major components - unit testing, emulation and canary testing.

Emulation is a useful testing framework for production networks. Similar to CrystalNet [35], we develop a BGP emulation framework for testing BGP agent, BGP configurations, and policy implementations, and modeling BGP behavior for the entire network. Emulation is used also for testing BGP behavior under failure scenarios – link flaps, link down, or BGP restart events. We also use emulation to test agent/config upgrade processes. The advantage of catching bugs in emulation is that they do not cause service disruptions in production. Emulation testing can greatly reduce developer's time and amount of physical testbed resources required. However, emulation cannot achieve high fidelity as it does not model the underlying switch software and hardware. Using emulation for BGP convergence regression is challenging as linux containers are considerably slower than hardware switches.

After successful emulation testing, we proceed to canary testing in production. We run a new version of the BGP agent/config on a small fraction of production switches called canaries. Canary testing allows us to run a new version of the agent/config in production settings to catch errors and gain confidence in the version before rolling out to production. We pick switches such that canaries can catch issues arising in production due to scale – e.g., delayed switch convergence. Canaries are used to test the following scenarios: (i) transitioning from old to new BGP agent/config (this occurs during deployment), (ii) transitioning from new to old BGP agent/config (when issues were found in production, we have to rollback to stable BGP version), and (iii) BGP graceful restart (which is an important feature for smooth deployment of BGP agent/config). Daily canaries are used to run new versions for longer periods (typically a day). Production monitoring systems will generate alerts for any abnormal behaviors. Canary testing helps us catch bugs not caught in emulation as it closely resembles BGP behavior in production, such as problems created by changes in underlying libraries.

6.2 Deployment

Once a change (agent/config) has been certified by our testing pipeline, we initiate the deployment phase of pushing the new agent/config to the switches. There is a trade-off between achieving high release velocity and maintaining overall reliability. We cannot simply switch off traffic across the data centers and upgrade the control plane in one-shot, as that would drastically impact services and our reliability requirements. Thus, we must ensure minimal network disruption while deploying the upgrades. This is to support quick and frequent BGP evolution in production. We devise a push plan which starts rolling out the upgrade gradually to ensure we can catch problems earlier in the deployment process.

Push Mechanisms. We classify upgrades in two classes: disruptive and non-disruptive, depending on if the upgrade affects existing forwarding state on the switch. Most upgrades in the data center are non-disruptive (performance optimizations, integration with other systems, etc.). To minimize routing instabilities during non-disruptive upgrades, we use BGP *graceful restart (GR)* [8]. When a switch is being upgraded, GR ensures that its peers do not delete existing routes for a

Phase	Specification
P1	Small number of RSWs in a random DC
P2	Small number of RSWs (> P1) in another random DC
P3	Small fraction of switches in all tiers in DC serving web traffic
P4	10% of switches across DCs (to account for site differences)
P5	20% of switches across DCs
P6	Global push to all switches

Table 4: Specification of the push phases

period of time during which the switch's BGP agent/config is upgraded. The switch then comes up, re-establishes the sessions with its peers and re-advertises routes. Since the upgrade is non-disruptive, the peers' forwarding state are unchanged. Without GR, the peers would think the switch is down, and withdraw routes through that switch, only to re-advertise them when the switch comes back up after the upgrade.

Disruptive upgrades (e.g., changes in policy affecting existing switch forwarding state) would trigger new advertisements/withdrawals to switches, and BGP re-convergence would occur subsequently. During this period, production traffic could be dropped or take longer paths causing increased latencies. Thus, if the binary or configuration change is disruptive, we drain (§3) and upgrade the device without impacting production traffic. Draining a device entails moving production traffic away from the device and reducing effective capacity in the network. Thus, we pool disruptive changes and upgrade the drained device at once instead of draining the device for each individual upgrade.

Push Phases. Our push plan comprises six phases P1-P6 performed sequentially to apply the upgrades to agent/config in production gradually. We describe the specification of the 6 phases in Table 4. In each phase, the push engine *randomly* selects a certain number of switches based on the phase's specification. After selection, the push engine upgrades these switches and restarts BGP on these switches. Our 6 push phases are to progressively increase scope of deployment with the last phase being the global push to all switches. P1-P5 can be construed as extensive testing phases: P1 and P2 modify a small number of rack switches to start the push. P3 is our first major deployment phase to all tiers in the topology. We choose a single data center which serves web traffic because our web applications have provisions such as load balancing to mitigate failures. Thus, failures in P3 have less impact to our services. To assess if our upgrade is safe in more diverse settings, P4 and P5 upgrade a significant fraction of our switches across different data center regions which serve different kinds of traffic workloads. Even if catastrophic outages occur during P4 or P5, we would still be able to achieve high-performance connectivity due to the in-built redundancy in the network topology and our backup path policies—switches running the stable BGP agent/config would re-converge quickly to reduce impact of the outage. Finally, in P6, we upgrade the rest of the switches in all data centers.

Push Monitoring. To detect problems during deployment, we have BGPMonitor, a scalable service to monitor all BGP speaking devices in the data center. All BGP speakers relay advertisements/withdrawals they receive to BGPMonitor. BGPMonitor then verifies the routes which are expected to be unchanged, e.g., routes for addresses originating from the switch. If we see route advertisements/withdrawals within the window of a non-disruptive upgrade, we stop the push and report the potential issue to an engineer, who analyzes the issue and determines if push can proceed. One of our outages was detected using BGPMonitor (§6.3).

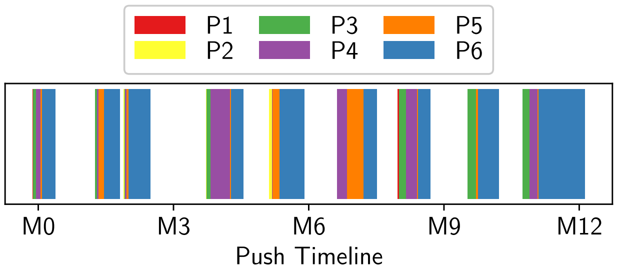


Figure 7: Timeline of BGP push phases over a year

Release	Total	P1	P2	P3	P4	P5	P6
7	0.57	0	0	0.28	0.20	0.82	0.56
8	0.43	0	0	0	0.12	0.13	0.54
9	0.51	0	0.94	0.95	1.12	0.25	0.49

Table 5: Push error percentages for the last 3 pushes for different push phases.

Push Results. Figure 7 shows the timeline of push releases over a 12 month period. We achieved 9 successful pushes of our BGP agent to production. On average, each push takes 2-3 weeks. Figure 7 highlights the high release velocity that we are able to achieve for BGP in our data center. We are able to fix performance and security issues as well as support new features at fast timescales. This also allows other applications, which leverage the BGP routing features, to innovate quickly. P6 is the most time-consuming phase of the push as it upgrades majority of the switches. We catch various errors in P1-P5, and thus, some of these phases can take longer (more than a day). Figure 7 also highlights the highly evolving nature of the data center. Our data centers are undergoing different changes to the BGP agent (adding support for BGP constructs, bug fixes, performance optimizations and security patches) for over 52% of the time in the 12 month duration.

Ideally, each phase should upgrade all the switches (100%). For instance, in one push, we fixed a security bug and we needed all the switches to run the fixed BGP agent version to ensure the network is not vulnerable. However, various devices were not reachable for a multitude of reasons. Devices are often brought down for various maintenance tasks, thus making them unreachable during push. Devices can also be experiencing hardware or power issues during the push phases. We cannot predict the downtime for such devices, and we do not want to block the push indeterminately because of a small fraction of these devices. Hence, for each phase, we set a threshold of 99% on the number of devices we want to upgrade in each phase, i.e., 1% of the devices in our data centers could be running older BGP versions. We expect these devices will be upgraded in the next push phases. We report the push errors (number of devices which did not get upgraded) encountered in the last 3 pushes of Figure 7 in Table 5. We upgrade more than 99.43% of our data center in each push. These numbers indicate that there is always a small fraction of the data center which is undergoing maintenance. We try to upgrade these devices in the next push.

6.3 SEVs

Despite our testing and push pipeline, the scale and evolving nature of our data center's control plane (§6.2), the complexity of BGP and its interaction with other services (e.g. push, draining, etc), and the inevitable nature of human errors make network outages an unavoidable obstacle. In this section, we discuss some of the major routing-related Site EVents (SEVs) that occurred over a 2 year period. Errors and routing issues can arise due to (1) a recent change in configuration or BGP software, or (2) latent bugs in the code which are triggered due to a previously unseen scenario. We use multiple monitoring tools to detect anomalies in our network. These include (i) event data stores (ODS [9]) to log BGP statistics like downtime of BGP sessions at a switch, (ii) netsonar [34] to detect unreachable devices, and (iii) netnorad [10] to measure server-to-server packet loss ratio and network latency.

We experienced a total of 14 SEVs. These BGP-related SEVs were caused due to a combination of errors in its policy, software and interaction with other tools (e.g. push framework, draining framework, etc) in our data centers.

One set of SEVs were caused due to incomplete/incorrect deployment of policies. For example, one of the updates required both changing communities set in a policy at one tier and changing policies that act on those communities at another tier. It also required the first to be applied after the latter. However, during a push, policies were applied in an incorrect order. This created blackholes within the data center, degrading performance of multiple services.

Another set of SEVs were caused due to an error in BGP software. One SEV was caused by a bug in implementation of a feature called max-route limit that limits the number of prefixes received from a peer. The bug was that the max-route counter was getting incremented incorrectly for previously announced prefixes. This made BGP tear down multiple sessions, leading services to experience SLA violations.

We also experienced problems due to interactions between different versions of the BGP software. In one SEV, different versions were using different graceful restart parameters [8]. During graceful restart, the old version of BGP used stale paths for $30s$. However, the new version deferred sending new routes for as long as $120s$, waiting for receiving End-of-RIB from all peers. Hence, the old version purged stale paths learned from its peer before receiving them from the new version. This resulted in temporary traffic loss for $\sim 90s$. BGPMonitor detected this outage during the push phases.

All these outages were resolved by rolling back to a previous stable version of BGP, followed by pushing a new fixed version in the next release cycle. Our design principles of uniformity and simplicity, while helpful, do not address issues such as software bugs and version incompatibilities, for which special care is needed. Our aim is to create a good testing framework to prevent these outages. We created the emulation platform during the later phases of our BGP development process and evolved ever since. As a follow-up to the aforementioned SEVs, we added new test cases to emulate those scenarios. As part of our ongoing work (§7), we are exploring ideas to further improve our testing pipeline.

7 Future Work

This section describes some of our ongoing work based on the gaps we have identified during our past years of data center network operations.

Policy Management. BGP supports a rich policy framework. The inbound and outbound policy is a decision tree with multiple rules capturing the policy designer's intent. Although the routing policies are uniform across tiers in our design, it is non-trivial to manage and reason about the full distributed policy set. Control plane verification tools [13, 15, 24, 40] verify policies by modeling device configurations. However, existing tools cannot scale to the size of our data centers, and they do not support such complex intent as flexible service reachability. Extending network verification to support our policy design at scale is an important future direction. Network synthesis tools [12, 16, 17, 19, 43] use high-level policy intents to produce policy-compliant configurations. Unfortunately, the policy intent language used by these tools cannot model all our policies (§2). Additionally, the configurations generated by them do not follow our design choices (§3). Extending network synthesis to support our BGP design and policies is also an ongoing direction we are pursuing.

Evolving Testing Framework. Policy verification tools assume the underlying software is error-free and homogeneous across devices. 8 of our SEVs occurred due to software errors. Existing tools cannot proactively detect such issues. To compensate, we use an emulation platform to detect control-plane errors before deployment. Some routing issues, like transient forwarding loops and black holes, materialize while deploying BGP configuration and software updates in a live network. Our deployment process monitoring (§6.2) demonstrates that the control plane is under constant churn. 10 of our SEVs were triggered while deploying changes. To address that, we are extending our emulation platform to mimic the deployment pipeline and validate the impact of various deployment strategies. We are further exploring techniques to closely emulate our hardware switches and combined hardware/software failure scenarios. We are also extending our testing framework to include network protocol validation tools [45] and fuzz testing [31]. Protocol validation tools can ensure our BGP agent is RFC-compliant. Fuzz testing can make our BGP agent robust against invalid, unexpected, or random external BGP messages with well-defined failure handling.

Load-sharing under Failures. Over the past few years, we observe that hardware failures or drains can create load imbalance. For example, SSW's uplinks to the DC aggregation layer are not balanced when the failure of an SSW-FSW link (or SSW/FSW node) creates topology asymmetry in the spine plane. If one of an RSW's (say R) four upstream FSWs (say F) cannot reach one of its four SSWs, then F's SSWs would serve 1/4 of the traffic over 3 uplinks unlike the other 3 FSWs that serve 1/4 of the traffic over 4 uplinks. To balance traffic load across SSW's uplinks, R should reduce the traffic sent towards F from 1/4 to 3/15, and shift the remaining traffic to the other 3 FSWs. Although a centralized controller would be the most direct way to shift traffic to balance the load, we are considering an approach like Weighted ECMP [48] to leverage our BGP-based routing design.

8 Related Work

Routing in Data Center. There are different designs for large-scale data center routing, some are based on BGP while others use a centralized software-defined networking (SDN) design. An alternative BGP-based routing design for data centers is described in RFC7938 [11]. Our design differs in a few significant ways. One difference is the use of BGP Confederations for pods (called "clusters" in RFC7938). That enables our design to stick with the two-octet private ASN numbering space and reuse the same ASN on all rack switches. Thus, we also do not use the "AllowAS In" BGP feature in our design and maintain native BGP loop prevention. The second difference is our extensive use of route summarization in order to keep the routing tables small and improve the stability and convergence speed of the distributed system. The RFC7938 proposes keeping full routing visibility for all prefixes on all rack switches. Another major difference is our extensive use of the routing policies to implement strict adherence to the reachability and reliability goals, realize the different operational states of the devices, establish pre-determined network backup paths, and provide means for host-signaled traffic engineering, such as primary/secondary path selection for VIPs.

Singh et. al [42] showed that Google uses an SDN-based design for its data center network routing. It has a central route controller to collect and distribute link state information over a reliable out-of-band Control Plane Network (CPN) that runs a custom IGP for topology state distribution. Their reasoning behind building a centralized routing plane from scratch was to be able to leverage the unique characteristics and homogeneity of their network which comprises custom hardware. We decided to use a decentralized BGP approach to take advantage of BGP's extensive policy control, scalability, third-party vendor support, operator familiarity, etc.

Operational Framework. CrystalNet [35] is a cloud-scale, high-fidelity network emulator used by Microsoft to proactively validate all network operations before rolling out to production. We use an in-house emulation framework to easily integrate with our monitoring tools and deployment pipelines. Janus [14] is a software and hardware update planner that uses operator specified risks to estimate and choose the push plan with minimal availability and performance impact on customers. We use a framework similar to Janus for our maintenance planning, which includes disruptive BGP agent/config push. Govindan et. al [26] conducted detailed analysis of over 100 high-impact network failure events at Google. They discovered that a large number of failures happened when a network management operation was in progress. Motivated by these failures, they proposed certain design principles for high availability, e.g. continuously monitor the network, use in-house testing and rollout procedures, make (network) update the common case, etc. We acknowledge these principles; they have always been a part of our operational workflow.

BGP at Edge. EdgeFabric [41] and Espresso [47] also run BGP at scale. However, they are deployed at the edge for the purpose of CDN traffic engineering. They are both designed by content providers to overcome challenges with BGP when dealing with large traffic volumes. They have centralized control over routing while retaining BGP as the interface to peers. They control which PoP and/or path traffic to a customer should choose as a function of path performance.

9 Conclusion

This paper presents our experience operating BGP in large-scale data centers. Our design follows the principles of *uniformity* and *simplicity*, and it espouses tight integration between the data center topology, configuration, switch software, and DC-wide operational pipeline. We show how we realize these principles and enable BGP to operate efficiently at scale. Nevertheless, our system is a work in progress. We describe some major operational issues we faced and how these are informing our routing evolution.

Acknowledgments. We thank many Facebook colleagues who have contributed to this work over the years and toward this paper. These include Allwyn Carvalho, Tian Fang, Jason Wilson, Hany Morsy, Mithun Aditya Muruganandam, Pavan Patil, Neil Spring, Srikanth Sundaresan, Sunil Khaunte, Omar Baldonado, and many others. We also thank the anonymous reviewers for their insightful comments. This work is supported by the National Science Foundation grants CNS-1637516 and CNS-1763512.

References

[1] Apache Thrift. http://thrift.apache.org/.

[2] BGP Path Hunting. https://paul.jakma.org/2020/01/21/bgp-path-hunting/.

[3] folly::fibers. https://github.com/facebook/folly/tree/master/folly/fibers.

[4] Introducing data center fabric, the next-generation Facebook data center network. https://engineering.fb.com/production-engineering/introducing-data-center-fabric-the-next-generation-facebook-data-center-network/.

[5] Standard for local and metropolitan area networks: Media access control (mac) bridges. *IEEE Std 802.1D-1990*, pages 1–176, 1991.

[6] A Border Gateway Protocol 4 (BGP-4). https://tools.ietf.org/html/rfc4271, 2006.

[7] Autonomous System Confederations for BGP. https://tools.ietf.org/html/rfc5065, 2007.

[8] Graceful Restart Mechanism for BGP. https://tools.ietf.org/html/rfc4724, 2007.

[9] Facebook's Top Open Data Problems. https://research.fb.com/blog/2014/10/facebook-s-top-open-data-problems/, 2014.

[10] NetNORAD: Troubleshooting networks via end-to-end probing. https://engineering.fb.com/core-data/netnorad-troubleshooting-networks-via-end-to-end-probing/, 2016.

[11] Use of BGP for routing in large-scale data centers. https://tools.ietf.org/html/rfc7938, 2016.

[12] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. Aed: incrementally synthesizing policy-compliant and manageable configurations. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, pages 482–495, 2020.

[13] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. Tiramisu: Fast and general network verification. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[14] Omid Alipourfard, Jiaqi Gao, Jeremie Koenig, Chris Harshaw, Amin Vahdat, and Minlan Yu. Risk based planning of network changes in evolving data centers. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 414–429. ACM, 2019.

[15] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 155–168, 2017.

[16] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 328–341, 2016.

[17] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Network configuration synthesis with abstract topologies. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 437–451, 2017.

[18] Sean Choi, Boris Burkov, Alex Eckert, Tian Fang, Saman Kazemkhani, Rob Sherwood, Ying Zhang, and Hongyi Zeng. Fboss: building switch software at scale. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 342–356. ACM, 2018.

[19] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. Network-wide configuration synthesis. In *International Conference on Computer Aided Verification*, pages 261–281. Springer, 2017.

[20] Rob Enns, Martin Bjorklund, and Juergen Schoenwaelder. Netconf configuration protocol. Technical report, RFC 4741, December, 2006.

[21] Nick Feamster and Hari Balakrishnan. Detecting bgp configuration faults with static analysis. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design and Implementation - Volume 2*, NSDI'05, page 43–56, USA, 2005. USENIX Association.

[22] Ondrej Filip, Libor Forst, Pavel Machek, Martin Mares, and Ondrej Zajicek. The bird internet routing daemon project. *Internet: www. bird. network. cz*, 2011.

[23] P. Francois, O. Bonaventure, B. Decraene, and P. Coste. Avoiding disruptions during maintenance operations on bgp sessions. *IEEE Transactions on Network and Service Management*, 4(3):1–11, 2007.

[24] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. Fast control plane analysis using an abstract representation. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 300–313, 2016.

[25] Les Ginsberg, Stefano Previdi, and Mach Chen. IS-IS Extensions for Advertising Router Information. RFC 7981, October 2016.

[26] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Evolve or die: High-availability design principles drawn from googles network infrastructure. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 58–72. ACM, 2016.

[27] David Harrington, Randy Presuhn, and Bert Wijnen. Rfc3411: An architecture for describing simple network management protocol (snmp) management frameworks, 2002.

[28] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 15–26, New York, NY, USA, 2013. Association for Computing Machinery.

[29] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jonathan Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally deployed software defined wan. In *Proceedings of the ACM SIGCOMM Conference*, Hong Kong, China, 2013.

[30] Paul Jakma and David Lamparter. Introduction to the quagga routing suite. *IEEE Network*, 28(2):42–48, 2014.

[31] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, 2018.

[32] C. Labovitz, G. R. Malan, and F. Jahanian. Origins of internet routing instability. In *IEEE INFOCOM '99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No.99CH36320)*, volume 1, pages 218–226 vol.1, 1999.

[33] Craig Labovitz, Abha Ahuja, Abhijit Bose, and Farnam Jahanian. Delayed internet routing convergence. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '00, page 175–187, New York, NY, USA, 2000. Association for Computing Machinery.

[34] Jose Leitao and David Rothera. Dr NMS or: How facebook learned to stop worrying and love the network. Dublin, May 2015. USENIX Association.

[35] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. Crystalnet: Faithfully emulating large production networks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 599–613. ACM, 2017.

[36] Ratul Mahajan, David Wetherall, and Tom Anderson. Understanding bgp misconfiguration. In *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '02, page 3–16, New York, NY, USA, 2002. Association for Computing Machinery.

[37] Zhuoqing Morley Mao, Ramesh Govindan, George Varghese, and Randy H. Katz. Route flap damping exacerbates internet routing convergence. In *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '02, page 221–233, New York, NY, USA, 2002. Association for Computing Machinery.

[38] Justin Meza, Tianyin Xu, Kaushik Veeraraghavan, and Onur Mutlu. A large scale study of data center network reliability. In *Proceedings of the Internet Measurement Conference 2018*, pages 393–407. ACM, 2018.

[39] John Moy. Ospf version 2. STD 54, RFC Editor, April 1998. http://www.rfc-editor.org/rfc/rfc2328.txt.

[40] Santhosh Prabhu, Kuan-Yen Chou, Ali Kheradmand, P Godfrey, and Matthew Caesar. Plankton: Scalable network configuration verification through model checking. *arXiv preprint arXiv:1911.02128*, 2019.

[41] Brandon Schlinker, Hyojeong Kim, Timothy Cui, Ethan Katz-Bassett, Harsha V Madhyastha, Italo Cunha, James Quinn, Saif Hasan, Petr Lapukhov, and Hongyi Zeng. Engineering egress with edge fabric: Steering oceans of content to the world. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 418–431. ACM, 2017.

[42] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. *ACM SIGCOMM computer communication review*, 45(4):183–197, 2015.

[43] Kausik Subramanian, Loris D'Antoni, and Aditya Akella. Synthesis of fault-tolerant distributed router configurations. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2(1):1–26, 2018.

[44] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky H.Y. Wong, and Hongyi Zeng. Robotron: Top-down network management at facebook scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page

426–439, New York, NY, USA, 2016. Association for Computing Machinery.

[45] Keysight Technologies. Ixanvl^{TM}–automated network validation library.

[46] Renata Teixeira, Aman Shaikh, Tim Griffin, and Jennifer Rexford. Dynamics of hot-potato routing in ip networks. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '04/Performance '04, page 307–319, New York, NY, USA, 2004. Association for Computing Machinery.

[47] Kok-Kiong Yap, Murtaza Motiwala, Jeremy Rahe, Steve Padgett, Matthew Holliman, Gary Baldus, Marcus Hines, Taeeun Kim, Ashok Narayanan, Ankur Jain, et al. Taking the edge off with espresso: Scale, reliability and programmability for global internet peering. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 432–445. ACM, 2017.

[48] Junlan Zhou, Malveeka Tewari, Min Zhu, Abdul Kabbani, Leon Poutievski, Arjun Singh, and Amin Vahdat. Wcmp: Weighted cost multipathing for improved fairness in data centers. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery.

A BGP Agent Features

As mentioned in §5, our BGP agent contains only those necessary protocol features that are required in our data center. We summarize the different agent features in Table 6. Additionally, we only implement a small subset of matches and actions mentioned in Table 7 to implement our routing policies specified in §3.

	Feature	Description	Rationale
Core Feature	eBGP	Establish external BGP session	To Exchange and forward route updates
	Confederations	Divide an AS into multiple sub ASes	To use the same private ASNs within a pod
	eBGP Multipath	Select and program multipath	To implement ECMP-based load-sharing
	IPv4/IPv6 Addresses	Support IPv4/IPv6 route exchange	To enable dual-stack
	Route Origination	Send update for IP prefixes assigned to a switch	
	Route Aggregation	Send update for less-specific IP prefixes aggregating (summarizing) more-specific routes	To minimize number of route updates
	Remove Private AS	Remove Private ASNs within AS-PATH	To reuse private ASNs.
	In/Out-bound Policy	Support BGP policies specified in §2	
	Dynamic Peer	Accept BGP session initiation from a range of peer addresses	To allow VIP injection from any server
Operational Feature	Graceful Restart	Wait for small graceful time period before removing routes	To reduce network churn
	Link Fail Detection	Fast BGP session termination upon link failure	To converge faster
	Propagation Delay	Delay advertisements of new routes	To wait for convergence before receiving traffic
	FIB Acknowledgement	Advertise routes after installation to hardware	To avoid blackholes if peer converges before us
	Max-route-limit	Limit number of prefixes received from a peer	To disallow unexpected volume of updates
	Peer Groups	Define and reuse peer configurations for multiple peers	To make configuration compact

Table 6: Core and operational BGP features

Match Fields	Action Fields
as-path	add/delete/set as-path
community-list	add/delete/set community
origin	set origin
local preference	inc/dec/set local preference
as-path-length	permit
prefix-list	deny

Table 7: Policy match-action fields

Orion: Google's Software-Defined Networking Control Plane

Andrew D. Ferguson, Steve Gribble, Chi-Yao Hong, Charles Killian, Waqar Mohsin
Henrik Muehe, Joon Ong, Leon Poutievski, Arjun Singh, Lorenzo Vicisano, Richard Alimi
Shawn Shuoshuo Chen, Mike Conley, Subhasree Mandal, Karthik Nagaraj, Kondapa Naidu Bollineni
Amr Sabaa, Shidong Zhang, Min Zhu, Amin Vahdat
Google
orion-nsdi2021@google.com

Abstract

We present Orion, a distributed Software-Defined Networking platform deployed globally in Google's datacenter (Jupiter) and Wide Area (B4) networks. Orion was designed around a modular, micro-service architecture with a central publish-subscribe database to enable a distributed, yet tightly-coupled, software-defined network control system. Orion enables intent-based management and control, is highly scalable and amenable to global control hierarchies.

Over the years, Orion has matured with continuously improving performance in convergence (up to 40x faster), throughput (handling up to 1.16 million network updates per second), system scalability (supporting 16x larger networks), and data plane availability (50x, 100x reduction in unavailable time in Jupiter and B4, respectively) while maintaining high development velocity with bi-weekly release cadence. Today, Orion enables Google's Software-Defined Networks, defending against failure modes that are both generic to large scale production networks as well as unique to SDN systems.

1 Introduction

The last decade has seen tremendous activity in Software-Defined Networking (SDN) motivated by delivering new network capabilities, fundamentally improving network reliability, and increasing the velocity of network evolution. SDN starts with a simple, but far-reaching shift in approach: moving network control, such as routing and configuration management, from individual hardware forwarding elements to a central pool of servers that collectively manage both real-time and static network state. This move to a logically centralized view of network state enables a profound transition from defining pairwise protocols with emergent behavior to distributed algorithms with guarantees on liveness, safety, scale, and performance.

For example, SDN's global view of network state presents an opportunity for more robust network verification and intent-based networking [16, 17]. At a high level, SDN affords the opportunity to transition the network from one consistent state to another, where consistency can be defined as policy compliant and blackhole-free. This same global view and real-time control enables traffic engineering responsive to topology, maintenance events, failures, and even fine-grained communication patterns such that the network as a whole can operate more efficiently and reliably [2, 12, 13]. There is ongoing work to tie end host and fabric networking together to ensure individual flows, RPCs, and Coflows meet higher-level application requirements [1, 11, 22], a capability that would be hard or impossible with traditional protocols. Perhaps one of the largest long-term benefits of SDN is support for software engineering and qualification practices to enable safe weekly software upgrades and incremental feature delivery, which can hasten network evolution by an order of magnitude.

While the promise of SDN is immense, realizing this promise requires a production-grade control plane that meets or exceeds existing network performance and availability levels. Further, the SDN must seamlessly inter-operate with peer legacy networks as no network, SDN or otherwise, operates solely in its own tech island.

In this paper, we describe the design and implementation of Orion, Google's SDN control plane. Orion is our second generation control plane and is responsible for the configuration, management, and real-time control of all of our data center (Jupiter [28]), campus, and private Wide Area (B4 [15]) networks. Orion has been in production for more than four years. The SDN transition from protocols to algorithms, together with a micro-services based controller architecture, enables bi-weekly software releases that together have not only delivered over 30 new significant capabilities, but also have improved scale by a factor of 16, availability by a factor of 50x in Jupiter and 100x in B4, and network convergence time by a factor of 40. Such rapid evolution would have been hard or impossible without SDN-based software velocity

Orion's design centers around a constellation of independent micro-services, from routing to configuration management to network management, that coordinate all state through an extensible Network Information Base (NIB). The NIB sequences and replicates updates through a key-value abstraction. We describe the performance, semantic, and availability requirements of the NIB and the development model that allows dozens of engineers to independently and simultaneously develop, test, and deploy their services through well-defined, simple, but long-lived contractual APIs.

While Orion has been a success at Google, neither our design nor the SDN approach are panaceas. We describe four key challenges we faced in Orion—some fundamental to SDN and some resulting from our own design choices—along with our approach to addressing them:

#1: Logically centralized control require fundamentally high performance for updates, in-memory representation of state, and appropriate consistency levels among loosely-coordinating micro-service SDN applications.

#2: The decoupling of control from hardware elements breaks fate sharing in ways that make corner-case failure handling more complex. In particular, control software failure does not always mean the corresponding hardware element has failed. Consider the case where the control software runs in a separate physical failure domain connected through an independent out-of-band control network. Either the physical infrastructure (control servers, their power or cooling) or control network failure can now result in at least the perception of a sudden, massively correlated failure in the data plane.

#3: Managing the tension between centralization and fault isolation must be balanced carefully. At an extreme, one could imagine a single logical controller for all of Google's network infrastructure. At another extreme, one could consider a single controller for every physical switch in our network. While both extremes can be discarded relatively easily, finding the appropriate middle ground is important. On the one hand, centralization is simpler to reason about, implement, and optimize. On the other, a centralized design is harder to scale up vertically and exposes a larger failure domain.

#4: In a global network setting, we must integrate existing routing protocols, primarily BGP, into Orion to allow inter-operation with non-SDN peer networks. The semantics of these protocols, including streaming updates and fate sharing between control and data plane, are a poor match to our choice of SDN semantics requiring adaptation at a number of levels.

This paper presents an introductory survey of Orion. We outline how we manage these concerns in its architecture, implementation, and evolution. We also discuss our production experiences with running Orion, pointing to a number of still open questions in SDN's evolution. We will share more details and experiences in subsequent work.

2 Related Work

Orion was designed with lessons learned from Onix [18]. Unlike Onix's monolithic design with cooperative multi-threading, Orion introduced a distributed design with each application in a separate process. While Onix introduced a NIB accessible only to applications in the same process, Orion's is accessible by applications within and across domains, providing a mechanism for hierarchy, which few existing controllers incorporate (Kandoo [35] being an exception). Hierarchy enabled fabric-level drain sequencing [1] and optimal WCMP-based (Weighted Cost Multi-Pathing) routing [36].

We distribute Orion's logic over multiple processes for scalability and fault-tolerance, a feature shared with other production-oriented controllers such as ONOS [4] and OpenDaylight [24], and originally proposed by Hyperflow [30]. Unlike our previous design, Orion uses a single configuration for all processes, applied atomically via the NIB, precluding errors due to inconsistent intended state.

Orion uses database-like tables to centrally organize state produced and consumed by SDN programs, a feature shared with a few other OpenFlow controllers such as ONOS [4], Flowlog [27], and Ravel [32]. The combination of all of these techniques – hierarchy, distribution, and database-like abstractions – allowed Orion to meet Google's availability and performance requirements in the datacenter and WAN.

While Orion is an evolution in the development of OpenFlow controllers, its modular decomposition of network functions (e.g., routing, flow programming, switch-level protocols, etc.) is a design goal shared with pre-OpenFlow systems such as 4D/Tesseract [33] and RCP [6]. Single-switch operating systems that similarly employ microservices and a centralized database architecture include Arista EOS [3] and SONiC [25].

3 Design Principles

We next describe principles governing Orion's design. We established many of these during the early stages of building Orion, while we derived others from our experience operating Orion-based networks. We group the principles into three categories: environmental – those that apply to production networks, architectural – those related to SDN, and implementation – those that guide our software design.

3.1 Principles of production networks

Intent-based network management and control. Intent-based networks specify management or design changes by describing the new intended end-state of the network (the "what") rather than prescribing the sequence of modifications to bring the network to that end-state (the "how"). Intent-based networking tends to be robust at scale, since high-level intent is usually stable over time, even when the low-level state of network elements fluctuates rapidly.

For example, consider a situation where we wish to temporarily "drain" (divert traffic away from) a cluster while we simultaneously add new network links to augment the ingress and egress capacity of the cluster. As those new links turn up, the stable drain intent will also apply to them, causing the underlying networking control system to avoid using them.

In Orion, we use an intent-based approach for updating the network design, invoking operational changes, and adding new features to the SDN controller. For example, we capture

[1]Fabric-level drain sequencing refers to redirecting traffic in a loss-free manner, throughout the fabric, away from a target device being drained.

intended changes to the network's topology in a model [26], which in turn triggers our deployment systems and operational staff to make the necessary physical and configuration changes to the network. As we will describe later, Orion propagates this top-level intent into network control applications, such as routing, through configuration and dynamic state changes. Applications react to top-level intent changes by mutating their internal state and by generating intermediate intent, which is in turn consumed by other applications. The overall system state evolves through a hierarchical propagation of intent ultimately resulting in changes to the programmed flow state in network switches.

Align control plane and physical failure domains. One potential challenge with decoupling control software from physical elements is failure domains that are misaligned or too large. For misalignment, consider the case in which a single SDN controller manages network hardware across portions of two buildings. A failure in that controller can cause correlated failures across two buildings, making it harder to meet higher-level service SLOs. Similarly, the failure of a single SDN controller responsible for all network elements in a campus would constitute too large a vulnerability even if it improved efficiency due to a centralized view.

We address these challenges by carefully aligning network control domains with physical, storage, and compute domains. As one simple example, a single failure in network control should not impact more than one physical, storage, or compute domain. To limit the "blast radius" of individual controller failures, we leverage hierarchical, partitioned control with soft state progressing up the hierarchy (§5.1). We explicitly design and test the network to continue correct, though likely degraded, operation in the face of controller failures.

3.2 Principles related to an SDN controller

SDN enables novel approaches to handling failures, but it also introduces new challenges requiring careful design. The SDN controller is remote from the network switches, resulting in the lack of fate sharing but also the possibility of not being able to communicate with the switches.

Lack of fate sharing can often be used to our advantage. For example, the network continues forwarding based on its existing state when the controller fails. Conversely, the controller can repair paths accurately and in a timely manner when individual switches fail, by rerouting around them.

React optimistically to correlated unreachability. The loss of communication between controller and switches poses a difficult design challenge as the controller must deal with incomplete information. We handle incomplete information by first deciding whether we are dealing with a minor failure or a major one, and then reacting pessimistically to the former and optimistically to the latter.

We start by associating a ternary health state with network elements: (i) *healthy* with a recent control communication (a switch reports healthy link and programming state with no packet loss), (ii) *unhealthy*, when a switch declares itself to be unhealthy, when neighbouring switches report unhealthy conditions or indirect signals implicate the switch, and (iii) *unknown*, with no recent control communication with a switch and no indirect signals to implicate the switch.

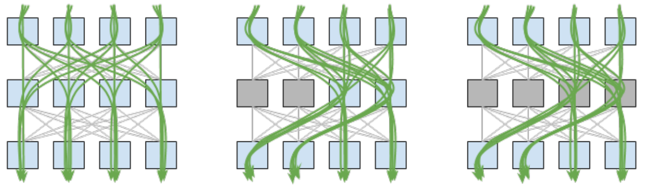


Figure 1: Network behavior in three cases: **Normal** (left): A network with healthy switches. Flows from top to bottom switches use all middle switches. ***Fail Closed*** (mid): With few switches in unknown state (grey), the controller conservatively routes around them. ***Fail Static*** (right): With enough switches in unknown state, the controller no longer routes around newly perceived failed switches.

A switch in the *unknown* state could be malfunctioning, or it could simply be unable to communicate with a controller (a fairly common occurrence at scale). In comparison, the *unhealthy* state is fairly rare, as there are few opportunities to diagnose unequivocal failure conditions in real time.[2]

The controller aggregates individual switch states into a network-wide health state, which it uses to decide between a pessimistic or an optimistic reaction. We call these ***Fail Closed*** and ***Fail Static***, respectively. In *Fail Closed*, the controller re-programs flows to route around a (perceived) failed switch. In *Fail Static*, the controller decides not to react to a switch in an *unknown*, potentially failed, state, keeping traffic flowing toward it until the switch state changes or the network operator intervenes. Figure 1 illustrates an example of normal operation, *Fail Closed* reaction, and *Fail Static* condition.

In *Fail Static*, the controller holds back from reacting to avoid worsening the overall state of the network, both in terms of connectivity and congestion. The trade-off between *Fail Closed* and *Fail Static* is governed by the cost/benefit implication of reacting to the *unknown* state: if the element in the *unknown* state can be avoided without a significant performance cost, the controller conservatively reacts to this state and triggers coordinated actions to steer traffic away from the possible failures. If the reaction would result in a significant loss in capacity or loss in end-to-end connectivity, the controller instead enters Fail Static mode for that switch. In practice we use a simple "capacity degradation threshold" to move from *Fail Closed* to *Fail Static*. The actual threshold value is directly related to: (1) the operating parameters of the network, especially the capacity headroom we typically reserve, for example, to support planned maintenance; (2) the level of redundancy we design in the topology and control

[2]It is not common for a software component to be able to self-diagnose a failure, without being able to avoid it in the first place, or at least repair it. Slightly more common is the ability to observe a failure from an external vantage point, e.g. a neighboring switch detecting a link "going down."

domains. We aim to preserve a certain amount of redundancy even in the face of capacity degradation.

In our experience, occurrences of *Fail Static* are fairly common and almost always appropriate, in the sense that they are not associated with loss of data plane performance. Often *Fail Static* is triggered by failures in the control plane connectivity between the SDN controller and the network elements, or by software failures in the controller. Neither of these directly affect the data plane health.

We view the ability to *Fail Static* as an advantage of SDN systems over traditional distributed-protocol systems. Distributed systems are also subject to some of the failures that could benefit from a *Fail Static* response. However, they are not easily amenable to realize a *Fail Static* behavior because they only have a local view. It is far easier for a centralized controller to assess if it should enter *Fail Static* when it can observe correlated network failures across its entire domain.

Exploit both out-of-band and in-band control plane connectivity. In a software-defined network, a key consideration is connectivity between the "off box" controller and the data plane switches. We must solve the bootstrap problem of requiring a functioning network to establish baseline control. Options include using: i) the very network being controlled ("in-band") or ii) a (physically or logically) separate "out-of-band" network. While a seemingly simple question, considerations regarding pure off-box SDN control, circular dependencies between the control and dataplane network, ease of debuggability, availability, manageability and cost of ownership make this topic surprisingly complex.

The simplest approach is to have a physically separate out-of-band control/management plane network (CPN) for communication between controller and switches orthogonal to the dataplane network. This approach cleanly avoids circular dependencies, keeping the control model simple and enabling easy recovery from bugs and misconfiguration. Ideally, we would like the out-of-band control network to be highly available, easy to manage and maintain, and cost effective. In the end, a separate CPN means installing and operating two distinct networks with different operational models and independent failure characteristics. While failure independence is often a desirable property, subtle or rare failure scenarios mean the entire data plane could go down if *either* the dataplane or control plane fails. We describe our choice of hybrid CPN for Orion in §5.

3.3 Software design principles

Enabling a large-scale software development environment was a key motivation for building our own SDN controller. Critical to the success of SDN is the ability to safely deploy new functionality across the network incrementally with frequent software releases. This, in turn, means that a substantial team of engineers must be able to develop multiple independent features concurrently. The need to scale engineering processes led to a modular system with a large number of decoupled components. At the same time, these components had to interact with one another to realize a tightly-coupled control system reflecting the structure and dynamics of network control. We achieved this goal through:

- a *microservice architecture* with separate processes rather than a monolithic block which we adopted in our first generation SDN controller [18], for software evolvability and fault isolation.
- a *central pub-sub system* (NIB) for all the communication between microservices, which took care of the tightly-coupled interaction across processes.

Failure domain containment (§3.1) imposes an upper limit to the size of control domains. Nevertheless, we were concerned with the performance, scalability, and fault model of a single NIB to coordinate all communication and state within a control domain. We satisfied our performance concerns through benchmarking efforts, and fault tolerance concerns by limiting control domain scope and the ability to fail static, including between control domains.

Based on years of experience, the NIB has been one of our most successful design elements. It manages all inter-component communications, allows us to create a "single arrow of time," establishing an order among the otherwise concurrent events across processes. This brought significantly useful side effects including much improved debuggability of the overall system. It also allows us to store event sequences (NIB traces) in external systems and use them for offline troubleshooting and independent validation of subsystems, which we use in component-level regression testing.

Next, we discuss the principles of intent based control, introduced in 3.1, reconciliation of state as well as the implications of various failure modes in an SDN-based system:

Intent flows from top to bottom. The top level intent for the system as a whole is the operator intent and the static configuration. As intent propagates through the system via NIB messages, it triggers local reactions in subsystems that generate *intermediate intent* consumable by other sub-systems. Higher-level intent is authoritative and any intermediate intent (also known as *derived state*) is rebuilt from it. The programmed switch state is the *ground truth* corresponding to the intent programmed into dataplane devices.

The authoritative intent must always be reflected in the ground truth. The controller ensures that any mismatch is corrected by migrating the ground truth toward the intended state in a way that is minimally disruptive to existing data plane traffic. We refer to this process as "state reconciliation".

Reconciliation is best performed in the controller which has a global view since minimal disruption often requires coordination across switches such that changes are sequenced in a graceful and loop-free manner. Reconciliation is a powerful concept that allows reasoning about complex failure modes such as Orion process restarts as well as lack of fate sharing between the data and control planes.

Availability of high level intent is crucial to keep the top-

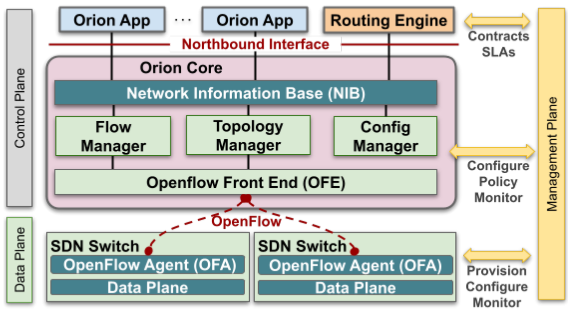


Figure 2: Overview of Orion SDN architecture and core apps.

down intent-based system simple. To achieve this goal we minimize the time when the intent is temporarily unavailable (e.g., because of process restarts or communication failure between components).

4 Architecture

Figure 2 depicts the high-level architecture of Orion, and how it maps to the textbook ONF view [7]. For scalability and fault isolation, we partition the network into domains, where each domain is an instance of an Orion SDN controller.

The data plane consists of SDN switches at the bottom. Orion uses OpenFlow [23] as the *Control to Data Plane Interface (CDPI)*. Each switch runs an OpenFlow Agent (OFA) for programmatic control of forwarding tables, statistics gathering, and event notification. The control plane consists of *Orion Core* in the center and *SDN applications* at the top. The control plane is physically separate from the data plane and *logically* centralized, providing a global view of the domain. Though logically centralized, the Orion Core controls the network through distributed controller processes. The NIB provides a uniform SDN *NorthBound Interface* for these applications to share state and communicate requirements. The Orion Core is responsible for (i) translating these requirements into OpenFlow primitives to reconcile switches' programmed state with intended state and (ii) providing a view of the runtime state of the network (forwarding rules, statistics, data plane events) to applications.

4.1 Orion Core

The **NIB** is the intent store for all Orion applications. It is implemented as a centralized, in-memory datastore with replicas that reconstruct the state from ground-truth on failure. The NIB is coupled with a publish-subscribe mechanism to share state among Orion applications. The same infrastructure is used externally to collect all changes in the NIB to facilitate debugging. The NIB must meet the following requirements:

- *Low External Dependency*. As Orion programs the network supporting all higher-level compute and storage services, it cannot itself depend on higher-level services.
- *Sequential Consistency of Event Ordering*. To simplify coordination among apps, all apps must see events in the same order (*arrow of time* [20]).

Of note, *durability* [14] was not a requirement for the NIB because its state could be reconstructed from network switches and other sources in the event of a catastrophic failure.

NIB Entities. The NIB consists of a set of NIB *entity* tables where each entity describes some information of interest to other applications or observers both local or external to the domain. Some of the entity types include:

- *Configured network topology*. These capture the configured identities and graph relationship between various network topological elements. Examples include `Port`, `Link`, `Interface`, and `Node` tables.
- *Network run-time state*. This could be topological state, forwarding state (e.g. `ProgrammedFlow` table), protocol state (e.g. `LLDPPeerPort` table), statistics (e.g. `PortStatistics` table).
- *Orion App Configuration*. Each app's configuration is captured as one or more NIB tables, e.g. `LLDPConfig`.

Protocol Buffer Schema. We represent the schema for each NIB entity as a protocol buffer message [8]. Each row in that NIB entity table is an instantiation of this schema. The first field of each entity schema is required to be a *NIBHeader* message which serves as the key for that entity. The NIB does not enforce referential integrity for foreign keys; however, inconsistencies fail an internal health-check.

An example entity represented in the NIB is a `Link` entity. A link is modelled as foreign key references to `Port` and `Node` entities respectively. This expresses the connection between two ports of two switches. Additionally, a status (up, down, or unknown), is modelled as part of the Link entity. The full protocol buffer is shown in the appendix.

Protocol buffers allow us to reuse well-understood patterns for schema migrations. For example, adding a new field to a table has built-in support for backward and forward compatibility during an upgrade despite some applications still running with the previous schema.

NIB API. The NIB provides a simple RPC API (*Read, Write, Subscribe*) to operate on NIB tables. The *Write* operation is atomic and supports batching. The *Subscribe* operation supports basic filtering to express entities of interest. The NIB notification model provides sequential consistency of event ordering. It also supports coalescing multiple updates into a single notification for scale and efficiency reasons.

The **Config Manager** provides an external management API to configure all components in an Orion domain. The domain configuration is the set of app configurations running in that domain. For uniformity and ease of sharing, an app config consists of one or more NIB tables. To ensure a new configuration is valid, it is first validated by the running instance. The semantics of pushing config need to be atomic, i.e. if one or more parts of the overall config fail validation, the overall config push must fail without any side effects. Since Orion apps that validate various parts of the config run decoupled, we employ a two-phase commit protocol to update the NIB: The config is first staged in *shadow* NIB tables, and each app verifies its config. Upon success, we commit the shadow tables to live tables atomically.

The **Topology Manager** sets and reports the runtime state of network dataplane topology (node, port, link, interface, etc.). It learns the intended topology from its config in the NIB. By subscribing to events from the switches, it writes the current topology to tables in the NIB. The Topology Manager also periodically queries port statistics from the switches.

The **Flow Manager** performs flow state reconciliation, ensuring forwarding state in switches matches intended state computed by Orion apps and reflected in the NIB. Reconciliation occurs when intent changes or every 30 seconds by comparing switch state. The latter primarily provides Orion with switch statistics and corrects out-of-sync state in the rare case that reconciliation on intent change failed.

The **OFE (Openflow FrontEnd)** multiplexes connections to each switch in an Orion domain. The OpenFlow protocol provides programmatic APIs for (i) capabilities advertisement, (ii) forwarding operations, (iii) packet IO, (iv) telemetry/statistics, and (v) dataplane event notifications (e.g. link down) [23]. These are exposed to the Topology and Flow Manager components via OFE's northbound RPC interface.

Packet-I/O. Orion supports apps that send or receive control messages to/from the data plane through OpenFlow's Packet-I/O API: a ***Packet-Out*** message sends a packet through a given port on the switch, while a ***Packet-In*** notification delivers a data plane packet punted by the switch to the control plane. The notification includes metadata such as the packet's ingress port. Orion apps can program punt flows and specify filters to receive packets of interest.

Orion Core apps are network-type agnostic by design. No "policy" is baked into them; it belongs to higher-level SDN applications instead. Core apps program, and faithfully reflect, the state of the data plane in the NIB in a generic manner.

4.2 Routing Engine

Routing Engine (RE) is Orion's intra-domain routing controller app, providing common routing mechanisms, such as L3 multi-path forwarding, load balancing, encapsulation, etc.

RE provides abstracted topology and reachability information to client routing applications (e.g. an *inter-domain routing* app or a *BGP speaker* app). It models a configured collection of switches within an Orion domain as an abstract routing node called a *supernode* [13] or *middleblock* [28]. Client routing applications provide *route advertisements* at supernode granularity, specifying nexthops for each route in terms of aggregate or singleton external ports.

RE disaggregates the route advertisements from its clients into individual node-level reachability over respective external ports and computes SPF (Shortest Path First) paths for each prefix. RE avoids paths that traverse drained, down or potentially miscabled links.[3] It also reacts to local failure by computing the next available shortest path when the current set of nexthops for a prefix becomes unreachable. For improved capacity, RE performs load balancing within a domain by spreading traffic across multiple viable paths, and through non-shortest-path forwarding, as requested by client apps. RE also manages the associated switch hardware resources (e.g. Layer-3 tables) among its client routing apps.

A key highlight of Orion Routing Engine is the ability to do loss-free sequencing from the currently programmed pathing solution to a new pathing solution. This may happen in reaction to changes in network states (e.g. a link being avoided). In a legacy network, the *eventually consistent* nature of updates from distributed routing protocols (e.g. BGP) can result in transient loops and blackholes in the data plane. In contrast, RE exploits its global view to sequence flow programming: before programming a flow that steers traffic to a set of switches, RE ensures the corresponding prefixes have been programmed on those nexthop switches. Analogous checks are done before removing a flow.

Figure 3 walks through an end-to-end route programming example. As evident from the sequence of operations, the NIB semantics lend themselves to an asynchronous ***intent-based programming model*** (as opposed to a strict *request-response* interaction). A common design pattern is to use a pair of NIB tables, where one expresses the *intent* from the producer, while the other captures the *result* from the consumer. Both *intent* and *result* tables are versioned. An app can change the intent many times without waiting for the result, and the result table is updated asynchronously.

4.3 Orion Application Framework

The Orion Application Framework is the foundation for every Orion application.The framework ensures developers use the same patterns to write applications so knowledge of one SDN application's control-flow translates to all applications. Furthermore, the framework provides basic functionality (e.g. leader-election, NIB-connectivity, health-monitoring) required by all applications in all deployments.

High Availability. Availability is a fundamental feature for networks and thereby SDN controllers. Orion apps run as separate binaries distributed across network control server machines. This ensures applications are isolated from bugs (e.g., memory corruption that leads to a crash) in other applications.

Beyond isolation, replicating each application on three different physical machines ensures fault tolerance for both planned (e.g. maintenance) as well as unplanned (e.g. power failures) outages. The application framework facilitates replication by providing an abstraction on top of leader election as well as life-cycle callbacks into the application.

An application goes through a life-cycle of being activated, receiving intent/state updates from the NIB, and then being deactivated. Identifying/arbitrating leadership and its transition (referred to as *failover*) among replicas is abstracted and

[3] A link is considered miscabled when a port ID learned by a neighbor node via LLDP and reported to Orion does not match the configured port ID.

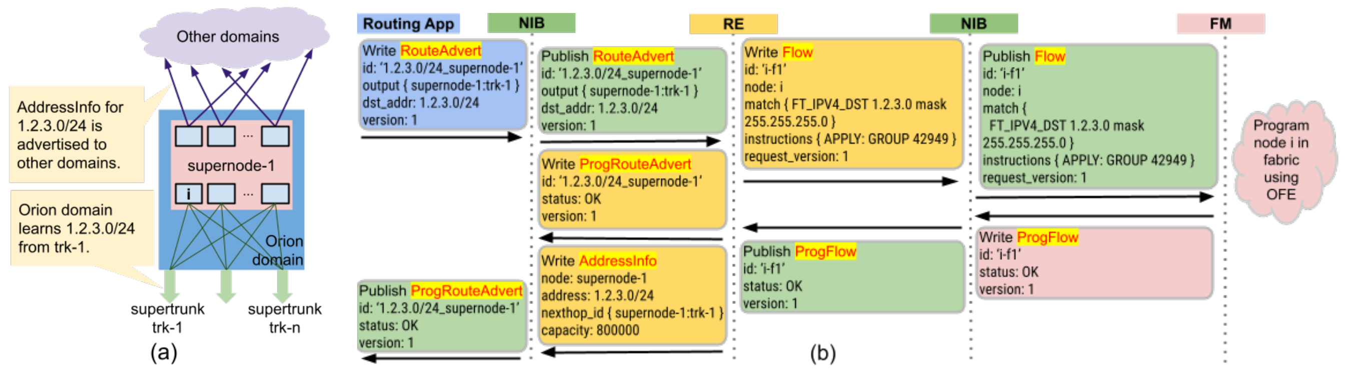


Figure 3: Intent-based route programming on abstracted domain topology: (a) Routing Engine learns external prefix 1.2.3.0/24 over trk-1, and programs nodes to establish reachability. (b) Example of end-to-end route programming. The Routing App provides a high-level RouteAdvert on supernode-1 via the NIB. Routing Engine translates the RouteAdvert to a low-level Flow update on node i and sends to Flow Manager. Acknowledgements follow the reverse direction to the Routing App. Similar route programming applies to all domain nodes.

thereby hidden from the application author, reducing surface area for bugs as well as complexity.

Capability Readiness Protocol. One of the challenges we faced previously was an orderly resumption of operation after controller failover. In particular, when a controller's NIB fails, the state of the new NIB needs to be made consistent with the runtime state of the network, as well as the functional state of all apps and remote controllers. In an extreme case, an Orion requirement is to be able to, without traffic loss, recover from a complete loss/restart of the control plane. To support this, the Orion architecture provides a *Capability Readiness Protocol*. With this protocol, applications have a uniform way of specifying which data they require to resume operation, and which data they provide for other applications.

A *capability* is an abstraction of NIB state, each can be provided and consumed by multiple apps. Capability-based coordination keeps the Orion apps from becoming "coupled", in which a specific implementation of one app relies on implementation details or deployment configuration of another app. Such dependencies are a problem for iteration and release velocity. For example, multiple apps can provide the capability of "producing flows to program", and the Flow Manager can be oblivious to which ones are present in the domain.

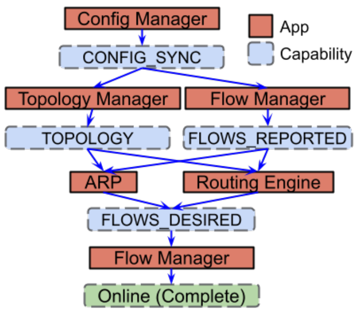


Figure 4: Capability Readiness graph for flow programming.

The Capability Readiness protocol requires, after a NIB failover, that all apps report readiness of their flows before Flow Manager begins reconciling NIB state to the physical switches. This prevents unintentional erasure of flow state from switches, which would lead to traffic loss. As Figure 4 shows, the required and provided data that each application specifies creates a directed acyclic graph of capabilities *depended upon* and *provided*, and thus the complete NIB state is reconciled consistently after any restart. Apps can have mutual dependency on different capabilities as long as they do not form a loop. A healthy Orion domain completes the full capability graph quickly on reconciliation, a condition we check in testing and alert on in production. Since this graph is static, such testing prevents introducing dependency loops.

In the event of a total loss of state, Config Manager retrieves the static topology from Chubby [5], an external, highly-available service for locking and small file storage. It then provides a `CONFIG_SYNC` capability to unblock Topology Manager and Flow Manager. The two connect to switches specified in the config and read switch states and programmed flows. Then, ARP and Routing Engine can be unblocked to generate intended flows that need to be programmed; they also provide their own `FLOWS_DESIRED` capability to Flow Manager, which proceeds to program the switches.

Apps that retrieve their state from a remote service must explicitly manage and support the case in which the service is unavailable or disconnected to prevent prolonged domain reconciliation delays. Cached data is typically used until the authoritative source of the inputs can be reached.

5 Orion-based Systems

Among the many design choices when implementing Orion to control a specific network, three prominent ones include the mapping of network elements to controllers, the method of controller to switch communication, and connectivity to external networks running standard routing protocols. We first review these common choices across two Google network architectures, Jupiter and B4, and then describe specific details for each architecture. Less relevant in a networking context, details of the NIB implementation are in the appendix.

Control domains. The choice of elements to control in an Orion domain involves multiple tradeoffs. Larger domains yield optimal traffic distributions and loss-free route sequencing for more intent changes, at the price of increased blast radius from any failure. In Jupiter, we use a hierarchy of partitioned Orion domains; in B4, a flat partitioning of Orion domains communicating with non-Orion global services. Each came with challenges in production, which we review in §7.

Control channel. As discussed in §3.1, we faced tradeoffs when designing the Control Plane Network (CPN) connecting Orion controllers to the data plane. The cost and complexity of a second network led us to a hybrid design where only the Top-of-Rack (ToR) switches were controlled in-band.

- *Separation of control and data plane:* When we embarked on building B4 and Jupiter, we embraced the SDN philosophy in its purest form: software-defined control of the network based on a logically centralized view of the network state outside the forwarding devices. To this end, we did not run any routing protocols on the switches. For the control plane, we ran a separate physical network connected to the switches' management ports. We ran conventional on-box distributed routing protocols on the CPN. Compared to the data plane network, the CPN has smaller bandwidth requirements, though it required N+1 redundancy.
- *CPN scale and cost:* Typical Clos-based data center networks are non-oversubscribed in the aggregation layers [28] with oversubscription of ToR uplinks based on the bandwidth requirements of compute and storage in the rack. A Clos network built with identical switches in each of its N stages will have the same number of switches (say, K) in all but two stages. The topmost stage will have $K/2$ switches since all ports are connected to the previous stage. The ToR stage will have SK switches, where S is the average oversubscription of uplinks compared to downlinks. Thus, the number of ToR switches as a fraction of the total is $2S/(2S+2N-3)$. In a Clos network with $N=5$ stages and an average ToR oversubscription, S, ranging from 2-4, ToR switches account for 36% to 53% of the total. Thus, not requiring CPN connectivity to them substantially reduces CPN scale and cost.
- *CPN cable management:* Managing ToRs inband removes the burden of deploying individual CPN cables to each rack spot in the datacenter.
- *Software complexity of inband ToRs:* Since ToRs are the leaf switches in a Clos topology, their inband management does not require on-box routing protocols. We designed simple in-band management logic in the switch stack to set the return path to the controller via the ToR uplink from which the ToR's CPU last heard from the controller.
- *Availability and debuggability considerations:* Over the years, we have hardened both the CPN and the inband-controlled ToR to improve availability. "Fail static" has been a key design to reduce vulnerability to CPN failures. Furthermore, we introduced in-band backup control of devices connected to the CPN for additional robustness.

External connectivity. We use BGP at the border of datacenter networks to exchange routes with Google's wide-area networks: B2 (which also connects to the Internet) and B4. These routes include machine addresses and also unicast and anycast IPs for Google services. BGP attributes such as communities, metrics, and AS path propagate state throughout Google's networks. In addition to reachability, this can include drain state, IPv6-readiness, and bandwidth for WCMP.

The use of BGP is a necessity for eventual route propagation to the Internet, but a design choice internally. The choice was made to simplify inter-connection with traditional, non-SDN routers as well as previous SDN software [18]. BGP also brings operator familiarity when defining polices to specify path preferences during topological changes.

An Orion app, **Raven** [34], integrates BGP and IS-IS into Orion. Raven exchanges messages with peers via Orion's Packet-I/O. Raven combines these updates with local routes from the NIB into a standard BGP RIB (Route Information Base). Routes selected by traditional Best-Path Selection are then sent, depending on policy, to peer speakers as BGP messages, as well as the local NIB in the form of `RouteAdvert` updates. To reduce complexity, Raven's associated BGP "router" is the abstract supernode provided by RE (§4.2).

Unfortunately, BGP is somewhat mismatched with our design principles: it uses streaming rather than full intent updates, its local view precludes a threshold-based fail static policy and global rebalancing during partial failures, and it ties control-plane liveness to data-plane liveness. In our production experience, we have had both kinds of uncorrelated failures, which, as in non-SDN networks, become correlated and cause a significant outage only due to BGP. By contrast, Orion's fail static policies explicitly consider control-plane and data-plane failure as independent. Adding fail static behavior to these adjacencies is an area of ongoing development.

5.1 Jupiter

We initially developed Jupiter [28], Google's datacenter network, with our first generation SDN-based control system, Onix [18]. The Orion-based solution presented here is a second iteration based on lessons from the Onix deployment.

The Jupiter datacenter network consists of three kinds of building blocks, each internally composed of switches forming a Clos-network topology: (i) aggregation blocks [28] connected to a set of hosts, (ii) FBRs (Fabric Border Routers, also called Cluster Border Routers in [28]) connected to the WAN/Campus network, and (iii) spine blocks that interconnect aggregation blocks and FBRs.

We organize Orion domains for Jupiter hierarchically as shown in Figure 5. First, we map physical Orion domains to the Jupiter building blocks. Each physical domain programs switches within that domain. Aggregation block domains es-

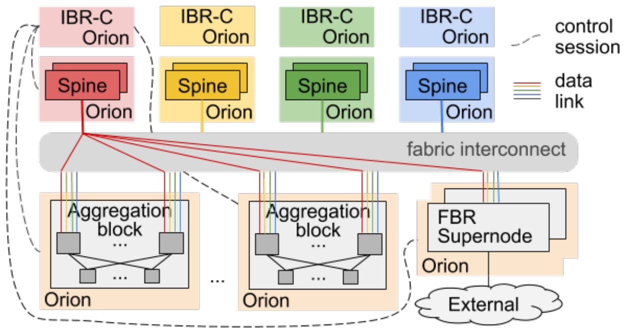


Figure 5: Jupiter topology overlaid with Orion domains partitioned by color. Colored links and spine domains are controlled by the respectively colored IBR-C. Uncolored aggregation block/FBR domains are controlled by all IBR-Cs. Only control sessions and data links of the red color are displayed.

tablish connectivity among hosts attached to it. FBR domains use Raven to maintain BGP sessions with fabric-external peers. Multiple spine blocks can map to a single Orion domain, but each domain must contain fewer than 25% of all spine blocks to limit the blast radius of domain failure.

Second-level Orion domains host a partitioned and centralized routing controller IBR-C (Inter-Block Routing Central). Operating over Routing Engine's abstract topology, IBR-C aggregates network states across physical domains, computes fabric-wide routes, and programs physical domains to establish fabric-wide reachability. While these virtual domains start from the same foundations, they do not contain some Orion core apps for controlling devices directly.

To avoid a single point of failure, we partitioned (or *sharded*) IBR-C into four planes called "colors," each controlling 25% of the spine blocks and hence a quarter of paths between each pair of spine blocks and aggregation blocks/FBRs. Therefore, the blast radius of a single controller does not exceed 25% of the fabric capacity. Sharding centralized controllers avoids failures where a single configuration or software upgrade affects the whole fabric. Additional protection was added to stage configuration changes and upgrades to avoid simultaneous updates across colors. While sharding provided higher resiliency to failures, the trade-off was an increased complexity in merging routing updates across colors in aggregation block and FBR domains, as well as a loss in routing optimality in case of asymmetric failures across colors. We have considered even deeper sharding by splitting aggregation blocks into separate domains, each controlling a portion of the switches. This option was rejected due to even higher complexity while marginally improving availability.

Figure 6 illustrates the fabric-level control flow of one IBR-C color. IBR-C subscribes to NIBs in all aggregation block/FBR domains and spine domains of the same color for state updates. After aggregation at Change Manager, the Solver computes inter-block routes and Operation Sequencer writes the next intended routing state into NIB tables of corresponding domains. IBR-D (Inter-Block Routing Domain), a

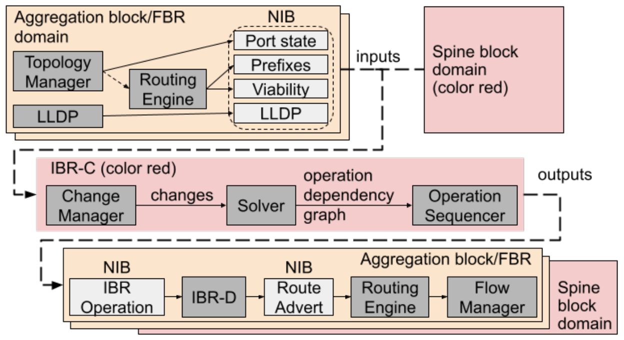


Figure 6: Jupiter fabric-level IBR-C control flow of one color.

domain-level component, merges routes from different IBR-C colors into `RouteAdvert` updates. Finally, Routing Engine and Orion Core program flows as shown in Figure 3.

Convergence. We care about two types of convergence in Jupiter: data plane convergence and control plane convergence. Data plane convergence ensures there are valid paths among all source/destination pairs (no blackholing) while control plane convergence restores (near) optimal paths and weights in the fabric. Workflows that require changes to the network use control plane convergence as a signal they can proceed safely. Convergence time is the duration between a triggering event and all work complete in data/control plane.

Jupiter's reaction to link/node failures is threefold. First, upon detection of link-layer disruption, switches adjacent to the failed entity perform local port pruning on the output group. However, this is not possible if no alternative port exists or peer failure is undetectable (e.g., switch memory corruption). Second, RE programs the domain to avoid this entity. This is similar to switch port pruning, but could happen on non-adjacent switches within the domain. For failures that do not affect inter-block routing, the chain of reaction ends here. Otherwise, in a third step, RE notifies IBR-C of the failure, as shown in Figure 6. When fabric-wide programming is complete, IBR-C signals the control plane has converged. This multi-tier reaction is advantageous for operations, as it minimizes data plane convergence time and thus traffic loss.

Since a single entity failure can lead to successive events in different domains (e.g., spine switch failure causing aggregation block links to fail), it could trigger multiple IBR-C and domain programming iterations to reach final convergence. Many independent events also happen simultaneously and get processed by Orion together, which can further delay convergence. Hence, we will evaluate Orion's performance in example convergence scenarios in §6.

Implementation Challenges. One challenge with the original Jupiter implementation [28] was optimally distributing traffic across multiple paths. Capacity across paths can differ due to link failures and topology asymmetry (e.g., different link count between a spine-aggregation block pair). In order to optimally allocate traffic, Jupiter/Orion employs WCMP to vary weights for each path and nexthop. Due to the precise

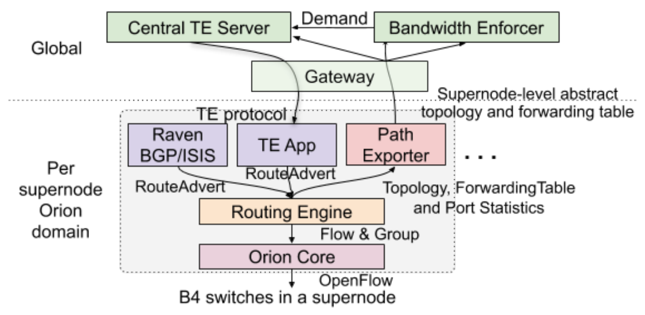


Figure 7: B4 Control Diagram

weight computation for each forwarding entry, weights need to be adjusted across the entire fabric to fully balance traffic. Another challenge was transient loops or blackholes during route changes. This is due to asynchronous flow programming in traditional routing protocols and in our previous SDN controller [18]. With Orion-based Jupiter, we implement end-to-end flow sequencing in both IBR-C and RE.

At the scale of Jupiter, network events arrive at IBR-C at a high frequency, which sometimes surpasses its processing speed. To avoid queue buildup, IBR-C prioritizes processing certain loss-inducing events (e.g., link down) over noncritical events (e.g., drain). Upon an influx of events, IBR-C only preempts its pipeline for loss-inducing events. It reorders/queues other events for batch processing upon the completion of higher priority processing. This is a trade-off to minimize traffic loss while avoiding starvation of lower priority events. §6 quantifies the benefits of this approach in more detail.

5.2 B4

Onix [18], the first-generation SDN controller for B4, ran control applications using cooperative multithreading. Onix had a tightly-coupled architecture, in which control apps share fate and a common threading pool. With Onix's architecture, it was increasingly challenging to meet B4's availability and scale requirements; both have grown by 100x over a five year period [13]. Orion solved B4's availability and scale problems via a distributed architecture in which B4's control logic is decoupled into micro-services with separate processes.

Figure 7 shows an overview of the B4 control architecture. Each Orion domain manages a B4 supernode, which is a 2-stage folded-Clos network where the lower stage switches are external facing (see details in [13]). In B4, Routing Engine sends ingress traffic to all viable switches in the upper stage using a link aggregation group (LAG), and uses two-layer WCMP to load-balance traffic toward the nexthop supernode.

The **TE App** is a traffic engineering agent for B4. It establishes a session with global TE server instances to synchronize the tunnel forwarding state. It learns TE tunneling ops from the primary TE server, and programs the ops via the `RouteAdvert` table. In addition, TE App also supports Fast ReRoute (FRR), which restores connectivity for broken tunnels by temporarily re-steering the traffic to the backup tunnel set or BGP/ISIS routes.

The **Path Exporter** subscribes to multiple NIB tables and exports the observed dataplane state to the global services. It reports the dataplane state at the supernode level, including the abstract topology (e.g., supernode-supernode link capacities), the abstract forwarding table (TE tunnels and BGP routes), and the abstract port statistics.

The **Central TE Server** [13,15] is a global traffic engineering service which optimizes B4 paths using the TE protocol offered by the TE App in each domain. The **Bandwidth Enforcer** [19] is Google's global bandwidth allocation service which provides bandwidth isolation between competing services via host rate limiting. For scalability, both the Central TE Server and Bandwidth Enforcer use the abstract network state provided by the Path Exporter.

6 Evaluation

We present microbenchmarks of the Orion NIB, followed by Jupiter evaluation using production monitoring traces collected since January 2018. Orion also improved B4 performance, as published previously [13].

NIB performance. To characterize NIB performance, we show results of a microbenchmark measuring the NIB's read/write throughput while varying the number of updates per batch. A batch is a set of write operations composed by an app that updates rows of different NIB tables atomically. In Figure 8, we observe throughput increase as the batch size becomes larger. At 50K updates per batch, the NIB achieves 1.16 million updates/sec in read throughput and 809K updates/sec in write throughput.

Write throughput at 500 updates per batch sees a decline. This reveals an implementation choice where the NIB switches from single-threaded write to multi-threaded write if the batch size is greater than 500. When the batch size is not large enough, up-front partitioning to enable parallelism is more expensive than the performance improvement. This fixed threshold achieves peak performance on sampled production test data and performs well in production overall. It could be removed in favor of a more dynamic adaption strategy to smooth the throughput curve.

Data and control plane convergence. One key Jupiter performance characteristic is convergence time (§5.1). We measure convergence times in several fabrics, ranging from 1/16-size Jupiter to full-size Jupiter (full-size means 64 aggregation blocks [28]). Figure 9 captures data and control plane convergence times of three types of common daily events in our fleet. The measurements are observed by Orion; switch-local port pruning is an independent decision that completes within a few milliseconds without Orion involvement.

In node/link down scenarios, the data plane converges within a fraction of a second. Both data and control plane convergence times become longer as the fabric scales up by 16x. This is mainly because a larger number of aggregation

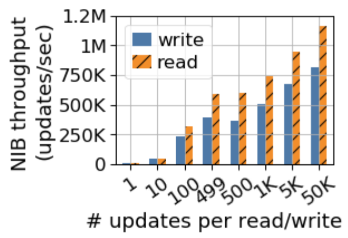


Figure 8: NIB throughput.

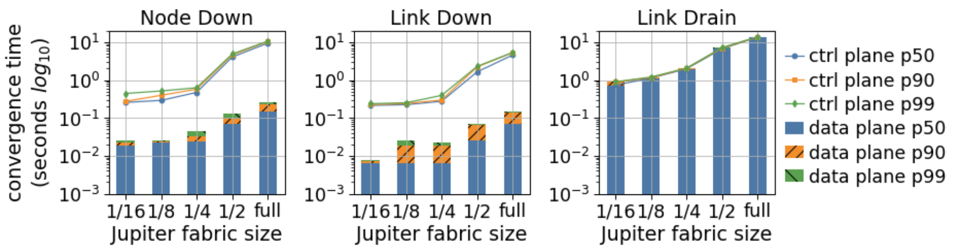


Figure 9: Jupiter data/control plane convergence time in response to various network events.

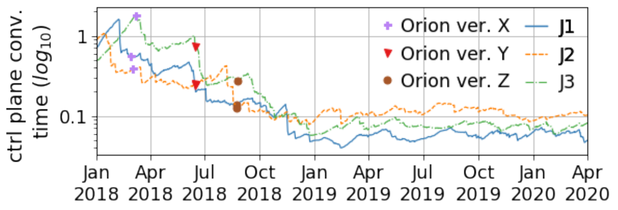


Figure 10: Time series of full fabric control plane convergence on three large Jupiter fabrics. Y-axis is normalized to the baseline convergence time in January 2018. Orion releases with major performance improvements are highlighted by markers.

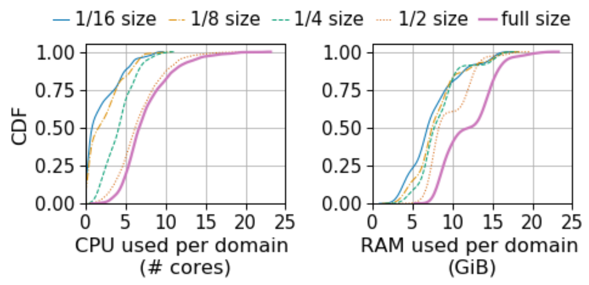


Figure 11: Orion CPU/RAM usage in Jupiter domains.

blocks and spine blocks require more affected paths to be re-computed and re-programmed. Data plane convergence is 35-43x faster than control plane convergence, which effectively keeps traffic loss at a minimum.

Jupiter's reaction to link drains is slightly different from failure handling. Drains are lossless, and do not include an initial sub-optimal data plane reaction to divert traffic. Instead, Orion only shifts traffic after computing a new optimal routing state. Therefore, data and control plane convergence are considered equal. Overall, control plane convergence time for link drain is on par with node/link down scenarios.

We have continuously evolved Orion to improve Jupiter scalability and workflow velocity. Key to this were enhancements in IBR-C such as prioritized handling of select updates, batch processing/reordering, and a conditionally preemptive control pipeline. Figure 10 shows the trend of three large fabrics from January 2018 to April 2020; the control plane convergence time in January 2018 was before these improvements. Deployed over three major releases, each contributing an average 2-4x reduction, the new processing pipeline (§5.1) delivered a 10-40x reduction in convergence time.

Controller footprint: CPU and memory. Orion controller jobs run on dedicated network control servers connected to the CPN. This pool is comprised of regular server-class platforms. We measure CPU and memory usage of each controller job (including all three replicas) and group them by domain. Figure 11 shows that even in a full-size Jupiter, Orion domains use no more than 23 CPU cores and 24 GiB of memory.

7 Production Experience

We briefly review some challenging experiences with Orion when adhering to our production principles of limited blast-radius and fail-static safety, and some more positive experiences from following our software design principles.

7.1 Reliability and robustness

Failure to enforce blast radius containment. As described in §5.1, the inter-block routing domain is global but sharded into four colors to limit the blast radius to 25%. A buggy IBR-C configuration upgrade caused a domain to revoke all forwarding rules from the switches in that domain resulting in 25% capacity loss. Since high-priority traffic demand was below 25% of the fabric's total capacity, only after all four domains' configurations were pushed did the workflow flag (complete) high-priority packet loss in the fabric. To prevent such a "slow wreck" and enforce blast radius containment, subsequent progressive updates proceeded only after confirming the previous domain's rollout was successful, and not simply the absence of high-priority packet loss.

Failure to align blast radius domains. A significant Orion outage occurred in 2019 due to misalignment of job-control and network-control failure domains. Like many services at Google, these Orion jobs were running in Borg cells [31]. Although the Orion jobs were themselves topologically-scoped to reduce blast radius (§3.1), their assignment to Borg cells was not. As described in the incident report [9], when a facility maintenance event triggered a series of misconfigured behaviors that disabled Orion in those Borg cells, the resulting failure was significantly larger than Google's networks had been previously designed to withstand. This outage highlighted the need for all management activities (job control, configuration update, OS maintenance, etc.) to be scoped and rate-limited in a coordinated manner to fully realize the principle of blast-radius reduction. In addition, it highlighted a gap in our fail-static implementation with regards to BGP.

Failure to differentiate between missing and empty state. In 2018, a significant B4 outage illustrated a challenge when integrating Orion and non-Orion control layers. A gradual increase in routes crossed an outdated validation threshold on the maximum number of routes a neighborhood should receive from attached clusters. Consequently, the TE app suppressed the data to the B4-Gateway, which correctly failed static using cached data. However, subsequent maintenance restarted B4-Gateway instances while in this state, clearing the cached data. Lacking any differentiation between *empty* data and *missing* data, subsequent actions by the TE Server and Bandwidth Enforcer resulted in severe congestion for low-priority traffic. Within Orion, the capability readiness protocol prevents reading missing or invalid derived state.

Orion's post-outage improvements and continuous feature evolution such as loss-free flow sequencing and in-band CPN backup, brought substantial improvements in data plane availability to Jupiter (50x less unavailable time) and B4 (100x less unavailable time [13]).

7.2 Deployability and software design

With Orion, we moved to a platform deeply integrated with Google's internal infrastructure. This enabled us to leverage existing integration testing, debugging, and release procedures to increase velocity. We also moved from a periodic release cycle to a continuous release: we start software validation of the next version as soon as the current version is ready. This reduces the amount of "queued up" bugs, which improves overall velocity.

Release cadence. SDN shifts the burden of validation from "distributed protocols" to "distributed algorithms", which is smaller. Software rollouts are also faster: the number of Network Control Servers is orders of magnitude smaller than the number of network devices. The embedded stack on the devices is also simpler and more stable over time.

Orion's micro-service-based architecture leads to clear component API boundaries and effective per-component testing. Onix, on the other hand, was more monolithic, and our process required more full-system, end-to-end testing to find all newly introduced bugs which was less efficient.

In steady state, after initial development and productionization, it took about five months to validate a new major Onix release. The process was manual, leveraging a quality assurance team and iterative cherry-picking of fixes for discovered issues. With Orion, we shrank validation latency to an average of 14.7 days after the initial stabilization phase, with a target of eventually qualifying a release every week.

Release granularity. As a distributed, micro-service-based architecture, each Orion application could release at its own cadence. In our move from the monolithic Onix to Orion, we have not yet leveraged this flexibility gain. Since Orion is widely deployed and has high availability demands, we strive to test all versions that run at the same time in production, for instance as some applications are upgraded but other upgrades are still pending. An increase in release granularity would increase both the skew duration and total number of combinations that need to be tested. Therefore, we releas all Orion applications that make up a particular product (e.g., B4 or Jupiter) together.

Improved debugging. Serializing all intent and state changes through the NIB facilitates debugging: Engineers investigating an issue can rely on the fact that the order of changes observed by all applications in the distributed system is the same and therefore establish causality more easily.

Storing the stream of NIB updates for every Orion deployment also allowed us to build replay tooling that automatically reproduces bugs in lab environments. This was first used in the aftermath of an outage: only the precise ordering of programming operations that occurred in production, replayed to a lab switch, reliably reproduced a memory corruption bug in Google's switch firmware. This enabled delivering as well as, more importantly, verifying the fix for this issue.

8 Conclusion and future work

This paper presents Orion, the SDN control plane for Google's Jupiter datacenter and B4 Wide Area Networks. Orion decouples control from individual hardware elements, enabling a transition from pair-wise coordination through slowly-evolving protocols to a high-performance distributed system with a logically centralized view of global fabric state. We highlight Orion's benefits in availability, feature velocity, and scale while addressing challenges with SDN including aligning failure domains, inter-operating with existing networks, and decoupled failures in the control versus data planes.

While Orion has been battle-tested in production for over 4 years, we still have open questions to consider as future work. These include (i) evaluating the split between on-box and off-box control, (ii) standardizing the Control to Data Plane Interface with P4Runtime API [10], (iii) exploring making the NIB durable, (iv) investigating fail-static features in BGP, and (v) experimenting with finer-grained application release.

Acknowledgments For their significant contributions to the evolution of Orion, we thank Deepak Arulkannan, Arda Balkanay, Matt Beaumont-Gay, Mike Bennett, Bryant Chang, Xinming Chen, Roshan Chepuri, Dharti Dhami, Charles Eckman, Oliver Fisher, Lisa Fowler, Barry Friedman, Goldie Gadde, Luca Giraudo, Anatoliy Glagolev, Jia Guo, Jahangir Hasan, Jay Kaimal, Stephen Kratzer, Nanfang Li, Zhuotao Liu, Aamer Mahmood, John McCullough, Bhuva Muthukumar, Susan Pence, Tony Qian, Bharath Raghavan, Mike Rubin, Jeffrey Seibert, Ruth Sivilotti, Mukarram Tariq, Malveeka Tewari, Lisa Vitolo, Jim Wanderer, Curt Wohlgemuth, Zhehua Wu, Yavuz Yetim, Sunghwan Yoo, Jiaying Zhang, and Tian Yu Zhang. We also thank our shepherd, Katerina Argyraki, Jeff Mogul, David Wetherall, and the anonymous reviewers for their valuable feedback.

References

[1] Saksham Agarwal, Shijin Rajakrishnan, Akshay Narayan, Rachit Agarwal, David Shmoys, and Amin Vahdat. Sincronia: Near-Optimal Network Design for Coflows. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 16–29, 2018.

[2] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *7th USENIX Symposium on Networked Systems Design and Implementation (NSDI 10)*, San Jose, CA, April 2010. USENIX Association.

[3] Arista. Arista EOS Whitepaper. https://www.arista.com/assets/data/pdf/EOSWhitepaper.pdf, 2013.

[4] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, and Guru Parulkar. ONOS: Towards an Open, Distributed SDN OS. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, page 1–6, New York, NY, USA, 2014. Association for Computing Machinery.

[5] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[6] Matthew Caesar, Donald Caldwell, Nick Feamster, Jennifer Rexford, Aman Shaikh, and Jacobus van der Merwe. Design and Implementation of a Routing Control Platform. In *Symposium on Networked Systems Design and Implementation (NSDI)*, NSDI'05, page 15–28, USA, 2005. USENIX Association.

[7] Open Networking Foundation. SDN Architecture Overview. https://www.opennetworking.org/wp-content/uploads/2013/02/SDN-architecture-overview-1.0.pdf, 2013.

[8] Google. Protocol buffers: Google's data interchange format. https://code.google.com/p/protobuf/, 2008.

[9] Google Cloud Networking Incident 19009, 2019. https://status.cloud.google.com/incident/cloud-networking/19009.

[10] The P4.org API Working Group. P4 Runtime Specification. https://p4.org/p4runtime/spec/v1.2.0/P4Runtime-Spec.html, 2020.

[11] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 29–42, 2017.

[12] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, pages 15–26, 2013.

[13] Chi-Yao Hong, Subhasree Mandal, Mohammad A. Alfares, Min Zhu, Rich Alimi, Kondapa Naidu Bollineni, Chandan Bhagat, Sourabh Jain, Jay Kaimal, Jeffrey Liang, Kirill Mendelev, Steve Padgett, Faro Thomas Rabe, Saikat Ray, Malveeka Tewari, Matt Tierney, Monika Zahn, Jon Zolla, Joon Ong, and Amin Vahdat. B4 and After: Managing Hierarchy, Partitioning, and Asymmetry for Availability and Scale in Google's Software-Defined WAN. In *SIGCOMM'18*, 2018.

[14] Theo Härder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, 1983.

[15] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jonathan Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a Globally Deployed Software Defined WAN. In *Proceedings of the ACM SIGCOMM Conference*, Hong Kong, China, 2013.

[16] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real time network policy checking using header space analysis. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 99–111, Lombard, IL, April 2013. USENIX Association.

[17] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 15–27, Lombard, IL, April 2013. USENIX Association.

[18] Teemu Koponen, Martín Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: A Distributed Control Platform for Large-scale Production Networks. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[19] Alok Kumar, Sushant Jain, Uday Naik, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C. Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, Mathieu Robin, Aspi Siganporia, Stephen Stuart, and Amin Vahdat. BwE: Flexible, Hierarchical Bandwidth Allocation for WAN Distributed Computing. In *SIGCOMM'15*, 2015.

[20] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[21] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 38–49. IEEE, 2013.

[22] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. HPCC: High Precision Congestion Control. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 44–58, New York, NY, USA, 2019. Association for Computing Machinery.

[23] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM Computer Communication Review (CCR)*, 38:69–74, 2008.

[24] J. Medved, R. Varga, A. Tkacik, and K. Gray. OpenDaylight: Towards a Model-Driven SDN Controller architecture. In *Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks 2014*, pages 1–6, 2014.

[25] Microsoft. SONiC System Architecture. `https://github.com/Azure/SONiC/wiki/Architecture`, 2016.

[26] Jeffrey C. Mogul, Drago Goricanec, Martin Pool, Anees Shaikh, Douglas Turk, Bikash Koley, and Xiaoxue Zhao. Experiences with Modeling Network Topologies at Multiple Levels of Abstraction. In *17th Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[27] Timothy Nelson, Andrew D. Ferguson, Michael J. G. Scheer, and Shriram Krishnamurthi. Tierless Programming and Reasoning for Software-Defined Networks. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[28] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Hanying Liu, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *SIGCOMM '15*, 2015.

[29] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full SHA-1. In *Annual International Cryptology Conference*, pages 570–596. Springer, 2017.

[30] Amin Tootoonchian and Yashar Ganjali. HyperFlow: A Distributed Control Plane for OpenFlow. In *Proceedings of the 2010 Internet Network Management Conference on Research on Enterprise Networking*, INM/WREN'10, page 3, USA, 2010. USENIX Association.

[31] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[32] Anduo Wang, Xueyuan Mei, Jason Croft, Matthew Caesar, and Brighten Godfrey. Ravel: A Database-Defined Network. In *Proceedings of the Symposium on SDN Research*, SOSR '16, New York, NY, USA, 2016. Association for Computing Machinery.

[33] Hong Yan, David A. Maltz, T. S. Eugene Ng, Hemant Gogineni, Hui Zhang, and Zheng Cai. Tesseract: a 4D network control plane. In *Symposium on Networked Systems Design and Implementation (NSDI)*, pages 27–27, 2007.

[34] Kok-Kiong Yap, Murtaza Motiwala, Jeremy Rahe, Steve Padgett, Matthew Holliman, Gary Baldus, Marcus Hines, Taeeun Kim, Ashok Narayanan, Ankur Jain, Victor Lin, Colin Rice, Brian Rogan, Arjun Singh, Bert Tanaka, Manish Verma, Puneet Sood, Mukarram Tariq, Matt Tierney, Dzevad Trumic, Vytautas Valancius, Calvin Ying, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Taking the Edge off with Espresso: Scale, Reliability and Programmability for Global Internet Peering. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 432–445, New York, NY, USA, 2017. Association for Computing Machinery.

[35] Soheil Hassas Yeganeh and Yashar Ganjali. Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications. In *Workshop on Hot Topics in SDN (HotSDN)*, 2012.

[36] Junlan Zhou, Malveeka Tewari, Min Zhu, Abdul Kabbani, Leon Poutievski, Arjun Singh, and Amin Vahdat. WCMP: Weighted Cost Multipathing for Improved Fairness in Data Centers. In *Proceedings of the Ninth European Conference on Computer Systems*, page Article No. 5, 2014.

9 Appendix

An example table schema definition in the NIB.

```
message Link {
  enum Status {
    STATUS_UNKNOWN = 0;
    DOWN = 1;
    UP = 2;
  }
  optional NIBHeader nib_header = 1;
  optional string name = 2;
  optional Status status = 3;
  // references Node.nib_header.id
  optional string src_node_id = 4;
  // references Port.nib_header.id
  optional string src_port_id = 5;
  // references Node.nib_header.id
  optional string dst_node_id = 6;
  // references Port.nib_header.id
  optional string dst_port_id = 7;
}
```

9.1 Orion Core Implementation

To productionize the Orion systems described in this paper, we address some common engineering challenges in the Orion Core. In particular, Orion's architectural choices require great care with both the memory footprint and performance of all data flowing through the NIB. Here, we illustrate multiple implementation choices we made to scale this architecture.

State replication and synchronization. Orion enables a large group of developers in Google to author new SDN applications. To simplify interacting with the intent/state stored in the NIB, we synchronize all relevant data into an app-local cache that trails the NIB's authoritative state. The state visible to each application is always a prefix of the sequential atomic writes applied to the NIB. Each application's ability to see a prefix and therefore (potentially) not the most recent NIB state is acceptable given that the NIB itself is only a trailing reflection of the global system state. Applications do not subscribe to the NIB data they wrote as they were previously responsible for the write.

The app-local cache allows developers to access data as they would access an in-memory hash table. Additionally, since data is local, developers write fast applications by default as they do not have to reason about network round-trips to load data from the NIB. This replication approach requires transmitting cached data efficiently to reduce reaction time to data-plane events. While sequential communication through the NIB has many upsides, e.g., reduced complexity and traces for debugging, it should not dominate reaction time to data-plane events.

Additionally, we require a compact memory representation of all state to support a large number of micro-services, with overlapping subsets of the NIB state in their local caches. Protocol buffers are not well-suited for this requirement because they hold copies of the same nested and repeated fields in more than one entity (e.g., the same output port ID string in multiple forwarding rules). Orion works around this space inefficiency by exploiting the read-only nature of cached data. As all entity mutations are sent directly to the NIB for full serializability, it is possible to de-duplicate select sub-messages and data types in the app-local cache to conserve memory. Given our schema, this reduces memory overhead by more than 5x while retaining the same read-only API offered by regular protocol buffers.

Hash table / key size. Both the dictionary implementation as well as the key length used for the data in the NIB influence memory utilization. In early testing, about 50% of memory was consumed by key-strings and dictionary data structures.

Many data entries in the NIB use descriptive identifiers. For instance, a physical port on a switch, represented in the `Port` table of the NIB, combines the FQDN (Fully Qualified Domain Name) of its parent switch and its own port name identifier as the entity key in the NIB. As the payload per port is only a handful of simple data types, the key size may be larger than the payload. To reduce the memory impact of many entities with relatively small payloads, the NIB initially stored all data in a *trie* modelled after [21]. This was advantageous both because of prefix compression and because it enables inexpensive creation of consistent snapshots.

While we retain the M-Trie data structure for its copy-on-write capability, we have changed to storing a SHA-1 hash of the identifier only. This reduces the NIB memory footprint at the price of a theoretical, but not practical [29], collision risk. If a collision occurred, it would require human intervention.

Large updates. Large updates must be reflected in all SDN apps that consume changed data as their input. In some situations, for instance extensive rerouting, or, when an SDN app restarts and needs to be brought in sync with the NIB, the size of changed intent/state can be multiple gigabytes.

To reduce wall-clock time for such synchronization and peak memory spikes, Orion handles large atomic writes by splitting them into many small ones with special annotations delineating that the small writes are part of one atomic batch. This allows pipeline processing in the NIB, data transfer and cache application on the receiver. As long as the partial small updates are guaranteed to be exposed only after the complete set is available, this optimization is transparent.

In case of synchronizing client application and NIB state after a disconnect, the update size is kept to a minimum to reduce transmission delay. Each application supplies the NIB with a vector of ID-hashes of entities and entity hashes. Matching ID-hashes and entity hashes between the NIB and the application cache tells the NIB to skip updating such entities. All other entities with mismatched or missing hashes will be replaced/deleted/inserted accordingly.

Software upgrade strategy. Since the controller is distributed across multiple apps running in separate binaries, software upgrades lead to version skew across applications. While this version skew is commonly short-lived, it can persist in rare cases as manual intervention is required to resolve the skew when an upgrade is stuck in a partially succeeded state. Given this, a key functionality that Orion provides to developers is enabling features atomically across applications when all are ready. Likewise, Orion can deactivate features atomically in case a participating app no longer supports it.

As an example, take two applications that interact through a NIB table A. Consider that the interaction pattern is changed to go through table B in a later version. Both applications have to ensure that the appropriate table is used depending on the readiness of the peer application. To reduce bug surface, Orion allows describing which applications need to support a certain feature for it to be enabled and abstracts this negotiation from the application developers. Developers simply protect the new interaction through table B inside a condition check, the condition is automatically marked true by Orion based on the feature readiness of both sides. If one of the two applications interacts through the old table, the feature stays disabled. Once both applications support the feature, Orion will enable it atomically.

9.2 Jupiter Fabric Drain

Orion provides support for network management operations. In order to take a network element out of service for repair or upgrade, the control systems needs to first drain it, i.e. divert traffic from it. Orion provides explicit handling and tracking of drain state. Drain Conductor (DC) is an Orion application in a separate non-sharded virtual control domain, providing an external API for network management systems to drain, undrain and check drain status for each element. Once DC receives a drain request for a networking element, it persists the new drain intent in Chubby [5] and dispatches the drain intent to the NIB in a physical domain. Drain Agent (DA), an application running in each physical domain, subscribes to drain intent changes published by DC, and dispatches the drain intent across routing applications. A routing app processes the drain intent by de-preferencing a networking element and updates the drain status. Finally, DC subscribes to drain statuses across physical domains and provides drain acknowledgements via its API.

Metha: Network Verifiers Need To Be Correct Too!

Rüdiger Birkner* Tobias Brodmann* Petar Tsankov Laurent Vanbever Martin Vechev

**These authors contributed equally to this work.*

ETH Zürich

Abstract

Network analysis and verification tools are often a godsend for network operators as they free them from the fear of introducing outages or security breaches. As with any complex software though, these tools can (and often do) have bugs. For the operators, these bugs are not necessarily problematic except if they affect the precision of the model as it applies to their specific network. In that case, the tool output might be wrong: it might fail to detect actual configuration errors and/or report non-existing ones.

In this paper, we present Metha, a framework that systematically tests network analysis and verification tools for bugs in their network models. Metha automatically generates syntactically- and semantically-valid configurations; compares the tool's output to that of the actual router software; and detects any discrepancy as a bug in the tool's model. The challenge in testing network analyzers this way is that a bug may occur very rarely and only when a specific set of configuration statements is present. We address this challenge by leveraging grammar-based fuzzing together with combinatorial testing to ensure thorough coverage of the search space and by identifying the minimal set of statements triggering the bug through delta debugging.

We implemented Metha and used it to test three well-known tools. In all of them, we found multiple (new) bugs in their models, most of which were confirmed by the developers.

1 Introduction

It's Friday night and you are about to push an important (network) configuration update in production. Usually, you would feel terribly nervous doing so as there is always the possibility that you may have missed something. You are only too aware that misconfigurations happen frequently and can lead to major network outages [22,24,27]. Tonight though you feel confident when pressing "deploy" as you have confirmed the correctness of your configuration update using a state-of-the-art configuration verifier. A few minutes later, your phone rings: none of your customers can reach the Internet anymore.

This fictitious situation illustrates an intrinsic problem with validation technologies: their results can only be completely trusted if their analysis is sound and complete. As with any complex software though, these tools can (and often do) have bugs. To be fair, this is not surprising: building an accurate and faithful network analysis tool is extremely difficult. Among others, one not only has to precisely capture all the different protocols' behaviors, but also all of the quirks of their specific implementations. Unfortunately, every vendor, every OS, every device can exhibit slightly different behaviors under certain conditions. For all it takes, these behaviors might be the results of bugs themselves. And yet, failing to accurately capture these behaviors—as we show—can lead to incorrect and possibly misleading analysis results.

A fundamental and practical research question is therefore: *How can developers make sure that their network analysis and verification tools are correct?*

Metha We introduce Metha, a system that thoroughly tests network analysis and verification tools to find subtle bugs in their network models using black-box differential testing. Metha automatically finds model discrepancies by generating input configurations and comparing the output of the tool under test with the output produced by the actual router software. For every discovered discrepancy, Metha provides a minimal configuration that helps developers pinpoint the bug. Later on, these configurations can be used to build up an adequate test suite for current and future network tools.

Challenges Precisely identifying bugs in network analyzers' models is challenging for at least three reasons. First, the search space of possible configurations is gigantic: there are hundreds of configuration statements, each of which can take many possible parameters. And yet, as our analysis reveals, most of the bugs only manifest themselves when specific configuration statements/values are present. Second, systematically exploring the search space is highly non-trivial (independently of its size) as one not only needs to generate syntactically-valid configurations, but also semantically-valid ones that involve all features and their interactions. Failing to do so could lead to miss bugs, hence lowering coverage.

Finally, upon finding a configuration triggering a discrepancy, figuring out the exact subset of statements requires to solve another tricky combinatorial search.

Insights Metha addresses the above challenges by first reducing the search space through restricting the parameters to their boundary values. Metha ensures thorough coverage of the search space by phrasing the search as a combinatorial testing problem and targeting the search towards single and pairwise interactions of configuration statements. To ensure syntactically- and semantically-valid configurations Metha relies on a hierarchical grammar-based approach. Finally, Metha employs delta debugging to identify the minimal bug-inducing set of configuration statements to help the developer better understand and reproduce the bug.

Bugs found We demonstrate Metha's effectiveness in practice by finding 62 *real-world bugs* across three popular network analysis and verification tools – Batfish [17], NV [13], and C-BGP [28] – 59 of which have been confirmed by the tools' developers. The majority of the discovered bugs are subtle, silent bugs that undermine the soundness of the tools' results. That is, they could lead operators to incorrect conclusions that their networks behave correctly, while, in fact, they do not.

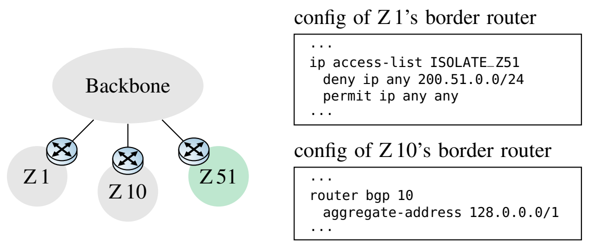
Our experiments also demonstrate that Metha's key components are essential for its effectiveness. In particular, a random baseline only found 3 bugs, while Metha found 20 bugs with the same number of tests. Last but not least, through our interactions with the tools' developers, we confirm that Metha's minimal configuration examples are indeed useful: some of the developers are already using them to analyze and fix the bugs Metha has discovered.

Contributions In summary, our main contributions are:

- A testing system capable of finding bugs in the network models of state-of-the-art network analyzers (§3).
- A formulation of the search problem in terms of combinatorial testing (§5).
- A precise localization procedure relying on delta debugging to isolate bugs and pinpoint the configuration statements causing them (§6).
- An end-to-end implementation of Metha[1] supporting both Cisco IOS and Juniper configurations (§7).
- An evaluation showing that Metha finds real (and unknown) bugs in all the tested tools (§8).

Limitations Metha treats the network verification tool it is testing as a black box. Hence, it cannot localize the bug within the tool's actual code. This task is left to the developer. However, by identifying the configuration statements responsible for the bug and by creating a minimal configuration that showcases it, the developer has a good starting point for her work. Similarly, Metha cannot detect whether two observed bugs that are triggered by different configuration statements are caused by the same bug in the underlying network model.

[1]Available at https://github.com/nsg-ethz/Metha



config of Z 1's border router

```
...
ip access-list ISOLATE_Z51
  deny ip any 200.51.0.0/24
  permit ip any any
...
```

config of Z 10's border router

```
...
router bgp 10
  aggregate-address 128.0.0.0/1
...
```

Figure 1: Zone 51 has to be isolated from all the other zones. This is achieved through access-lists at the border routers with the exception of zone 10 where it was forgotten.

2 Motivation

We now illustrate how subtle bugs in the network model of network analyzers can lead operators to deploy erroneous configuration changes. We start with two case studies on common configuration features known for easily causing forwarding anomalies: route aggregation and redistribution. In these situations, validating the change with an analyzer is of utmost importance, provided the analysis is correct. We end with a collection of Cisco IOS configuration statements whose semantics were not correctly captured by Batfish [17]. The bugs in this Section were discovered by Metha.

2.1 Example 1: Excess Null Route

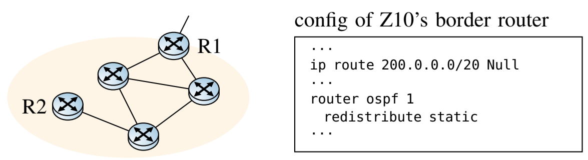
Consider the network in Fig. 1. It consists of a backbone with multiple zones attached to it. The backbone and the zones are interconnected using BGP. Each zone receives a default-route from the backbone. Zone 51 hosts critical infrastructure in the prefix `200.51.0.0/24`, which should not be accessible from any other zone. To enforce this, the routers connecting the zones to the backbone have an access-list (ACL) in place to filter that traffic. However, in zone 10, this ACL was forgotten and, instead, there happens to be a left-over statement from a previous configuration: "`aggregate-address 128.0.0.0 128.0.0.0`". This statement directs the router to advertise the specified aggregate route *if* any more-specific BGP routes in that range exist in the routing table.

Property violation Due to the lack of an ACL on the border router, the requirement that mandates to keep zone 51 isolated is violated: traffic from zone 10 can reach zone 51.

Analyzer mishap When used on the network above, Batfish, a recent network validation tool, will falsely assert that zone 51 is isolated. The problem is due to the semantics of the left-over `aggregate-address` statement. Batfish wrongly activates the aggregate because of a non-BGP route in the routing table and installs the `null` route. Because of this `null` route, Batfish wrongly assumes that all traffic in zone 10 falling within the aggregate range will be dropped. In practice, the routers only install a `null` route if a BGP route within the aggregate is present, which is not the case here.

Feature	Description	Possible Consequence
`max-metric router-lsa`	The model sets maximum metric not only for point-to-point links, but also for stub links. This should only be done when the keyword `include-stub` is used.	A router might appear to be free of traffic and safe to reboot, even though it is not.
`default-information originate`	The OSPF routing process should only generate a default-route if the route table has a default-route from another protocol. The model, however, also announces a default-route if there is one in the routing table of different OSPF type, i.e., E1 type.	Additional default-routes might appear in the routing tables.
`distance XX`	The model does not consider any changes to the administrative distance.	The forwarding state could be completely wrong.
`area X range A.B.C.D/Y`	When summarizing routes between OSPF areas, the model does not insert a null route for the summary to prevent routing loops.	A routing loop could be falsely detected.
`set community no-export`	When redistributing a static route into BGP and setting the `no-export` community, the model still advertises the route to its eBGP neighbors.	Reachability properties could be falsely asserted.
`neighbor A.B.C.D maximum-prefix X`	Even when a BGP neighbor advertises more prefixes than the specified threshold, the model does not drop the peering to the neighbor.	Reachability properties could be falsely asserted.

Table 1: A selection of Cisco IOS configuration features that are not correctly modelled by Batfish [17] as discovered by Metha.



```
...
ip route 200.0.0.0/20 Null
...
router ospf 1
  redistribute static
...
```

Figure 2: All routers should be able to reach the Internet. The static route at R2 creates a blackhole and violates that.

2.2 Example 2: Incomplete Redistribution

Consider the small company network depicted in Fig. 2. It consists of a single OSPF area. R1 acts as Internet gateway and announces a default-route internally. A static route on R2 drops all the traffic for `200.0.0.0/20` by directing it to the `null` interface. This is intended. What is not intended, however, is the `redistribute static` command at R2.

Property violation The following reachability property must always hold: all routers, with the exception of R2, are able to reach the entire Internet. However, this property, is violated since R2 redistributes the static route in the network and, in turn, creates a blackhole for `200.0.0.0/20`.

Analyzer mishap When run on this network, Batfish will falsely attest that all routers, with the exception of R2 can reach the entire Internet. The problem is the redistribution command. By default, Cisco routers only redistribute classful networks [7] and only by specifying the `subnets` keyword, they also redistribute any subnets of them. Less-specific networks however are redistributed regardless of the `subnets` keyword (e.g., `200.0.0.0/20` is less-specific than the corresponding class C network `200.0.0.0/24`). Batfish's network model does not incorporate that as it only redistributes classful networks and not less-specific networks.

2.3 Selection of Bugs

In addition to the two bugs illustrated in the previous examples, we found several other configuration statements that trigger bugs. We present a selection of them in Table 1 alongside a short description of the observed behavior and possible consequences. All of the presented bugs concern Cisco IOS configuration statements. In our tests, we also used Juniper configurations and found that, in many cases, the same bugs occur. Hence, some of these bugs are not due to vendor-specific behaviors, but due to general inaccuracies of the network model.

3 Overview

In this section, we first present the key insights enabling Metha to efficiently uncover bugs in network analyzers. Then, we provide a high-level overview of Metha.

3.1 Key Insights

The main challenge in testing network analyzers is that bugs may occur rarely and only for very specific configurations, which we address with a combination of five insights:

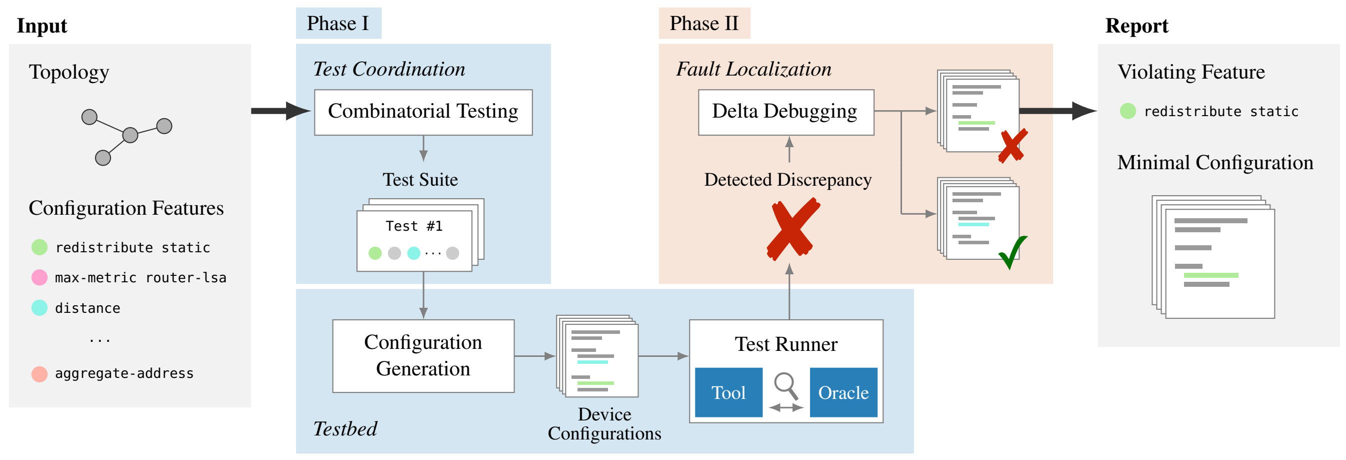


Figure 3: Metha generates a test suite based on the test topology and supplied configuration features. The testbed runs one test after another and compares the computed routing tables of the tool under test to those of an oracle. It then analyzes every discrepancy to localize all the bugs and creates a report for each one of them.

Producing valid inputs with grammar-based generation When testing network analyzers, it is of utmost importance to use syntactically- and semantically-valid configurations, meaning the configurations need to be parseable and constraints have to be met such that actual computation takes place in the network. Our key insight is to use a hierarchical grammar-based approach. Approaching it hierarchically allows to resolve the intra- and inter-device constraints. This provides the structure that is then completed using grammar-based configuration generation ensuring syntactical validity.

Reducing the search space through boundary values The search space of all possible configurations is prohibitively large. Even a single parameter, such as an OSPF cost, for example, already has 2^{16} possible values to test. By focusing the testing on the boundary values (the minimum, maximum, and a normal value), we reduce the search space significantly.

Exploring the search space with combinatorial testing Network devices support a wide variety of configuration features that all need to be tested not just by themselves, but also their interactions. Hence, we use combinatorial testing to design a test suite that systematically covers all pairwise interactions of configuration features.

Comparing the tested tool's output to ground truth Detecting crash bugs is straightforward as the tool will just fail or report an error. Silent bugs, on the other hand, can only be detected by comparing the output to a ground truth, which is hard to come by. We address this by leveraging a testbed running real router images as an oracle.

Isolating bugs with delta debugging Lastly, once one identifies a network configuration that triggers a bug, one needs to identify the configuration statements causing it to provide any useful insights to the tool's developer. Therefore, we use iterative delta debugging to obtain a minimal configuration example reproducing the bug.

3.2 Metha

Metha operates in two phases as shown in Fig. 3: First, it aims to find network configurations exhibiting discrepancies between the tool under test and the oracle. To that end, the test coordination determines all the tests to run in the testbed. Second, it identifies the configuration statements responsible for the observed discrepancies through fault localization.

Input and output Metha takes two inputs: *(i)* a physical topology, i.e., an undirected graph; and *(ii)* a set of configuration features to be tested, such as, route-maps, and route-summarization. For every discovered bug, Metha creates a report, which consists of the identified discrepancy between the routing tables of the tool and the oracle, the configuration statements causing it and a configuration set to reproduce it.

Phase I: Test coordination (§4, §5) The configuration features and the topology provided as input define the search space of Metha's testing efforts which consists of all possible configurations that can be built using these features.

This search space of network configurations is prohibitively large. Therefore, Metha first reduces the values of all parameters to their boundary values, which means it only uses the two extreme values (i.e., the minimum and the maximum) and one "normal" value. Even with this reduction, it is difficult to systematically cover the search space. Hence, Metha creates a test suite relying on combinatorial testing, which allows it to cover all pairs of feature and parameter combinations requiring a minimal number of tests. Each test in the test suite is a set of configuration statements that should be active.

Phase I: Testbed (§7) For every single test, Metha generates the device configurations based on the statements as defined by the test suite. Then, it runs these configurations in the tool and the oracle. Once both converged, Metha analyzes the routing tables of the two tools and reports any discrepancies.

BGPProcess	→	`router bgp` *Integer16* [*Options*]
Options	→	*Option* | *Options Option*
Option	→	*Redistribute* | *Neighbor* | *Network* | ···
Redistribute	→	`redistribute` *Source*
Source	→	`direct` | `static` | ···
Neighbor	→	`neighbor` *Address Property*
Property	→	*RemoteAS* | *RouteMap* | ···
RemoteAS	→	`remote-as` *Integer16*
RouteMap	→	`route-map` *String Direction*
Direction	→	`in` | `out`

Figure 4: Partial BNF grammar for device configurations.

Phase II: Fault localization (§6) A discovered discrepancy can be caused by multiple bugs in the network analyzer. Therefore, Metha applies delta debugging to identify every single bug and the configuration statements causing it. It does so by iteratively testing subsets of the active configuration statements until the entire discrepancy is resolved.

4 Search Space

In this section, we define the search space of all possible configurations. We also show how we reduce the search space by restricting the parameter values used in configuration statements to *boundary* values.

4.1 Network Configurations

The search space is given by all possible configurations that one can deploy at the network's routers.

Configurations A device configuration defines the enabled features along with their parameter values. Formally, the set of all possible configurations is defined by a context-free grammar whose terminals consist of feature names and parameter values. To illustrate this, in Fig. 4 we show a subset of the production rules in Backus-Naur form (BNF). An example configuration derived from this grammar is:

```
router bgp 100
  redistribute static
  neighbor 1.1.1.2 remote-as 50
  neighbor 1.1.1.2 route-map map10 out
```

This configuration defines the AS identifier, neighbors, neighbor properties, and route redistribution associated with the BGP routing process `100`. Here, `router bgp`, `redistribute`, `neighbor A.B.C.D remote-as`, and `neighbor A.B.C.D route-map` are configuration statements, while the values to the right define their parameter values. We distinguish three types of parameter values:

Keywords are used in configuration statements parameterized by a value drawn from a fixed set of options. For example, the configuration statement `neighbor A.B.C.D route-map` is parameterized by a direction, which is set to either `in` or `out`. For some statements, one can also omit the parameter value altogether, which we model with the designated value ∅. For example, `redistribute connected` is parametrized by a value drawn from the set {∅,`subnets`}, and so both `redistribute connected` and `redistribute connected subnets` are valid statements.

Integers are used to define 16- and 32-bit numbers. For example, the configuration statement `router bgp` is parameterized by a 16-bit integer defining the AS number.

Strings are used in configuration statements parameterized by custom names. For example, `neighbor A.B.C.D route-map` is parameterized by the route-map's name.

Semantic constraints Besides conforming to the syntax in Fig. 4, configurations must also comply with semantic constraints. For example, consider the following configurations:

```
1    interface FastEthernet0/0
2      ip address 1.1.1.1 255.255.255.0
3    !
4    router bgp 100
5      neighbor 1.1.1.2 remote-as 50
6      neighbor 1.1.1.2 route-map map10 out
7    !
8    route-map map10 permit 10
9      match ip address prefixList
```

```
1    interface FastEthernet0/0
2      ip address 1.1.1.2 255.255.255.0
3    !
4    router bgp 50
5      neighbor 1.1.1.1 remote-as 100
```

The top configuration (C_1) defines a BGP process with AS number `100` (Line 4), and declares that announcements sent to its BGP neighbor with IP `1.1.1.2` (Line 5) are processed using route-map `map10` (Line 6). The bottom configuration (C_2) defines a BGP process with AS number 50 (Line 4), and declares `1.1.1.1` in AS `100` as a neighbor (Line 5). These two configurations illustrate two kinds of semantic constraints:

Intra-device constraints, which stipulate conditions that must hold on any (individual) configuration. For example, the route-map `map10` used at Line 6 must be defined within the configuration C_1. This constraint holds as `map10` is defined at Line 8.

Inter-device constraints, which stipulate conditions across multiple configurations. For example, the AS number assigned to neighbor `1.1.1.2` in C_1 at Line 5 must match the AS number declared in C_2 at Line 4. This constraint holds as at both lines the AS number is 50.

Finally, we note that we specify the semantic constraints separately from the syntactic production rules as some are not context-free and thus cannot be encoded in the grammar.

Search space The search space used by Metha is defined by the set of configurations that one can deploy at the network's routers. As the set of configurations derived from the grammar is, in general, infinite, we restrict all recursive rules so that its language consists of finitely many configurations. For instance, for the grammar given in Fig. 4, we fix the set of BGP options (such as route redistribution) that can appear when defining a BGP routing process. Finally, the search space of Metha is defined as C^R, where C is the set of all configurations and R is the set of routers. Note that each element of C^R defines a *network-wide configuration*, assigning a configuration from C to each router in R.

4.2 Boundary Values

The search space is extremely large due to the enormous number of configurations and the exponentially many combinations in which they can be deployed at the routers. To cope with the large set of configurations, we apply a *boundary values* reduction by restricting the parameters to a small set of representative values. The intuition behind this reduction is that most parameter values lead to the same behavior such that testing them individually provides no additional insights.

The reduction to boundary values ensures that various behaviors of a feature are exercised. For example, the Cisco BGP feature `neighbor X.X.X.X maximum-prefix n` terminates the session when the neighbor announces more than `n` prefixes. When randomly choosing `n`, the feature will most likely not come into action. However, with the boundary values, both the minimum and maximum value are tested, ensuring that the feature is at least once active and once not.

For integer parameters, the values are restricted to: the maximum value, the minimum value, and a non-boundary value. For example, for 16-bit integers, which contain all integers in the range $[0,65535]$, our boundary value reduction selects three values: 0, 65535, and a value x such that $0 < x < 65535$. Similarly, we reduce the values assigned to string parameters by predefining a fixed set of strings.

5 Effective Search Space Exploration

Metha must cover a wide variety of different network configurations to thoroughly test the tool, including many combinations of device features and parameter values. The key challenge is that it is impossible to iterate through every single combination of features and their respective parameter values, even after considering our reduction to boundary values. To address this, Metha relies on *combinatorial testing* [16, 20], which is able to uncover all bugs involving a small number of interacting features. In the following, we first provide relevant background on combinatorial testing, and then we show how Metha uses it to effectively test network tools.

5.1 Combinatorial Testing

Combinatorial testing is a black-box test generation technique which is effective at uncovering *interaction bugs*, i.e., bugs that occur because of multiple interacting features and their parameter values. The main assumption behind combinatorial testing is that interaction bugs are revealed by considering a small number of features and parameter values. In this case, one can generate a test suite, called *combinatorial* test suite, that uncovers all such bugs.

To use combinatorial testing, one needs to define a specification of the system's parameters and their values:

Definition 1 (Combinatorial specification)**.** A combinatorial specification $\mathcal{S}$ is a tuple (P,V,Δ), where P is a set of parameters, V is a set of values, and $\Delta\colon P \to 2^V$ defines the domain of values $\Delta(x) \subseteq V$ for any parameter $x \in P$.

For example, the combinatorial specification for a program that accepts three boolean flags as input has parameters $P = \{a,b,c\}$, values $V = \{0,1\}$, and domains $\Delta(a) = \Delta(b) = \Delta(c) = \{0,1\}$. A *test case* is a total function $tc\colon P \to V$ mapping parameters to values from their respective domains, i.e., with $P(x) \in \Delta(x)$ for any x. An example test case for our program is $tc = \{a \mapsto 0, b \mapsto 0, c \mapsto 1\}$. In contrast to test cases, a *t-wise combination* maps only some parameters to values:

Definition 2 (t-wise combination)**.** Given a combinatorial specification $\mathcal{S} = (P,V,\Delta)$, a t-wise combination for $\mathcal{S}$ is a function $c\colon Q \to V$ such that $Q \subseteq P$ with $|Q| = t$ and $c(x) \in \Delta(x)$ for any $x \in P$.

An example pairwise combination (i.e., $t = 2$) for our example is $c = \{a \mapsto 0, b \mapsto 1\}$. We write $C_t^{\mathcal{S}}$ to denote the set of all t-wise combinations for a given combinatorial specification $\mathcal{S}$. Note that a test case can cover multiple t-wise combinations:

$$comb_t(tc) = \{c \subseteq tc \mid |c| = t\}$$

For instance, our example test case above covers the following three pairwise combinations: $\{a \mapsto 0, b \mapsto 0\}$, $\{a \mapsto 0, c \mapsto 1\}$, and $\{b \mapsto 0, c \mapsto 1\}$.

Definition 3 (t-wise combinatorial coverage)**.** Given a combinatorial specification $\mathcal{S}$, we define the t-wise combinatorial coverage of a test suite T as:

$$cov_t(T) = \frac{|\bigcup_{tc \in T} comb_t(tc)|}{|C_t^{\mathcal{S}}|}\,.$$

A test suite T is called a *t-combinatorial test suite* if $cov_t(T) = 1$. If the assumption that interaction faults are caused by up to t-wise interactions holds, then T finds all bugs. The goal of combinatorial testing is to generate the smallest t-wise combinatorial test suite.

5.2 Combinatorial Testing of Configurations

In Metha, we apply pairwise combinatorial testing to the generation of network configurations. Concretely, we phrase the search space defined in §4 as a combinatorial specification $\mathcal{S} = (P, V, \Delta)$ as follows. First, each statement that can appear in the configuration, such as route redistribution or route-map as defined in §4, defines a configuration feature. We set F to be the set of all configuration features. The set of parameters P is then given by $R \times F$, where R is the set of routers. Namely, the parameters consist of all configuration features one can define in the device configurations.

Second, the domains of values for each configuration feature contain the boundary values that can be used in the given configuration statement, along with the designated value $\perp$, which indicates whether the configuration feature is enabled or not. That is, $\perp$ results in omitting the configuration statement altogether. We note that for configuration statements with multiple parameters, we take the product as the domain of possible values. For example, the Cisco OSPF configuration feature `default-information-originate` has three optional parameters: `always`, `metric` combined with an integer value, and `metric-type` combined with 1 or 2. After reduction to boundary values this leads to the following three parameters:

$$A = \{\varnothing, \texttt{always}\}$$
$$B = \{\varnothing, \texttt{metric 1}, \texttt{metric 100}, \texttt{metric 1677214}\}$$
$$C = \{\varnothing, \texttt{metric-type 1}, \texttt{metric-type 2}\}$$

The domain of values for this configuration feature is then given by $\{\perp\} \cup (A \times B \times C)$.

Finally, Metha uses the above combinatorial specification to derive a test suite of configurations that covers all pairwise combinations.

6 Fault Localization

A discovered discrepancy between the network model and the oracle is only of limited use as the developer still needs to isolate its cause. Often understanding the bug is the most time-consuming part of the debugging process, and fixing it can be done relatively quickly. To help with this, Metha pinpoints the configuration features that cause a discrepancy and finds a minimal configuration, i.e., a configuration with as few configuration features enabled as possible. To do this, Metha uses *iterative delta debugging*, an extended version of classic delta debugging, which lifts the assumption that a single fault causes failures. This extension is important as network configurations are large and complex, and discrepancies are often caused by multiple faults. In the following, we first introduce classic delta debugging and then present its iterative extension.

6.1 Delta Debugging

Delta debugging [35] is a well-established fault localization technique, which finds minimal failure-inducing inputs from failing test cases. Below, we present delta debugging in our context, and then define its assumptions and algorithmic steps.

Terminology As defined in §5, a test case tc assigns configuration features F to either parameter values or $\perp$, where $\perp$ indicates that a given feature is disabled (i.e., it is omitted from the configuration). Given a test case tc and features $Q \subseteq F$, we write $tc|_Q$ for the test case obtained by disabling all features in tc that are not contained in Q:

$$tc|_Q(f) = \begin{cases} tc(f) & \text{if } f \in Q \\ \perp & \text{otherwise} \end{cases}$$

Given a failing test case tc, the goal of delta debugging is to find the minimal set Q of features such that $tc|_Q$ fails. We denote the complement of Q by $\bar{Q} = F \setminus Q$.

Assumptions Delta debugging relies on three assumptions: *(i)* test cases are *monotone*, i.e., if $tc|_Q$ fails, then for any superset $Q' \supseteq Q$ of features $tc_{Q'}$ also fails; *(ii)* test cases are *unambiguous*, meaning that for a failing test case tc there is a unique minimal set Q that causes the failure; and *(iii)* every subset of features is *consistent*, meaning that for any $Q \subseteq F$, $tc|_Q$ terminates with a definite fail or success result.

Algorithm Given a test case tc, delta debugging finds a minimal set of features Q that causes a failure. Initially, Q contains all enabled features in tc, i.e., $Q = \{f \in F \mid tc(f) \neq \perp\}$. Then it applies the following steps:

1. *Split:* Split Q into n partitions $Q_1, \ldots, Q_n$, where n is the current granularity. Test $tc|_{Q_1}, \ldots, tc|_{Q_n}$ for failures. If some $tc|_{Q_i}$ fails, then use Q_i as the new current set of features and continue with step 1.

2. *Complement:* If none of the new tests $tc|_{Q_1}, \ldots, tc|_{Q_n}$ fail, check the complement of each partition by testing $tc|_{\bar{Q}_1}, \ldots, tc|_{\bar{Q}_n}$. If some $tc|_{\bar{Q}_i}$ fails, then use $\bar{Q}_i$ as the new current set of features and continue with step 1.

3. *Increase Granularity:* If no smaller set of features is found and $n < |Q|$, then set n to $\min(2n, |Q|)$ and continue with step 1.

4. *Terminate:* If it is not possible to split the current set of features into a smaller set, terminate and return Q.

6.2 Iterative Delta Debugging

In our setting, test cases are often ambiguous as a discrepancy often arises due to multiple faults in the network model. To this end, we apply the delta debugging algorithm *iteratively* and find all minimal sets of features that cause a given discrepancy. Intuitively, starting with a test case tc with enabled features Q, we first apply the delta debugging steps (given

Algorithm 1: Iterative Delta Debugging

```
Input   : Test case $tc$, initially enabled features $Q$ in $tc$.
Output  : A set of minimal feature subsets $\mathcal{S} = \{Q_1, \ldots, Q_n\}$.
1  $\mathcal{S} = \emptyset$
2  $Queue = \texttt{queue()}$
3  put($Queue, Q$)
4  while $\neg$empty($Queue$) do
5      $H = \texttt{head}(Queue)$
6      if $run(tc|_H) = \textit{failure}$ then
7          $Q' = minimize(H)$
8          for $f$ in $Q'$ do
9              put($Queue, H \setminus \{f\}$)
10         $\mathcal{S} = \mathcal{S} \cup \{Q'\}$
11 return $\mathcal{S}$
```

in §6.1) to find a minimal configuration feature set Q' such that $tc|_{Q'}$ triggers the discrepancy. Then, we generate new test cases $tc_1, \ldots, tc_{|Q'|}$, by disabling a feature from Q in each new test case tc_i, and iteratively apply delta debugging to these. We apply this process repeatedly until no further failing test cases are found. Once Metha identifies all minimal sets of configuration features that trigger a given bug, Metha creates a minimal configuration for the developer to reproduce it.

We present our iterative delta debugging algorithm in Algorithm 1. We start from a set of initially enabled features Q in tc and return all minimal subsets of Q that trigger a discrepancy. We keep all sets of features to be checked in a queue and continue until the queue is empty (Line 2 - Line 4). For every set H of features in the queue, we check if the test case $tc|_H$ triggers a discrepancy (Line 5, Line 6). If this is the case, then we find a minimal subset $Q \subseteq H$ of features that triggers the discrepancy using classic delta debugging, and create new subsets that need to be checked (Line 8, Line 9). For example, if we find a minimal set of features $Q = \{a, b\}$ that triggers the discrepancy, then we check if there are any other minimal sets of features that do not contain a or b (and are thus non-comparable to Q). We note that we generate two new sets of features $H \setminus \{a\}$ and $H \setminus \{b\}$ instead of a single one $H \setminus \{a, b\}$ because there may be overlapping discrepancies. For example, even though we know that b can trigger a discrepancy with a, b might also trigger a discrepancy with another feature c. Finally, the algorithm keeps all found minimal feature subsets and returns them (Line 10, Line 11). We conclude by stating the correctness of our algoirthm:

Theorem 1. *For any test case tc with enabled features Q, Algorithm 1 finds all minimal fault-inducing subsets of features.*

We present the proof of this theorem in App. A.

Runtime The running time of Algorithm 1 is $O(|Q|!)$. The worst-case behavior is when the size of the set H of features is reduced by 1 element in each step, introducing $|H - 1|$ new features sets to the set $\mathcal{S}$. To improve the running time, we cache (not shown in Algorithm 1) feature sets that have been added to the queue. This strictly reduces the algorithm's running time and yields a worst-case running time complexity of $O(2^{|Q|})$. We note that the running time in practice is reasonable as the reduction of the set H by the delta debugging minimization step (Line 7) is significant (down to $2-3$ elements in practice).

Limitations As with classic delta debugging, there may be a fault in the interaction between a set of parameters, say a, b, and c, as well as a different fault in the interaction between a subset of these parameters, say a and b. We cannot distinguish these two faults and will only identify the latter fault. However, once the identified fault is fixed, our algorithm will then identify the fault in the interaction among a, b, and c as well, assuming it is still present in the verification tool.

7 System

We have fully implemented Metha in 7k lines of Python code.[2] This covers the entire testing pipeline from the input, the list of configuration features to be tested and the topology, to the outputs, the bug reports. In the following, we highlight key points of Metha's implementation, which consists of a vendor- and tool-agnostic core that uses runners to interface with the different network analysis and verification tools.

Semantic constraints To run the tests, Metha uses a logical topology, which consists of the physical topology extended with logical groupings. These groupings map the routers to BGP ASes and their interfaces to OSPF areas. This trivially ensures that the base configuration meets all the necessary semantic constraints (cf. §4.1). In a next step, Metha starts to randomly assign IP subnets to links and IP addresses to the router interfaces on these links. Specifically, every router is assigned a router ID, which is also assigned to the loopback interface of that router. Finally, Metha generates additional resources that are needed to test specific configuration features. For example, Metha adds several prefix-lists and static routes which can then be used in the test generation, for example, for a match statement of a route-map and route redistribution, respectively. All these additional resources are generated based on the predefined logical topology. Hence, a prefix-list, for example, will only consist of prefixes that are actually defined in the network, such that a route-map statement using that list for a match will also be reachable.

Testing coordination Once Metha laid the groundwork, it has to define a test strategy based on the specified configuration features. At the moment, Metha supports configuration features pertaining to four categories: static routes, OSPF, BGP and route-maps. As part of that, the system supports additional constructs such as prefix-lists and community-lists. These are currently not tested on their own, but added when needed to test the main features, such as route-maps. Metha

[2] Available at https://github.com/nsg-ethz/Metha

then uses all features and the logical topology to prepare the parameters to come up with the test suite. To do that, Metha passes all the parameters and their possible values to a state-of-the-art combinatorial testing tool: PICT [25]. PICT devises a test suite that consists of a set of tests ensuring complete coverage of all pairwise feature interactions.

Configuration generation A single test from the PICT test suite is an abstract network configuration. It simply specifies which feature and corresponding value needs to be activated and where (i.e., on which router and, if applicable, at which interface). Metha then translates the abstract network configuration to concrete device configurations using a grammar-based approach to ensure lexical and syntactical validity.

Metha implements a large portion of both Cisco IOS and Juniper grammars for which we relied on the respective official command references. This means Metha can generate both Cisco IOS and Juniper configurations for the tests. Metha even supports to test hybrid networks in which devices of both vendors are used at the same time.

Testbed Metha runs the generated configurations in parallel on both the tool under test and the oracle. After both of them converged, it retrieves the routing tables and compares them. Metha is able to test any tool that takes the device configurations as inputs and provides direct access to the computed routing tables out-of-the-box. Otherwise, Metha uses tool-specific runners to process the inputs such that they meet the tool's requirements and map the output back to Metha's format. Metha comes with runners for three well-known network analysis and verification tools: Batfish [17], NV [13] and C-BGP [28]. For NV, for example, Metha first has to compile the simulation program from the network configurations. As a source of ground truth, Metha uses a virtualized network running real device images of both Cisco and Juniper routers. It connects to these devices over Telnet and retrieves the routing tables (e.g., `show ip route` for Cisco devices). To ensure full convergence, Metha retrieves the routing tables every 10 seconds and proceeds once the tables have not changed for ten consecutive checks. With this setup, Metha allows to freely choose any oracle (e.g., hardware testbed) as long as it exposes the computed routing tables.

Output Finally, Metha localizes all bugs within a discovered discrepancy by relying on delta debugging. For every single bug, it generates a report highlighting the observed difference in the routing tables of the tool under test and the oracle, such as a mismatch in a route's metric or a missing route. This helps the developer understand the expected behavior. In addition, it identifies the configuration statements required to trigger the bug and comes up with a minimal network configuration to reproduce the bug. This allows Metha to provide actionable feedback to the developers of the tool, helping them to faster locate and understand the bug. The minimal configuration example can also be used as an extra test case for traditional system testing.

8 Evaluation

In this section, we evaluate Metha to address the following research questions:

RQ1 How does Metha's semantical configuration generation, the search space reduction using boundary values and the test suite from combinatorial testing contribute to Metha's effectiveness? We show that Metha finds 20 bugs and achieves a higher combinatorial coverage than the random baseline, which only discovers 3 bugs with the same number of tests.

RQ2 How many test cases does Metha need to localize all bugs in a single discrepancy between the tool under test and the oracle? Metha requires on average 14.1 test cases to isolate all the bugs causing a discrepancy.

RQ3 Is Metha practical? We ran Metha on three different state-of-the-art network analysis and verification tools and found a total of 62 bugs, 59 of them have been confirmed by the respective developers.

8.1 Comparison to Random Baseline

We begin our evaluation by studying how the three components of Metha contribute to its effectiveness. To this end, we compare a random baseline to three versions of Metha: step-by-step, we enable each component starting with semantic Metha, then we add the reduction to boundary values, and finally, we use full Metha using combinatorial testing to define a test suite. The results show that the semantical configuration generation is the most fundamental part of Metha. Reducing the parameters to boundary values and applying combinatorial testing help to find additional bugs as both manage to increase the combinatorial coverage.

In the following, we introduce the four approaches:

Random baseline The random baseline relies on random syntactic test generation, meaning it uses a traditional grammar-based fuzzing approach. Thanks to the grammar, the configurations generated by the baseline are lexically- and syntactically-valid, but they are not necessarily semantically-valid: the baseline generates device configurations that are parseable and look realistic. However, the configurations might not always be practical: for example, referenced route-maps and prefix-lists do not always exist, and IP addresses on interfaces might not match those of their neighbors. Inter- and intra-device dependencies are not factored in.

Semantic Metha The initial Metha approach implements random semantic test generation. Similar to the random baseline, it uses a grammar-based fuzzing approach with the only difference that it ensures semantical validity within the configuration: while, for example, interface costs are completely random, other values are more constrained based on inter- and intra-device dependencies. This approach ensures, for example, that only defined route-maps are referenced, and that BGP sessions are configured with matching parameters.

Approach	# Discovered Bugs
Random Baseline	3
Semantic Metha	16
Bounded Metha	17
Full Metha	**20**

Table 2: Every component of Metha allows it to find more bugs with the same number of test runs.

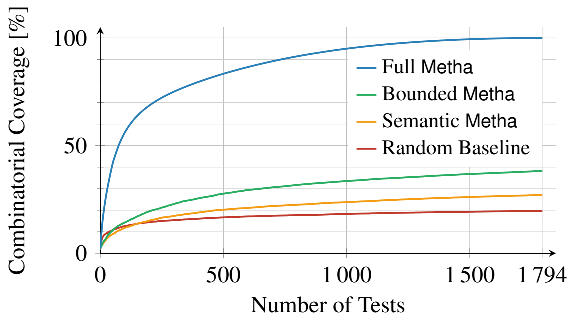


Figure 5: The achieved combinatorial coverage increases with every single component of Metha. Full Metha achieves complete combinatorial coverage.

Bounded Metha The bounded approach adds the reduction to boundary values as introduced in §4.2 to semantic Metha. This means instead of assigning completely random numeric values, the approach reduces the allowed values to three options: the minimum, the maximum, and a "normal" value, randomly chosen between the two extremes.

Full Metha Finally, we run the full testing system. We add combinatorial testing as introduced in §5 to define a test suite that maximizes combinatorial coverage on top of the semantic configuration generation and the boundary values reduction.

Experiment setup We ran all four approaches for the same number of tests and used them to test Batfish [17]. Whenever one of them detected a discrepancy between Batfish and the oracle, we applied the full fault localization procedure as described in §6 to detect the underlying bugs and the features causing it. Thanks to that, we are able to detect duplicates and count only the unique bugs that each approach discovered.

For all the tests, we used the same simple topology consisting of four routers connected in a star topology and tested configuration features belonging to the following four categories: static routes, BGP, OSPF, and route-maps. For the entire experiment, we used Cisco IOS configurations. For the given configuration features, combinatorial testing generated a test suite consisting of 1 794 tests. While the full Metha approach followed the test suite, the other approaches randomly chose the active configuration statements for every single test.

Results Table 2 shows the number of unique bugs that every approach found within the 1 794 test runs. The full Metha detected 20 unique bugs, while the random baseline only found 3 bugs. The semantic configuration generation is the most fundamental component of Metha. It comes as no surprise as without semantical validity, many of the configurations do not allow for any meaningful control plane computations and will not fully exercise the network model of the tool under test.

Boundary values and combinatorial testing allow finding 1 and 3 additional bugs within the 1 794 test runs, respectively. This is because both approaches achieve higher combinatorial coverage and therefore test a wider variety of features. These results show that the boundary values reduction strikes a good balance between testing different parameter values, while keeping the search space tractable. It is important to note that the detected bugs are inclusive, meaning that full Metha detected all 17 bugs that bounded Metha detected and 3 additional bugs. There is one exception: the baseline found a bug in the parser, which the other approaches did not find.

The random baseline is strong at discovering parser bugs since that is where grammar-based fuzzing excels. Two out of its three discovered bugs are parser bugs. In both cases, the problem was an, according to the specification, unsigned 32-bit integer being parsed as signed. For example, `ip ospf 100 area 3933914791` could not be parsed. Metha did not catch this bug as it uses fixed area numbers as part of the logical topology. By adding the area numbers to the set of configuration features being tested, Metha also finds this bug.

Fig. 5 shows the combinatorial coverage achieved by the four approaches, i.e., it shows the pairwise feature combinations covered during testing. We focus on feature instead of code coverage for two reasons: First, one can easily achieve high code coverage with random, semantically-invalid configurations. Second, code coverage is specific to the tool under test and makes it difficult to compare. To measure the combinatorial coverage of the random baseline and semantic Metha, we partitioned the input space in the same manner as we did for bounded Metha, i.e., into minimum, maximum, and middle values. Any configuration which did not specifically use the minimum or maximum value for a parameter was then considered as a middle configuration. Metha achieves full combinatorial coverage by design as it is guaranteed by combinatorial testing. These results underline the importance of semantically-valid configurations. While both the random baseline, which relies on syntactically-valid configurations, and semantic Metha achieve a similar combinatorial coverage, semantic Metha finds many more bugs as its configuration actually ensures control plane computations.

Performance Running a single test case took an average of two minutes. We run both the tool under test as well as the virtualized testbed in parallel and found that most of the time is spent waiting on the testbed to converge. The generation of a combinatorial test suite with PICT for the baseline network with 4 routers took an average of 6 minutes. Over the entire test suite, this time is negligible. Running the entire setup took us several days. The runtime depends highly on the number of discrepancies and the number of bugs causing them.

	Bugs		Type		Feature Category			
	discovered	confirmed	crash	silent	OSPF	BGP	route-filter	other
Batfish [17]	29	29	5	24	10	10	9	0
NV [13]	30	30	5	25	13	9	7	1
C-BGP [28]	3	?	0	3	1	1	1	0

Table 3: Bugs discovered by Metha for Batfish, NV and C-BGP and classification.

8.2 Fault Localization

Whenever Metha detects a discrepancy between the routing tables of the tool under test and those of the oracle, it goes into fault localization to isolate all independent bugs. Fault localization relies on delta debugging (cf. §6) which creates additional test cases to identify the configuration statements causing the bugs. In the following, we evaluate its overhead, i.e., the number of additional test cases Metha had to create.

Experiment setup For this experiment, we ran Metha using the same topology as before and tested the full set of configuration features. Whenever Metha detected a discrepancy, we recorded the number of additional test cases required to find all independent bugs and the number of discovered bugs.

Results On average, Metha used 14.1 additional test cases to locate all bugs within a test case. The number of additional test cases ranged from as low as 7, to localize a single bug, up to as high as 58, to localize 5 independent bugs. The number of additional test cases mostly depends on the number of independent bugs within a single detected discrepancy. The number of configuration statements that actually cause the bug plays a minor role. Also, we have observed that the detected bugs are all caused by a few configuration statements (one or two), even though multiple configuration statements were active during the tests. This confirms the observation that bugs are often caused by the interaction of few features [20, 31] and shows that combinatorial testing is a useful technique in this setting.

8.3 Real Bugs

In addition, we showcase our end-to-end implementation of Metha by testing three different network analysis and verification tools: Batfish [17], NV [13], and C-BGP [28]. We show that Metha finds real bugs and report them in Table 3.

Experiment setup We ran Metha for several days on all three tools and with several different setups. Batfish is the most complete and advanced tool as it can handle configurations of many different vendors and supports a wide variety of configuration features. NV itself is an intermediate language for control plane verification that allows to build models of any routing protocols and their configurations. It provides simulation and verification abilities. We tested the simulation only, the discovered bugs, however, most likely also exist in the verification part as both rely on the same network model. For Batfish and NV, we used both Cisco IOS and Juniper configurations. C-BGP has its own configuration language.

Results As shown in Table 3, Metha found a total of 62 bugs. The developers of both Batfish and NV confirmed the discovered bugs to be real bugs. To better understand the nature of the bugs, we classified them by their type (i.e., whether they lead to a crash or go unnoticed) and by the configuration feature category itself (e.g., OSPF). Only a few of the bugs produce a clear error. This is most likely also because these are noticed more often and reported. The large majority of the bugs are silent semantic bugs which are extremely difficult to notice. These are the sneakiest bugs and can lead to false analyses and answers by the verifier. These bugs include all the configuration features discussed in §2 showing that they affect the analysis of commonly used features, such as route redistribution and aggregation, and named communities.

The bugs are distributed quite evenly among all tested parts of the network model. We did not find one specific protocol or configuration feature that is especially error-prone.

9 Discussion

What about the testbed? Metha detects bugs by looking for discrepancies between the tool under test and an oracle. For the oracle, Metha uses a testbed running real router firmwares. The testbed just needs to be large enough to fully exercise all configuration features. Normally, a small testbed of few routers suffices and also helps speed up the testing. In this paper, we rely on a virtualized testbed. To use a physical testbed instead, one simply has to change the SSH/Telnet configurations to connect to the physical devices.

A virtualized testbed comes with several advantages. It provides more flexibility in terms of the settings one can test and the time needed to setup. For example, there is no re-wiring needed to test different topologies. In addition, it is very simple to test the same topology with a different device category or with devices from another vendor: one simply has to exchange the router image.

What about more targeted tests? Metha's test suite can be adjusted to the developers requirements by restricting the set of configuration features, adjusting the number of values per feature, and changing the number of interacting features. The tests required to cover the search space mainly depend on the number of values per feature and the number of simultaneous feature interactions, while the set of features is secondary. By default Metha tests three values per feature and considers pairwise interactions. This choice strikes a good balance between the number of tests required and thorough testing, as our results confirm: Metha found all bugs that the random approaches discovered with fewer tests, despite using "only" the boundary values; and all discovered bugs are caused by one or two interacting configuration features, despite considering interactions of more than just two features.

Metha does not replace traditional unit and system testing, but provides an additional way to find latent bugs anywhere in the system. The advantage of Metha is that it requires minimal developer involvement and can be run alongside traditional tests without any additional effort. If desired one can run extensive tests by considering more elaborate feature interactions and more than three values per feature. Often with fuzz testing, one just lets the testing system run indefinitely and collect bug reports along the way.

10 Related Work

In this section, we first discuss current network analysis and verification tools. Then, we survey related work on testing static analyzers and verifiers, the various testing initiatives in the field of networking, delta debugging, and fuzz testing.

Network analyzers & verifiers Our work aims to facilitate the development of network analysis and verification tools through thorough testing. Over the years, we have seen a rise in tools that simulate networks [28], verify properties of networks and their configurations [3, 14, 19, 30], and tools that analyze aspects of networks [11, 12, 18, 23]. All of these tools have in common that they in some way or another use a network model to analyze and verify the network. Any bug or inaccuracy that exists within that network model undermines the soundness of the tools' results and analyses.

In contrast, CrystalNet [21] is a cloud-scale, high-fideltiy network emulator running real network device firmwares instead of relying on a network model. Hence, it accurately resembles the real network (e.g., vendor-specific behaviors and bugs in device firmwares are captured).

Testing analyzers and verifiers The problem of ensuring the correctness of analysis and verification tools is not specific to networks. In the field of static analysis, several works exist that pursue the same goal. Bugariu et al. [5] apply a unit testing approach, meaning they do not test the entire system but components thereof which simplifies the test generation. Since Metha treats the tool under test as a black box it cannot test certain components separately. Cuoq et al. [8] randomly generate input programs. This technique is mostly effective at testing the robustness of the analyzers. Similar to Metha, Andreasen et. al [1] apply delta debugging to find small input programs that help developers understand the bug faster.

Testing in networking Prior work on testing in networking has mainly focused on testing the network and its forwarding state [36], and SDN controllers [2, 6, 29].

Closest to Metha is Hoyan [32], a large-scale configuration verifier, in which the results of the verifier (i.e., network model) are continuously compared to the actual network for inaccuracies. It does so during operation and only covers cases that have actually occurred in the network. Metha in contrast proactively tests to detect the bugs before deployment.

Delta debugging In automated testing tools, delta debugging is a well-established technique [33, 35] that allows to automatically reduce a failing test case to the relevant circumstances (e.g., lines of code or input parameters). Over the years, researchers came up with several extensions to the general delta debugging algorithm, such as a hierarchical approach [26] that takes the structure of the inputs into account. It first explores the more important inputs allowing to prune larger parts of the input space and hence, requiring fewer test cases.

Traditional delta debugging finds one bug at a time even if the test case is ambiguous and exhibits multiple independent bugs. The developer then fixes one bug and reruns delta debugging to find the next. Metha automatically detects the causes of all independent bugs without developer involvement.

Fuzz testing Fuzz testing [15, 34] is an umbrella term for various testing techniques relying on "randomized" input generation. Metha uses a form of grammar-based fuzzing. Due to the complex dependencies within network-wide configurations, Metha first builds a basic configuration structure to ensure semantical validity. Then, it uses fuzzing to test different feature combinations restricted to that structure.

11 Conclusion

We presented Metha, an automated testing framework for network analysis and verification tools that discovers the bugs in their network models before deploying them to production. It does so by generating a wide variety of network configurations according to a test suite defined through combinatorial testing. Metha provides developers with actionable reports about all discovered bugs including a configuration to reproduce them. We implemented Metha and evaluated it on three state-of-the-art tools. In all tools, Metha discovered a total of 62 bugs, 59 of them have been confirmed by the developers. An interesting avenue for future work would be to extend Metha so that it can also test configuration synthesizers such as [4, 9, 10] as bugs in their models would render them useless.

Acknowledgements

We thank our shepherd Michael Schapira and the anonymous reviewers for their insightful comments and helpful feedback. The research leading to these results was partially supported by an ERC Starting Grant (SyNET) 851809.

References

[1] Esben Sparre Andreasen, Anders Møller, and Benjamin Barslev Nielsen. Systematic Approaches for Increasing Soundness and Precision of Static Analyzers. In *ACM SOAP*, Barcelona, Spain, 2017.

[2] Thomas Ball, Nikolaj Bjørner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Mooly Sagiv, Michael Schapira, and Asaf Valadarsky. VeriCon: Towards Verifying Controller Programs in Software-defined Networks. In *ACM PLDI*, Edinburgh, United Kingdom, 2014.

[3] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A General Approach to Network Configuration Verification. In *ACM SIGCOMM*, Los Angeles, CA, USA, 2017.

[4] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Don't Mind the Gap: Bridging Network-Wide Objectives and Device-Level Configurations. In *ACM SIGCOMM*, Florianopolis, Brazil, 2016.

[5] Alexandra Bugariu, Valentin Wüstholz, Maria Christakis, and Peter Müller. Automatically Testing Implementations of Numerical Abstract Domains. In *ACM ASE*, Montpellier, France, 2018.

[6] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. A NICE Way to Test OpenFlow Applications. In *USENIX NSDI*, San Jose, CA, USA, 2012.

[7] Inc. Cisco Systems. Redistributing Routing Protocols. https://www.cisco.com/c/en/us/support/docs/ip/enhanced-interior-gateway-routing-protocol-eigrp/8606-redist.html, March 2012. Accessed: 2020-09-12.

[8] Pascal Cuoq, Benjamin Monate, Anne Pacalet, Virgile Prevosto, John Regehr, Boris Yakobowski, and Xuejun Yang. Testing Static Analyzers with Randomly Generated Programs. In *NFM*, Norfolk, VA, USA, 2012.

[9] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. Network-wide Configuration Synthesis. In *CAV*, Heidelberg, Germany, 2017.

[10] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. Netcomplete: Practical Network-Wide Configuration Synthesis with Autocompletion. In *USENIX NSDI*, Renton, WA, USA, 2018.

[11] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd D Millstein. A General Approach to Network Configuration Analysis. In *USENIX NSDI*, Oakland, CA, USA, 2015.

[12] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. Fast Control Plane Analysis using an Abstract Representation. In *ACM SIGCOMM*, Florianopolis, Brazil, 2016.

[13] Nick Giannarakis. NV - An Intermediate Language for Network Verification. https://github.com/NetworkVerification/nv, 2020. Commit: d058c4ce5c1549ad4e22d97cb01b8ea19d07741c.

[14] Nick Giannarakis, Devon Loehr, Ryan Beckett, and David Walker. NV: An Intermediate Language for Verification of Network Control Planes. In *ACM PLDI*, London, UK, 2020.

[15] Patrice Godefroid. Fuzzing: Hack, Art, and Science. *Communications of the ACM*, 63(2), 2020.

[16] Linghuan Hu, W Eric Wong, D Richard Kuhn, and Raghu N Kacker. How does combinatorial testing perform in the real world: an empirical study. *Empirical Software Engineering*, 25(4), 2020.

[17] Intentionet. Batfish. https://github.com/batfish/batfish, 2020. Commit: 95099bc5ad77af57d92c484e2e5634827f63e724.

[18] Peyman Kazemian, George Varghese, and Nick McKeown. Header Space Analysis: Static Checking for Networks. In *USENIX NSDI*, San Jose, CA, USA, 2012.

[19] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *USENIX NSDI*, Lombard, IL, USA, 2013.

[20] D Richard Kuhn, Dolores R Wallace, and Albert M Gallo. Software Fault Interactions and Implications for Software Testing. *IEEE Transactions on Software Engineering*, 30(6), 2004.

[21] Hongqiang Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. CrystalNet: Faithfully Emulating Large Production Networks. In *ACM SOSP*, 2017.

[22] Mallory Locklear. Google accidentally broke the internet throughout Japan. https://www.engadget.com/2017-08-28-google-accidentally-broke-internet-japan.html, August 2017. Accessed: 2020-09-12.

[23] Nuno P Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. Checking Beliefs in Dynamic Networks. In *USENIX NSDI*, Oakland, CA, USA, 2015.

[24] Doug Madory. Widespread Impact Caused by Level 3 BGP Route Leak. https://blogs.oracle.com/internetintelligence/widespread-impact-caused-by-level-3-bgp-route-leak, November 2017. Accessed: 2020-09-12.

[25] Microsoft. PICT - Pairwise Independent Combinatorial Testing. https://github.com/microsoft/pict, 2020.

[26] Ghassan Misherghi and Zhendong Su. HDD: Hierarchical Delta Debugging. In *ICSE*, Shanghai, China, 2006.

[27] Matthew Prince. August 30th 2020: Analysis of CenturyLink/Level(3) Outage. https://blog.cloudflare.com/analysis-of-todays-centurylink-level-3-outage/, August 2020. Accessed: 2020-09-12.

[28] Bruno Quoitin and Steve Uhlig. Modeling the Routing of an Autonomous System with C-BGP. *IEEE Network*, 19(6), November 2005.

[29] Leonid Ryzhyk, Nikolaj Bjørner, Marco Canini, Jean-Baptiste Jeannin, Cole Schlesinger, Douglas B Terry, and George Varghese. Correct by Construction Networks Using Stepwise Refinement. In *USENIX NSDI*, Boston, MA, USA, 2017.

[30] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D. Ernst, Arvind Krishnamurthy, and Zachary Tatlock. Scalable Verification of Border Gateway Protocol Configurations with an SMT Solver. In *ACM OOPSLA*, Amsterdam, Netherlands, 2016.

[31] W. Eric Wong, Xuelin Li, and Philip A. Laplante. Be more familiar with our enemies and pave the way forward: A review of the roles bugs played in software failures. *Journal of Systems and Software*, 133, 2017.

[32] Fangdan Ye, Da Yu, Ennan Zhai, Hongqiang Harry Liu, Bingchuan Tianx, Qiaobo Ye, Chunsheng Wang, Xin Wu, Tianchen Guo, Cheng Jin, et al. Accuracy, Scalability, Coverage: A Practical Configuration Verifier on a Global WAN. In *ACM SIGCOMM*, Virtual Event, NY, USA, 2020.

[33] Andreas Zeller. Yesterday, my program worked. Today, it does not. Why? *ACM SIGSOFT FSE-7*, 1999.

[34] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. Fuzzing: Breaking things with random inputs. In *The Fuzzing Book*. Saarland University, 2020. Accessed: 2020-09-12.

[35] Andreas Zeller and Ralf Hildebrandt. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering*, 28(2), 2002.

[36] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. Automatic Test Packet Generation. In *ACM CoNEXT*, Nice, France, 2012.

A Proof of Theorem 1

Proof. By contradiction. Assume that there is a minimal subset $Q' \subseteq Q$ such that $tc|_{Q'}$ fails which is not returned in $\mathcal{S}$. We check at least one superset of Q' for a failure since we will always check the initial set Q. Assume $C \supseteq Q'$ is a smallest superset of Q' which is checked. By the assumption of monotonicity, $tc|_C$ must fail, therefore we will minimize C. If $C = Q'$, then we must minimize to Q' since Q' is assumed to be minimal, violating the assumption that Q' is not returned by the algorithm. If $Q' \subset C$, then C will either minimize to Q' (again violating the original assumption that Q' is not returned by the algorithm) or to a different minimal subset P. In this case, we generate additional sets to be tested. However, both Q' and P are minimal subsets of C, therefore $Q' \not\subset P$ and $P \not\subset Q'$. Since $Q' \neq P$, we know that there must be an element $e \in P$ which is not in Q', i.e., such that $Q' \subseteq C \setminus \{e\}$. The set $C \setminus \{e\}$ is both strictly smaller than C and will be added to the sets to check by the algorithm in Line 9 and therefore violates our assumption that C was a smallest superset of Q' which is checked. $\square$

Finding Invariants of Distributed Systems: It's a Small (Enough) World After All

Travis Hance, Marijn Heule, Ruben Martins, and Bryan Parno
Carnegie Mellon University

Abstract

Today's distributed systems are increasingly complex, leading to subtle bugs that are difficult to detect with standard testing methods. Formal verification can provably rule out such bugs, but historically it has been excessively labor intensive. For distributed systems, recent work shows that, given a correct inductive invariant, nearly all other proof work can be automated; however, the *construction* of such invariants is still a difficult manual task.

In this paper, we demonstrate a new methodology for automating the construction of inductive invariants, given as input a (formal) description of the distributed system and a desired safety condition. Our system performs an exhaustive search within a given space of candidate invariants in order to find and verify inductive invariants which suffice to prove the safety condition. Central to our ability to search efficiently is our algorithm's ability to learn from counterexamples whenever a candidate fails to be invariant, allowing us to check the remaining candidates more efficiently. We hypothesize that many distributed systems, even complex ones, may have concise invariants that make this approach practical, and in support of this, we show that our system is able to identify and verify inductive invariants for the Paxos protocol, which proved too complex for previous work.

1 Introduction

The world increasingly relies on distributed computer systems, but the correctness and reliability of these systems depend on the imperfect coverage of testing. To obtain strong guarantees, developers are starting to turn to formal verification techniques, both in research [3, 21, 24, 31, 46, 54] and industry [38, 58]. These techniques can, in theory, prove that all possible execution traces of the system conform to a high-level safety specification (e.g., all nodes agree on the next input value, or no two nodes hold a lock at the same time).

However, verifying the safety of distributed systems can be extremely labor-intensive, especially when using general-purpose theorem provers [21, 45, 54]. For example, Hawblitzel et al. [21] report that the effort to build and verify two distributed systems (including their protocols and implementations) required 3.7 person-years, and their safety proofs (over 19K lines of code) account for $\sim$40% of the total codebase. To reduce this cost, some work has developed specialized languages that either restrict the kinds of systems that can be encoded [12, 13, 36, 53] (e.g., the protocol must proceed in synchronous rounds), or restrict the language used to describe the systems' properties [42]. In exchange for these restrictions, much of the safety proof can be dispatched automatically.

In all of these systems, the core of the safety proof requires identifying system *invariants*, and even with tools that can automate all other proof work, finding these invariants still relies on human labor and ingenuity. Anecdotally, this is a challenging task even for researchers [34], requiring days for toy systems and months for complex protocols like Paxos [29].

Ideally, we would automate invariant discovery, but theoretical results show that even in languages where *checking* an invariant is decidable [42], *finding* such an invariant is not decidable [40], meaning that no algorithm can guarantee that it will find an invariant for arbitrary protocols, even if they are indeed safe. Hence, we must turn to domain-specific insights to develop a methodology which can apply to the specific kinds of distributed systems developed in practice.

Recent approaches have been proposed to automatically infer invariants for distributed systems [27, 34]. For instance, the I4 [34] system observes that the invariants for some distributed systems are scale-invariant. Hence they ask the developer to specify appropriate finite-model parameters and to concretize some of the protocol variables. I4 then invokes a specialized model checker [19] that can deduce invariants on finite, fixed-size instances. It then employs various heuristics to generalize those invariants to arbitrary instances. Unfortunately, I4 cannot discover invariants that use existential quantifiers. Furthermore, since the model checker is a black box, the developer has no recourse when it fails. More recently, Koenig et al. [27] extend the IC3/PDR algorithm [6, 14] to infer invariants with existential quantifiers. However, neither they nor I4 can handle complex protocols like Paxos [29], for which the simplest known safety proofs require many invariants, including some with existential quantifiers.

To tackle complex protocols like Paxos, we explore an alternate approach based on our "small world" hypothesis: in many practical systems, the invariants should be relatively concise. After all, these protocols are designed by humans who have some (finite) intuition for why the protocol is correct. They are clearly not beyond human comprehension. If accurate, this hypothesis suggests we are faced with a finite space of possible invariants, which we can potentially search exhaustively.

Of course, finite is not the same as small. For a protocol like Paxos, there could be over 100 billion candidate invariants with just six terms. Even if it only took a millisecond to test each invariant, a brute force search of the entire space would require over three computation-years.

Hence, this paper presents the Small World Invariant Search System (SWISS), our system for automatically proving the safety of distributed systems by efficiently searching through the space of succinct invariants. At its core, SWISS learns from counterexample models obtained from failed invariant candidates to speed up future invariant checks. SWISS uses an SMT solver to perform these checks. SWISS also exploits various symmetries and parallelism to reduce search time. Further, for a class of invariants (described formally in §3.4) SWISS is guaranteed to find an invariant when one exists. SWISS also supports user-supplied guidance to make the search more efficient. Moreover, unlike prior approaches, even when SWISS does not produce a safety proof within a given time limit, the user still benefits from partial invariants that SWISS did find, as they can use those invariants as a starting point for finding complete invariants. The net effect is that SWISS is the *first approach that can automatically prove the safety of Paxos*, and it does so in around 4 hours on an 8-core machine. If the user provides some light guidance (§5.3.3), then the time decreases to 20 minutes.

We compare SWISS to I4 [34] and Koenig et al. [27] on a large variety of protocols (§5). None of the approaches fully dominates the others, neither in protocols solved nor in runtime. For instance, for protocols that only require invariants without existential quantifiers, I4 is generally fastest. However, for some complex distributed protocols with many invariants that contain existential quantifiers, SWISS outperforms the other tools, and in particular, it is the first to automatically prove the safety of Paxos and two variants of Paxos: Multi-Paxos and Flexible-Paxos [22]. There are still some variants of Paxos that SWISS cannot fully prove in a reasonable amount of time. However, even in these cases, SWISS still finds many partial invariants that may help the user complete the proof.

In summary, our key contributions are as follows.

- We propose and evaluate the *Small World Hypothesis*: that many distributed systems we care for can be proven safe using a sequence of concisely specified invariants.
- We present SWISS, a methodology for automatically proving the safety of distributed systems by efficiently searching through the space of candidate invariants, guided by knowledge learned from counterexamples encountered during the search. We report both our successful *and* unsuccessful optimizations.
- To our knowledge, SWISS is the first approach to identify and verify inductive invariants that prove the safety of Paxos in an *automated* fashion.

2 Background

We briefly present background on formalizing and verifying distributed systems.

2.1 Proving Safety Conditions Via Inductive Invariants

We aim to prove *safety conditions*. A safety condition is a desired property that ought to hold true at any point in a system's execution. For an exclusion lock, for example, the safety condition might be that no two agents hold a lock at the same time. For a consensus protocol such as Paxos, the condition would be that no two machines decide on different results. These safety conditions are in contrast with *liveness conditions*, which say that the system eventually performs a useful action. We do not currently consider liveness conditions.

To formalize a distributed system, one must formally describe the internal state of the participating nodes, the state of the network (packets in-flight between nodes), the initial conditions of the system, and the ways the system can evolve. Following standard practice, we formalize the latter as a sequence of atomic actions that update the system state [28].

Given this formal description, our goal then becomes to prove that the safety condition will hold for any reachable state in any possible execution of the system. The typical strategy for such a proof is to find an *inductive invariant*. An inductive invariant should **(i)** hold on all possible initial states of the system and **(ii)** continue to hold when the system transitions to a new state from a state where the invariant held.

If we could prove that the safety condition was inductive, we would be done; unfortunately, this is rarely the case for non-trivial systems. Instead, we typically must find an invariant which is stronger than the safety condition and then show this invariant is both inductive *and* implies the safety condition.

2.2 A Running Example: Simple Decentralized Lock

To make the verification process more concrete, we present a toy example: a Simple Decentralized Lock (SDL) protocol. SDL supports a single mutual-exclusion lock that is shared among multiple nodes. A node with the lock can send the lock to another machine via a message over the network, which, in this simple example only, does not permit packet duplication.

Figure 1 formally describes the SDL protocol. The state at a snapshot in time is represented by two relations: message and lock. The value message(src, dst) indicates a message in the network from machine *src* to machine *dst*, while lock$(node)$ indicates that *node* believes it holds the lock.

In the protocol's initial state (the **init** lines), the network is empty, and only one node (*start_node*) holds the lock.

The transitions are written in RML [42], an abstract language for describing system transitions. SDL has two transition actions: `send` and `recv`. In `send`, a node *with the lock* can atomically release the lock and send a packet to another machine. In `recv`, a node can receive a packet (removing it from the network) and accept the lock.

The safety condition (Figure 2) we wish to verify for SDL is

```
type node
relation message(src : node, dst : node): bool
relation lock(N : node): bool
init ∀src, dst. ¬message(src, dst)
init ∃startnode : node.
    has_lock(startnode)
    ∧ ∀N : node. (N ≠ startnode ⟹ ¬lock(N))
action send(src: node, dst: node) {
    require lock(src);
    message(src, dst) := true;
    lock(src) := false;
}
action recv(src: node, dst: node) {
    require message(src, dst);
    message(src, dst) := false;
    lock(dst) := true;
}
```

Figure 1: **The Simple Decentralized Lock Protocol**

SDL safety condition

$$\forall n_1, n_2 : \text{node.}\ \text{lock}(n_1) \wedge \text{lock}(n_2) \implies n_1 = n_2$$

SDL inductive invariant

$$(\forall n_1, n_2 : \text{node.}\ \text{lock}(n_1) \wedge \text{lock}(n_2) \implies n_1 = n_2) \wedge$$
$$(\forall n_1, n_2, n_3 : \text{node.}\ \neg(\text{lock}(n_1) \wedge \text{message}(n_2, n_3))) \wedge$$
$$(\forall n_1, n_2, n_3, n_4 : \text{node.}\ \text{message}(n_1, n_2) \wedge \text{message}(n_3, n_4) \implies n_1 = n_3 \wedge n_2 = n_4)$$

Figure 2: **Safety Condition and Invariants for the SDL**

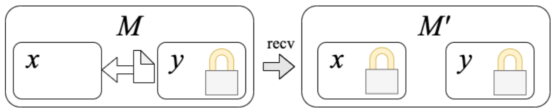


Figure 3: Two models which demonstrate that the safety condition (Figure 2) for SDL is not inductive on its own. M and M' each have a *domain* of two nodes, n_1 and n_2. In M, n_2 holds the lock, while a packet is in-flight from n_2 to n_1. M can transition to M' via the `recv` action. M does not violate the safety condition, but M' does (n_1 and n_2 both hold the lock).

that no two nodes ever hold the lock at the same time. Figure 3 shows that this condition is *not* inductive. There is a state, M, which satisfies the safety condition (only one machine holds the lock) which can transition to a state, M' (via a `recv` on y) which does not satisfy the safety condition (since two machines hold the lock). Figure 2 shows a stronger predicate which also rules out the first state. This predicate is both inductive and (trivially) implies the safety condition; thus, it completes our safety proof. It states that **(i)** no two machines hold the lock, **(ii)** no machine holds the lock while a message exists, and **(iii)** no two messages exist at the same time.

2.3 Formal Notation

To keep our subsequent discussion precise we introduce some additional notation. Formally, a *transition system* is a triple $\mathcal{T} = (\Sigma, \mathit{INIT}, \mathit{TR})$, where Σ defines the types and relations representing the state of a system, and INIT and TR are predicates describing the initial state of the system and allowed transitions of the system, respectively. In our SDL example, Σ contains the type `node` and the relations lock and message.

Since TR relates two states, we will use primes to indicate the new state, e.g., $\mathit{TR} := (x' = x + 1)$ represents a transition that increments x by one.

A *model* M represents a single possible state of the system. It contains an assignment for each variable and each possible evaluation of the system's relations. For a predicate P, we write $M \models P$ if the predicate P evaluates to *true* on model M. When P is a predicate over two states, we write $(M, M') \models P$, e.g., we write $(M, M') \models \mathit{TR}$ if M can transition to M'.

We say that M is *reachable* if there exists a sequence $M_0, \ldots, M_k$ where $M_0 \models \mathit{INIT}$, $M_k = M$, and $(M_i, M_{i+1}) \models \mathit{TR}$ for all i. A formula S is said to be *safe* if for all reachable M, we have $M \models S$.

We say that I (a formula over Σ) is *inductive* if we can prove that $\mathit{INIT} \implies I$ and $\mathit{TR} \wedge I \implies I'$ (i.e., if I is true and the system can take a transition to a new state, then I holds there as well). Here, we use I' to denote the predicate I evaluated on the second state. It is clear that any inductive invariant I will be safe. Therefore, proving that S is safe amounts to finding an inductive invariant I such that $I \implies S$; equivalently, to find a formula I such that $I \wedge S$ is an inductive invariant.

2.4 Decidability of Inductiveness

Given a candidate invariant I, we must prove that **(i)** $\mathit{INIT} \implies I$, **(ii)** $\mathit{TR} \wedge I \implies I'$, and **(iii)** $I \implies S$. We call these *verification conditions*. To check that a verification condition P holds for all possible models, we can show that $\neg P$ is unsatisfiable; i.e., there is no model M such that $M \models \neg P$.

Checking the validity of arbitrary first-order logic formulas is undecidable [52], but prior work [41, 42] shows that many distributed systems, including multiple variants of Paxos, can be encoded in RML [42], which can be translated to a restricted class of formulas where satisfiability *is* decidable. In particular, the class of *effectively propositional* (EPR) formulas, also known as the Bernays–Schönfinkel class, are the class of formulas that can be written in a form with quantifier prefix $\exists^*\forall^*$ and no function symbols. Satisfiability for this class of formulas is decidable [33, 43].

By ensuring our verification conditions lie within this class, we can always either verify the predicate $\mathit{INIT} \implies I$ or find a satisfying instance for $\mathit{INIT} \wedge \neg I$. Likewise, we can either verify the predicate $\mathit{TR} \wedge I \implies I'$ or find a pair (M, M') where $(M, M') \models \mathit{TR} \wedge I \wedge \neg I'$ (e.g., the pair in Figure 3). In either

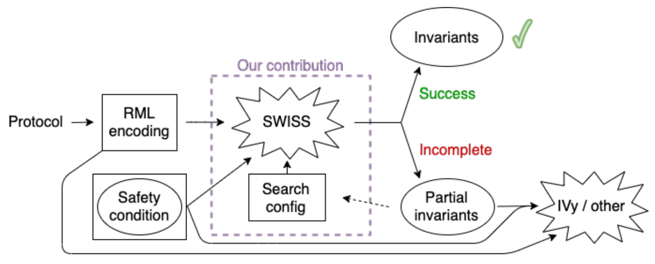


Figure 4: **SWISS Overview**. The intended workflow for using SWISS to synthesize an invariant and safety proof. A rectangle indicates a machine-readable, human-supplied input. An oval represents a collection of first-order predicates.

case, we obtain a concrete counterexample.

EPR, as stated, is a bit too restrictive; it allows only relations (not general functions), and it does not allow for invariants with quantifier alternation. However, EPR can be extended to include *stratified* function symbols—functions for which the edges from input types to output types form no cycles—while maintaining its decidability. In the same way, we can allow stratified alternation of universal and existential quantifiers in a fragment called *the extended EPR fragment* [41]. To keep our verification conditions within this class, we must impose some restrictions on the quantifier alternations which appear in I; these restrictions are determined by the shape of the protocol under consideration, in particular, the quantifier alternations which appear in $INIT$ and TR.

3 Overview: The SWISS Algorithm

SWISS is an algorithm for inferring inductive invariants of a protocol in order to prove a desired safety condition. The intended usage of SWISS is shown in Figure 4. First, the user provides an RML-encoding [42] of the protocol, a desired safety condition, and a specification of the search space (§4.1.1). SWISS either succeeds with an invariant that proves the safety condition, or it fails, with only partial invariants generated. In that case, the user may choose how to continue: they might try SWISS again with a different search configuration, or they might continue with other means, e.g., the interactive invariant-finding tool IVy [42], using the partial invariants as a starting point.

For example, suppose the user is interested in the SDL protocol (§2) and wants to prove the lock-exclusivity safety condition. They would first encode the SDL protocol into a machine-readable RML specification (Figure 1). Protocols are often concise, although in some cases it is challenging to produce a specification where the inductive invariant will be in EPR [41]. However, we consider this out of scope for SWISS.

Next, the user would write the exclusivity condition as a predicate (Figure 2). They would also choose the space of predicates to search over (§4.1.1). If the user knows nothing about the protocol, they would likely choose the most general option, to generate templates automatically. If SWISS succeeds, then they will know that the lock-exclusivity safety condition is true, and they can use the invariants from SWISS's output to validate that SWISS ran correctly.

If SWISS does not succeed, there are a few possible paths. For example, it might be that SWISS does not complete in a reasonable amount of time, in which case the user might choose to restrict the search space. For example, they might have some idea of what the invariant should look like because they have worked with similar locking protocols previously. Alternatively, if SWISS completed quickly but did not succeed, the user might choose a broader search space.

Finally, they might choose to take the incomplete invariants generated by SWISS and attempt to complete them through other means. Even though SWISS did not succeed, the user could learn useful information about the protocol from these incomplete invariants.

3.1 High-Level Algorithm

We begin with a high-level overview of our algorithm for finding the inductive invariants needed to prove safety conditions. Section 4 then explains how we make the algorithm scale to large search spaces.

SWISS takes as input **(i)** a transition system $\mathcal{T}$ encoded via RML (§2.2) **(ii)** a safety condition S, and **(iii)** a configuration of the search space (§4.1.1). In this section, we refer to search spaces with the symbols $\mathcal{B}$ and $\mathcal{F}$, which here may be viewed as arbitrary sets of first-order predicates.

SWISS is designed to exploit our hypothesis that a distributed system designed by humans will have a concise invariant, or a larger invariant composed of concise invariants. After all, the designer presumably has a finite intuition for the correctness of their system, either as a whole or as the conjunction of correct subsystems or subproperties.

Internally, SWISS uses different algorithms to target these two possible invariant styles. One algorithm, Finisher (§3.2), tries to directly find one inductive invariant that proves the safety condition. Hence any invariant it finds will necessarily complete the safety proof. Using the safety condition as a target helps Finisher search the invariant space efficiently. However, for complex protocols, searching for the entire system invariant in one shot is infeasible; e.g., a human-derived invariant for Paxos has 10 conjuncts with 34 terms, corresponding to a search space of over 10^{75} candidate invariants.

Hence, SWISS employs a second algorithm, Breadth (§3.3), that greedily searches for as many protocol invariants as possible within a finite space $\mathcal{B}$, without requiring that they directly prove the safety condition. We run Breadth multiple times so that invariants may build on each other: once Breadth finds an invariant P, the next run can then find an invariant Q that is inductive relative to P, even when Q might not be inductive on its own. More formally, we say that Q is *relatively inductive* with respect to P if $INIT \implies Q$ and $TR \wedge P \wedge Q \implies Q'$.

Algorithm 1: Solve $(\mathcal{T}, S, \mathcal{B}, \mathcal{F})$

```
invariants ⟵ {};
while true do
    newInvariants ⟵ Breadth($\mathcal{T}$, invariants, $\mathcal{B}$);
    if newInvariants imply safety then
        return newInvariants
    if newInvariants = invariants then
        break;
    invariants ⟵ newInvariants;
return Finisher($\mathcal{T}$, invariants, $S$, $\mathcal{F}$) ∪ invariants
```

The Breadth loop ultimately builds an invariant of the form $P_1 \wedge \cdots \wedge P_n$, where each P_i is relatively inductive to $P_1 \wedge \cdots \wedge P_{i-1}$. In practice (§5), without the safety condition guiding it, Breadth is slower than Finisher, but it can incrementally construct a larger invariant than Finisher can.

To benefit from the strengths of both Breadth and Finisher, SWISS's top-level Solve (Algorithm 1) combines them. It takes as input a transition system, $\mathcal{T}$, and a desired safety condition S. It also takes in two spaces of candidate invariants, $\mathcal{B}$ and $\mathcal{F}$, for Breadth and Finisher, respectively, to search. In our implementation, these spaces are defined through a combination of the protocol description and (potentially) user input (§4.1). Solve runs Breadth over $\mathcal{B}$ until no new invariants are produced, and then it runs Finisher, if necessary, to find an additional invariant needed to complete the safety proof. Section 3.4 summarizes SWISS's coverage guarantee.

3.2 The Finisher Algorithm

Finisher aims to find a single invariant P which proves a safety condition S. More formally, it tries to solve:

Task 1 (Conjecture-proving task.) *Given a transition system $\mathcal{T}$, formulas $I_1, ..., I_n$, already established (or assumed) to be invariant, and a conjectured safety condition S, find an invariant predicate P such that $P \wedge S$ is inductive relative to $I_1 \wedge \cdots \wedge I_n$.*

Evaluating a candidate invariant P requires checking the validity of the following *verification conditions* (VCs):

$$\begin{aligned} INIT &\implies S \\ INIT &\implies P \\ TR \wedge I_1 \wedge \cdots \wedge I_n \wedge S \wedge P &\implies S' \\ TR \wedge I_1 \wedge \cdots \wedge I_n \wedge S \wedge P &\implies P' \end{aligned}$$

Since the $INIT \implies S$ condition does not depend on P, we can check it once in advance of evaluating any candidat invariant.

Since we consider protocols expressed in RML (§2.4), checking the validity of these VCs is decidable, and in practice, typically quite efficient with modern SMT solvers. Hence, for a candidate invariant P, we can run a subroutine CheckVCsF to determine either that the VCs above hold, or that a finite counterexample shows they do not. As Algorithm 2 shows, rather than simply check the VCs above for each candidate predicate P in $\mathcal{F}$, Finisher accumulates a collection of counterexamples from failed candidates. As we describe in §4.2, we use these counterexamples to filter subsequent candidates, as our counterexample check is orders of magnitude faster than the VC check.

Algorithm 2: Finisher$(\mathcal{T}, \{I_1, \ldots, I_n\}, S, \mathcal{F})$

```
cexamples ⟵ {};
for $P \in \mathcal{F}$ do
    if $\forall cex \in$ cexamples . Passes($P$, $cex$) then
        $cex$ ⟵ CheckVCsF($\mathcal{T}$, $\{I_1,\ldots,I_n\}$, $S$, $P$);
        if $cex$ is None then
            return $P$;
        else
            cexamples ⟵ cexamples ∪ {$cex$};
return None
```

Algorithm 3: Breadth$(\mathcal{T}, \{I_1, \ldots, I_n\}, S, \mathcal{B})$

```
cexamples ⟵ {};
allInv ⟵ $\{I_1,\ldots,I_n\}$;
indInv ⟵ $\{I_1,\ldots,I_n\}$;
for $P \in \mathcal{B}$ do
    if $\forall I \in$ allInv . ¬FastImplies($I$, $P$) then
        if $\forall cex \in$ cexamples . Passes($P$, $cex$) then
            $cex$ ⟵ CheckVCsB($\mathcal{T}$, $\{I_1,\ldots,I_n\}$, $S$, $P$);
            if $cex$ is None then
                allInv ⟵ allInv ∪ {$P$};
                if ¬ Redundant ($P$, indInv) then
                    indInv ⟵ indInv ∪ {$P$};
            else
                cexamples ⟵ cexamples ∪ {$cex$};
return indInv;
```

3.3 The Breadth Algorithm

In the Breadth algorithm, our goal is simply to find as many invariants as possible. More formally, we wish to solve the following task.

Task 2 (Invariant-finding task.) *Given a transition system $\mathcal{T}$ and formulas $I_1, ..., I_n$, already established (or assumed) to be invariant, find any invariant predicate P which is inductive relative to $I_1 \wedge \cdots \wedge I_n$.*

The corresponding VCs are as follows.

$$\begin{aligned} INIT &\implies P \\ TR \wedge I_1 \wedge \cdots \wedge I_n \wedge P &\implies P' \end{aligned}$$

Naively, we could start with an algorithm similar to Finisher, but use the VCs above instead of Finisher's. However, this would result in a highly inefficient search since as stated, the invariant-finding task permits many tautological solutions, such as $true$ or I_i. More generally, if P is any predicate such that $I_1 \wedge \cdots \wedge I_n \implies P$, then P will be invariant, but we do not actually learn anything about the protocol by finding P: P does not rule out any states which were not ruled out by the I_i. We call such a P a *redundant invariant* with respect to $I_1 \wedge \cdots \wedge I_n$. For example, $\forall x. f(x) \vee g(x)$ is redundant with respect to $\forall x. f(x)$. As the number of terms in our search space increases, the number of redundant invariants increases exponentially. Furthermore, since any redundant invariant is, in fact, inductive, it will always pass our counterexample filters, guaranteeing an expensive VC check for each.

We devised two ways to cope with redundant invariants. First, we maintain a set *indInv* of non-redundant invariants, and whenever we find a new invariant, we explicitly check whether it is redundant with *indInv*. Second, we also track *allInv*, i.e., all invariants we find—including the redundant ones—and use these to syntactically filter future candidates by performing quick checks for logical implication using a subroutine FastImplies(I,P) described in §4.4. These checks are vastly cheaper than SMT calls.

As described thus far, Breadth works without any knowledge of the safety condition S that we aim to prove about our system. However, we observe that we will eventually need S to be an invariant of the system, so we strengthen the second VC above to assume S is true in the initial state.

$$TR \wedge I_1 \wedge \cdots \wedge I_n \wedge S \wedge P \implies P'$$

Algorithm 3 brings all of this together. This algorithm finds exactly the invariants from the space $\mathcal{B}$ which are invariant with respect to the original input invariants. More precisely, if $P \in \mathcal{B}$ is invariant with respect to the inputs, then Breadth will output a set of predicates whose conjunction implies P.

3.4 Swiss Coverage

Swiss is guaranteed to prove a safety condition S as long as the system invariant conforms to the following form.

Claim 1 Solve$(\mathcal{T},S,\mathcal{B},\mathcal{F})$ *will always succeed at proving the conjectured safety condition S provided there exist invariants* $I_1,\cdots,I_n$ *such that:*

- $I_i \in \mathcal{B}$ *for* $1 \le j \le n-1$.
- $I_n \in \mathcal{F}$.
- $I_1 \wedge \ldots \wedge I_j$ *is inductive relative to S for* $1 \le j \le n-1$.
- $I_1 \wedge \ldots \wedge I_n \wedge S$ *is inductive.*

The claim follows from inspection of Algorithms 1-3 and the fact that our counterexample and implication filters will never eliminate a valid invariant.

4 Making Invariant Exploration Scale

While the algorithms described in §3 would theoretically suffice to find invariants, implemented naively they are impractically slow. Hence, we present crucial steps we take to reduce the search space (§4.1-§4.4) and optimize the overall process (§4.5). For the sake of completeness, we also present three optimizations that our evaluation has shown *do not* improve performance in this domain (Appendix A).

4.1 Exploiting User Guidance & Candidate Symmetries

The Finisher and Breadth algorithms each take as input a space of candidate invariants to explore (§4.1.1). These spaces often contain many invariants that are identical modulo symmetries, so symmetry pruning is critical (§4.1.2).

4.1.1 Defining Candidate Spaces

Swiss searches for invariants based on *invariant templates*. Intuitively, a template defines the rough shape of a class of invariants (e.g., the number and types of the quantified variables, and a bound on the formula's size).

By default, Swiss automatically defines invariant templates based on the protocol description and the safety condition, as well as user-supplied upper bounds on the formula size. Swiss then enumerates all template spaces within these constraints, and the search space $\mathcal{B}$ or $\mathcal{F}$ is defined as the union of these spaces. Finisher prioritizes these templates in order of increasing size. Thus, if a small invariant exists, Finisher will find it without having to search the entire space.

For many protocols, this fully automatic approach suffices to produce a safety proof. For protocols where this automated search runs slower than desired, the user can specify a particular template T, rather than have Swiss enumerate all templates. As §5.3.3 demonstrates, such user guidance can dramatically speed up the search process, even if the user guesses a few incorrect templates before the right one.

More formally, every candidate invariant P that we consider is a sequence of quantifiers followed by a quantifier-free expression E. The expression E is a tree of conjunctions and disjunctions of *terms* C, expressed over values V, which is either a name and a type (e.g., $x : t$), or a function application.

$$\begin{array}{lll} V & ::= & x:t \mid f(V_1,\ldots,V_n) \\ C & ::= & V_1 = V_2 \mid \neg C \mid r(V_1,\ldots,V_n) \\ E & ::= & C \mid E_1 \wedge \cdots \wedge E_n \mid E_1 \vee \cdots \vee E_n \\ P & ::= & E \mid \forall x:t.\, P \mid \exists x:t.\, P \end{array}$$

We specify a set of candidate formulas by a triple (T,k,d). The *template* T is simply a formula P with a wildcard for the quantifier-free expression E. We define TemplateSpace(T,k,d) to be the set of formulas that match T when the wildcard is instantiated with any quantifier-free expression E that has at most k terms and a conjunction-disjunction tree of depth at most d. For instance, an expression of the form $c_1 \vee c_2 \vee c_3 \vee c_4$ has depth 1, while $(c_1 \vee c_2) \wedge (c_3 \vee c_4)$ has depth 2.

Hence, a valid candidate space might be defined by,

$$T = \forall r : \text{round}, v : \text{value}.\ \exists q : \text{quorum}.\ \forall n_1, n_2 : \text{node}.\ *$$

with $k = 3$, and $d = 1$, which would contain all invariants with the quantifiers in T and disjunctions of up to 3 terms. Swiss expects the user to specify the maximum values of k, d, m (the maximum total number of quantified variables) and q (the maximum number of *existentially* quantified variables). The user must also specify a quantifier nesting order for the types defined by the protocols: a fixed nesting order is required so that the resulting verification conditions remain in EPR.

To enumerate the formulas in TemplateSpace(T,k,d), we first enumerate a set of possible terms, $\mathcal{C}$, and then arrange them in every possible tree shape. While succinct to describe, the size of the space grows exponentially, so we take steps to prune the space as rapidly as possible.

4.1.2 Symmetry-Breaking

As defined above, a candidate space contains many logically equivalent candidates, such as:

$$\forall r_1, r_2 : \text{round} \;.\; leq(r_1,r_2) \vee geq(r_1,r_2)$$
$$\forall r_1, r_2 : \text{round} \;.\; geq(r_1,r_2) \vee leq(r_1,r_2)$$
$$\forall r_1, r_2 : \text{round} \;.\; geq(r_2,r_1) \vee leq(r_2,r_1)$$

all identical up to term reordering and variable renaming.

When enumerating candidate formulas, SWISS aims to only consider one representative from each class of identical candidates. However, checking for such logical equivalence via SMT calls would be prohibitively expensive. Hence, we use a sound set of syntactic constraints to break these symmetries far more efficiently.

Specifically, SWISS assigns an arbitrary ordering to each possible base term (e.g., *leq*) and will only produce formulas where, within any conjunction or disjunction, the terms are in increasing order. SWISS also produces formulas such that for any set of quantified variables which are interchangeable (e.g., r_1 and r_2 in the example above), their first appearances are in increasing order of quantifier nesting index.

These two steps efficiently break symmetries arising from term ordering and variable permutation. In practice, they reduce the size of the search space by over two orders of magnitude (§5).

4.2 Filtering Based on Counterexamples

As explained in §2.4, when a candidate invariant P fails a verification condition, we can extract a counterexample, specifically a concrete model where the condition failed. For example, if $INIT \implies P$ fails to hold, then we can extract a concrete model M such that $M \models INIT \wedge \neg P$; i.e., the initial conditions hold for M but P does not. Since the initial conditions hold on M, if any other formula P' fails to hold for M, then it cannot be an invariant either. SWISS exploits this observation by remembering the models from failed VC checks and using those models to quickly rule out future candidates.

More technically, we define a *counterexample filter* (*cex*) as a model or a pair of models which demonstrate that a given candidate P fails to be an inductive invariant. We consider three types of counterexample filters:

$$cex \;::=\; \mathsf{True}(M) \;\mid\; \mathsf{False}(M) \;\mid\; \mathsf{Transition}(M,M')$$

A candidate P *passes* a filter *cex* if one of the following holds.

- $cex = \mathsf{True}(M)$ and $M \models P$; i.e., all valid invariants should evaluate to true on M.
- $cex = \mathsf{False}(M)$ and $M \models \neg P$.
- $cex = \mathsf{Transition}(M,M')$ and $(M,M') \models P \implies P'$.

The last is equivalent to: P passes $\mathsf{False}(M)$ *or* $\mathsf{True}(M')$.

We construct counterexample filters based on the model(s) returned when a candidate P fails a VC check. The specific type of filter constructed depends on which type of check fails. Our construction guarantees that **(i)** P does not pass the filter *cex*, and **(ii)** any formula P' which *does* pass the same verification condition *will* pass the filter *cex*. The three possible constructions are as follows.

- A failed verification condition of the form $A \implies P$ yields a model M such that $M \models A \wedge \neg P$, so we produce a counterexample filter $\mathsf{True}(M)$. In other words, P does not evaluate to *true* on M; but any valid invariant should.
- A failed verification condition of the form $A \wedge P \implies B'$ yields models M and M' such that $(M,M') \models A \wedge P \wedge \neg B'$, which gives a counterexample filter $\mathsf{False}(M)$ (since the model represents a state M that P failed to reject and that all valid invariants ought to reject).
- A failed verification condition of the form $A \wedge P \implies P'$ yields models M and M' such that $(M,M') \models A \wedge P \wedge \neg P'$, which gives a counterexample filter $\mathsf{Transition}(M,M')$.

Essentially, these counterexample filters allow us to learn from each type of failed induction check.

Efficient Implementation. Counterexample filtering is only useful if we can apply our filters faster than executing the original VC checks. Hence, whenever we add a new model M to our set of counterexample filters, we precompute the evaluation of each term $c \in \mathcal{C}$ on M. $\mathcal{C}$ is the set of base terms (§4.1.1) produced for every possible instantiation of the quantifier variables in the template T within the model M. The evaluations are saved in a bitstring, so when we encounter a new candidate Q, the evaluation of Q on M is computed quickly with bitwise operations. As a result, counterexample filters are orders of magnitude faster than VC checks (§5).

4.3 Checking Verification Conditions

We encode our verification conditions into SMT formulas in a manner similar to prior work [42]. The encoding ensures that it is decidable to evaluate the validity of the formulas. Our encoding conforms to the standardized `smtlib2` format, although for better performance, our implementation includes the SMT solver as a library so that it can directly invoke its API rather than communicating via files.

In practice, we find that thanks to the community's large effort invested in optimizing SMT solvers, our validity queries are not just decidable, but *rapidly* decidable, typically in tens of milliseconds. However, our initial experiments found occasional outliers that consumed minutes, or more. Since a SWISS run may perform thousands of SMT calls, these outliers become an issue. After some iteration, we settled on a routine where we retry a problem instance with a different SMT solver if our first attempt exceeds a given time limit (§5.1). If the SMT instance continues to time out, we skip the invariant in question. While not a perfect solution, we found that this routine satisfactorily avoids stalling SWISS's execution.

4.4 Filtering Redundant Invariants

Since the Breadth algorithm searches rather indiscriminately for any possible invariant, it can easily waste time on redundant invariants, i.e., invariants already implied by previously established invariants.

To efficiently prune many such redundant invariants, SWISS includes a rapid FastImplies filter to see if an existing invariant Q already implies a candidate P. For correctness, FastImplies(Q,P) can return *true* only when $Q \implies P$. Our implementation of FastImplies(Q,P) always detects the case where Q and P can be written as,

$$Q = \forall^* \exists^* \dots \ (a_1 \vee \dots \vee a_j)$$
$$P = \forall^* \exists^* \dots \ (b_1 \vee \dots \vee b_k)$$

where $a_1, \dots, a_j$ is a subsequence of $b_1, \dots, b_k$ up to variable renaming. This condition certainly ensures that $Q \implies P$, meaning P is redundant if Q is already known to be invariant. Our implementation also has limited support for substituting universally-quantified variables for existential ones.

Efficient Implementation. To implement this filter efficiently, we store a set of sequences representing known invariants and query if any sequence in this set is a subsequence of a candidate sequence. We use a trie [10] for efficient queries.

4.5 Additional Optimizations

4.5.1 Minimizing Models

When extracting a counterexample from a failed VC check, we can either take the first model produced by the SMT solver, whatever it may be, or we can try to make the model as small as possible. Smaller models are faster to evaluate with our filters, but come at the cost of additional SMT queries.

In our *minimal-models* optimization, we always attempt to find a *minimal* model that satisfies a given SMT query, i.e., a model where there is no other satisfying model with a smaller domain. To bound the cost of the SMT queries made to check minimality, we apply an aggressive time limit to each query.

This is an essential optimization for complex protocols; without it, SWISS takes significantly longer (§5.3.5).

4.5.2 Parallelism

Since our algorithm is based on exhaustive search, we expect it to be parallelizable. To parallelize across n threads we do the following: for each run of either the Breadth or Finisher algorithm, we split the space ($\mathcal{B}$ or $\mathcal{F}$) into n parts, each randomly permuted. The random permutation attempts to ensure that each thread has a roughly equal amount of work. We then run our algorithm on each partition independently and combine the results. In the future, we plan to explore a more complex approach in which the threads communicate at a finer granularity, e.g., sharing counterexample filters and invariants as they are discovered.

4.6 Failed Optimizations

In the interest of full disclosure, and to save others unnecessary work, in Appendix A we briefly summarize three optimizations we implemented and evaluated, only to find that they provide little benefit or actively hurt performance.

5 Evaluation

Our evaluation seeks to answer two key questions:

- How does SWISS compare to prior work (§5.2)?
- How effective are SWISS's various optimizations (§5.3)?

5.1 Experimental Setup and Implementation Details

Benchmark Suite. To evaluate SWISS, we apply it to a test suite of well-established protocols from three sources: **(i)** the I4 benchmarks [34], **(ii)** the benchmarks from Koenig et al.'s work on first-order-logic separators [27] (henceforth, FOL), and **(iii)** six Paxos variants developed by Padon et al. [41].

Setup. All experiments are conducted on 8-core machines with Intel i9-9900K CPU processors at 3.60GHz, with 125GB of memory, running Ubuntu 19.10, and using a timeout of 6 hours. Unless specified, we run SWISS with 8 threads using automatic template generation (§4.1.1) and the following optimizations from §4: **(i)** symmetry breaking, **(ii)** cex filtering, **(iii)** hybrid SMT solvers, **(iv)** filtering redundant invariants, **(v)** model minimization, and **(vi)** parallelism. For template auto-generation, by default we use parameters (d,k,q,m) (§4.1.1) of $(1,3,1,5)$ for Breadth and $(2,6,1,6)$ for Finisher,

However, these choices were not ideal for all benchmarks. In particular, if we found that a benchmark completed in under six hours without proving the safety condition, we enlarged the $\mathcal{F}$ space by increasing k, and then increasing q. On the other hand, if a benchmark timed out after six hours without completing the breadth phase, we made the $\mathcal{B}$ space smaller by decreasing its parameters. See §5.2.1.

Implementation. We implement SWISS in approximately 14,000 lines of C++ code. To check the verification conditions, we use the Z3 SMT solver [11] version 4.8.8 with default settings. If Z3 times out after 45 seconds, then we use the CVC4 [2] SMT solver version 1.8-prerelease with the `--finite-model-find` option, an option tuned for finding finite models [44]. We found that for some satisfiable instances, Z3 would often take a very long time to return a model, whereas CVC4, *with this specific option*, was much more efficient. When applying model minimization, we set a time limit of 45 seconds; if the time limit is reached, we use the current partially minimized model.

5.2 Top-Level Protocol Results

Table 1 summarizes the results of running SWISS on our benchmark suite. It also compares SWISS's performance with that of the two most closely related systems (see §6 for details),

Source	Benchmark	∃?	I4 [34]	FOL [27]	SWISS	Partial	t_B	t_F	n_B	m_n	m_{n-1}
§2.2	sdl		10 s.	7 s.	5 s.		2 s.	2 s.	2	5	2
[34]	ring-election		3 s.	16 s.	11 s.		11 s.	-	3	3	3
	learning-switch-ternary		8 s.	▨	195 s.		195 s.	-	2	3	3
	lock-server-sync		0.1 s.	1 s.	0.2 s.		0.2 s.	-	1	2	
	two-phase-commit		2 s.	15 s.	6 s.		6 s.	-	1	3	3
	chain		17 s.	▨	▨	(6 / 7)				7	4
	chord		506 s.	7 s.	▨	(7 / 10)				6	4
	distributed-lock		131 s.	▨	▨	(1 / 4)				12	12
[27]	toy-consensus-forall		▨	4 s.	3 s.		3 s.	-	2	3	3
	consensus-forall		▨	1047 s.	29 s.		29 s.	-	3	3	3
	consensus-wo-decide		▨	23 s.	18 s.		18 s.	-	3	3	3
	learning-switch-quad		▨	▨	959 s.		223 s.	736 s.	2	4	4
	lock-server-async		0.4 s.	2 s.	3684 s.		1 s.	3682 s.	1	8	
	sharded-kv		1.0 s.	7 s.	4024 s.		2 s.	4021 s.	2	8	2
	ticket		▨	60 s.	▨	(5 / 9)				6	6
	toy-consensus-epr	✓	▨	22 s.	2 s.		2 s.	-	2	3	3
	consensus-epr	✓	▨	377 s.	20 s.		20 s.	-	3	3	3
	client-server-ae	✓	▨	303 s.	3 s.		2 s.	0.5 s.	2	3	
	client-server-db-ae	✓	▨	2739 s.	24 s.		21 s.	2 s.	4	3	3
	sharded-kv-no-lost-keys	✓	▨	1 s.	0.9 s.		0.9 s.	-	1	2	
	hybrid-reliable-broadcast	✓	▨	791 s.	▨	(1 / 7)				7	6
[41]	paxos	✓	▨	▨	15950 s.		803 s.	15146 s.	3	6	3
	flexible-paxos	✓	▨	▨	18232 s.		239 s.	17993 s.	3	6	3
	multi-paxos	✓	▨	▨	▨	(6 / 11)				6	4
	multi-paxos*	✓	▨	▨	984 s.		589 s.	395 s.	3	6	4
	fast-paxos	✓	▨	▨	▨	(8 / 12)				8	8
	stoppable-paxos	✓	▨	▨	▨	(6 / 14)				6	6
	vertical-paxos	✓	▨	▨	▨	(14 / 17)				16	6

Table 1: **Comparison with Prior Work.** ∃? indicates that the human-written invariant uses an existential. The time for SWISS is broken down into the time for Breadth (t_B) and Finisher (t_F), with the latter omitted when synthesis succeeds during the Breadth phase. n_B denotes number of iterations of Breadth. Each SWISS result is the median of five runs. For benchmarks where SWISS times out after 6 hours, we report its partial success (§5.2.2) instead of time. Shaded boxes indicate the system did not produce any invariants. In the multi-paxos* benchmark, the user provides a correct template for SWISS as guidance. The columns m_n and m_{n-1}, discussed in §5.3.4, summarize stats related to the sizes of invariants.

namely the I4 [34] and FOL [27] tools. Below, we first discuss the comparative results, followed by SWISS-specific analysis.

5.2.1 Comparative Results

We analyze our protocol results by bucketing them into coarse-grained categories. Note that I4 can only generate invariants containing universal quantifiers, and hence cannot succeed for benchmarks that require existentially quantified invariants.

Paxos Variants. Unlike prior work, SWISS automatically finds all of the invariants for Paxos and Flexible Paxos, which were previously painstakingly constructed by hand [41]. Furthermore, it succeeds at Multi-Paxos if the user provides the correct templates; however, when we attempted to use the automatically-enumerated templates, we found that Breadth does not complete in time. All three protocols require Finisher to find an invariant with six terms and an existential.

Unfortunately, neither I4, FOL, nor SWISS can prove the safety of Fast Paxos [30], Stoppable Paxos [35], or Vertical Paxos [32]. Among the known, handwritten invariants, Fast Paxos and Vertical Paxos each have two 8-term invariants; Stoppable Paxos has three 6-term invariants. SWISS cannot currently synthesize invariants this large—indeed, as it stands, it would need several orders of magnitude more compute time to handle even a single 8-term invariant—and we did not observe it finding equivalent, smaller invariants for these protocols. However, Table 1 shows that SWISS is at least able to synthesize partial invariants equivalent to many of the handwritten ones (discussed in more detail below — §5.2.2).

Finally, we note one interesting occurrence during our preliminary testing: we initially found that SWISS succeeded on Fast Paxos far too quickly and with invariants that looked at a glance to be far too strong. This allowed us to identify a typo in the spec which caused an action to never be enabled.

Mutual-Exclusion Protocols. Protocols such as sdl, distributed-lock, lock-server-sync, lock-server-async, and sharded-kv have safety conditions asserting mutual-exclusion properties, such as those of locks. The invariants for these protocols tend to be large conjuncts of smaller, *mutually* inductive invariants. SWISS struggles with these, since it is forced to discover the entire collection at once, as none are inductive individually. For some, Finisher succeeds by inferring all of these mutually inductive invariants as a single large invariant. For sharded-kv and lock-server-async, we needed to increase the maximum value of k from our default configuration. For distributed-lock, the invariant was simply too large for SWISS to find. FOL was also unable to solve this benchmark, although I4 solves it quite handily, as it was able to infer the mutually inductive invariants on a small finite instance of the problem. Finally, for sharded-kv-no-lost-keys we needed to use $q = 2$ to allow more existentials.

Learning Switch. We have two different benchmarks called "learning-switch," one from I4 and one from FOL. We found that unlike I4 and FOL, SWISS succeeds on learning-switch-quad quite handily. In learning-switch-ternary, we found that Breadth takes far too long in the default configuration, due in part to a large number of redundant invariants not filtered by our FastImplies test and thus requiring SMT calls to identify as redundant. We reduced the size of $\mathcal{B}$ by configuring $q = 0$ (i.e., searching for universal invariants only) for this one benchmark, allowing SWISS to also be able to complete it efficiently.

5.2.2 SWISS-Specific Analysis

Invariants. In some cases, SWISS uncovered invariants that were simpler than the ones written by the original researchers. For instance, for the ring-election protocol, SWISS's Breadth algorithm identifies three 3-term invariants (9 terms total) similar to the ones from prior work [42]; however, if we run Finisher on its own rather than the full SWISS algorithm, then it instead generates a single 5-term inductive invariant. For the Paxos protocol, we initially expected that the Breadth algorithm would need to run with $k = 5$; however, we found that SWISS succeeded even with Breadth at $k = 3$. This shows that mechanized search can find simpler invariants that may be missed by trained researchers.

Partial Progress. When an execution of SWISS exceeds its time limit, it still generates many invariants which may be useful. To measure how successful this "partial progress" is, we computed how many of the handwritten invariants SWISS finds. More precisely, we count how many of these invariants are logical implications of SWISS's invariants. These results are shown in Table 1. Notice that in many cases, SWISS finds a majority of the invariants; e.g., for our largest benchmark, Vertical Paxos, SWISS finds 79%. Furthermore, for the chain benchmark, which has two safety conditions (one for linearizability and one for atomicity), SWISS solves the latter.

We do not report any partially solved invariants for the other tools in our comparison. For one, the IC3 approach does not immediately allow extraction of any inductive invariants; a partially-completed IC3 execution would provide a different kind of information: constraints on the possible states of a system after a finite number of transitions. In the case of I4, we did not find any case where it constructed a nontrivial inductive invariant for any benchmark where it did not succeed.

5.3 Understanding SWISS

To better understand SWISS, we evaluate the effectiveness of our various design choices.

5.3.1 The Utility of Combining Breadth and Finisher

In §3, we argue that Breadth and Finisher target different kinds of invariants and hence are more effective when used together. To validate this claim, we re-ran SWISS on our benchmark suite, first using only Breadth, and then using only Finisher. In both cases, we limited their execution time to that taken by the full SWISS algorithm.

Compared to the 19 benchmarks solved by the full SWISS algorithm, the Breadth algorithm alone solves only 10, while Finisher alone solves only 7.

5.3.2 Execution Times of Breadth Versus Finisher

In Table 1, Breadth is fairly quick, always less than 15 minutes, compared to Finisher which can vary from quite quick (a few seconds) to several hours. In fact, this is not an accident and follows directly from the fact that the space $\mathcal{B}$ is *chosen* to be small, whereas Finisher is designed to keep going until it finds a solution or times out. Thus, the runtime of Finisher depends on how large that solution turns out to be.

5.3.3 Impact of User Guidance

While SWISS is designed to automatically search through the space of invariant templates, users can provide more specific guidance via one or more templates. For example, for Paxos, considering the Finisher portion, the user might suggest a template like (1) or (2) in Table 2. Compared to searching the automatically generated space of templates, this guidance cuts the search space by 75% and as a result, completes in 20 minutes, a 13.4× speedup.

Of course, the user may erroneously suggest a template not containing any useful invariant. To evaluate the impact of such a mistake, we ran SWISS on a template (3) of comparable size to the "correct" templates, as well as some which are much smaller (4-6). The results suggest the user can afford to make some incorrect guesses and still outperform the full auto mode.

Finally, template (7) is the largest template generated automatically with our default configuration. It takes significantly longer, indicating the importance of exploring templates in increasing size order when using auto mode.

5.3.4 Is it a Small World?

Our Small World Hypothesis holds that many of the protocols we care about can be solved using a sequence of invariants $I_1, \ldots, I_n$, which are *individually* concise. We analyze this hypothesis by measuring the size of the invariants in each of our benchmark protocols. This, in turn, helps us understand

#	Template	Inv?	Size	Time (s)
1	$\forall r_1, r_2 : \text{round}, v_1, v_2 : \text{value}, q_1 : \text{quorum}.\ \exists n_1 : \text{node}.\ *$	✓	232,460,599,445	2036
2	$\forall r_1, r_2 : \text{round}, v_1, v_2 : \text{value}.\ \exists q_1 : \text{quorum}.\ \forall n_1 : \text{node}.\ *$	✓	232,460,599,445	4072
3	$\forall r_1, r_2 : \text{round}, v_1, v_2 : \text{value}, q_1 : \text{quorum}.\ \forall n_1 : \text{node}.\ *$		232,460,599,445	3547
4	$\forall r_1 : \text{round}, v_1, v_2, v_3 : \text{value}.\ \exists n_1 : \text{node}.\ *$		26,863,311,982	408
5	$\forall r_1 : \text{round}, v_1 : \text{value}, n_1, n_2, n_3, n_4 : \text{node}.\ *$		76,397,976,796	1015
6	$\forall r_1, r_2 : \text{round}, v_1 : \text{value}, q_1, q_2 : \text{quorum}.\ \exists n_1 : \text{node}.\ *$		39,834,946,595	557
7	$\forall r_1, r_2 : \text{round}, v_1, v_2 : \text{value}, n_1, n_2 : \text{node}.\ *$		2,621,795,213,086	47231

Table 2: **User-Provided Templates.** Experiments running Finisher on Paxos with the given template as input. Each experiment starts with many invariants provided as input, simulating the scenario that Breadth has already completed. The 'Inv?' column indicates whether the template contains an invariant that proves the safety condition. Time is given in seconds. Each experiment is run to completion, exploring the entire space, even if an invariant is found.

which protocols SWISS will likely succeed at by the degree to which they meet the Small World Hypothesis.

For a given protocol, we examine an invariant $I = I_1 \wedge \cdots \wedge I_n$ which proves the desired safety condition, chosen so that $I_1 \wedge \ldots \wedge I_j$ is inductive relative to the safety condition (as in Claim 1). The m_n column in Table 1 gives the maximum size of any invariant out of $I_1, \ldots, I_n$. Here, the "size" of a predicate is measured as its number of terms.

We do not claim that our numbers are the minimal possible—we simply use the smallest out of any I that we know of. These may be from invariants synthesized by SWISS itself; in other cases, we use human-determined invariants.

We can see that 25 out of our 27 benchmarks have $m_n \leq 8$, and 22 of them have even fewer. There were two exceptions, distributed-lock ($m_n = 12$) and vertical-paxos ($m_n = 16$). These two invariants are much bigger than any single invariant we have seen SWISS synthesize.

However, it may be worth noting that these larger invariants are conjuncts of smaller, mutually inductive invariants. For example, the 16-term invariant of vertical-paxos is actually the conjunct of two 8-term invariants (although these invariants are still on the larger side). Thus, these protocols would score much better on a weaker Small World Hypothesis, one where mutual invariants were counted separately. Other approaches may be able to take advantage of this.

To more fully understand SWISS's behavior, we also measure m_{n-1}, the maximum number of terms among $I_1, \ldots, I_{n-1}$. This statistic is interesting here because for SWISS to succeed in its current form, these $n-1$ invariants must be in $\mathcal{B}$.

To rough approximation, we see that SWISS can succeed approximately when $m_n \leq 6$ and $m_{n-1} \leq 3$. In some cases, SWISS does go beyond: some benchmarks succeed with up to $m_n = 8$, and SWISS can solve multi-paxos (where $m_{n-1} = 4$) if the search space is restricted in other ways.

5.3.5 Impact of Filtering

Table 3 shows the impact of each of our filtering stages on Paxos. Counterexample filtering drastically decreases the number of candidates which require an SMT call (i.e., those remaining after FastImplies). However, a vast number of candidates (the "Symmetries" column) must be processed by a counterexample-filter. How fast is such filtering?

	Baseline	Sym.	Cex filters	FastImpl	Inv.
$\mathcal{B}$	$820 \cdot 10^6$	$3 \cdot 10^6$	911,275	2,250	801
$\mathcal{F}$	$99 \cdot 10^{12}$	$232 \cdot 10^9$	155	155	5

Table 3: **Winnowing the Paxos Search Space**. Number of candidate predicates that remain *after* a given SWISS feature is applied. Runs are with a single thread over a single template.

To evaluate filtering efficiency, we measure the total time spent on filtering versus SMT inductivity checks in a Paxos benchmark. We find that Finisher (using Template (1) of Table 2) performs 155 inductivity checks via SMT. The average SMT call takes 96.5 ms, with a median of 7 ms and a 95th percentile of 55 ms. In contrast, filtering a single candidate takes 74 *nanoseconds* on average. Notably, both measures are important characteristics of SWISS, as some workloads are dominated by filtering and others by SMT calls (Appendix B.2).

5.3.6 Additional Analysis

See Appendix B for further analysis of SWISS's performance, e.g., the impact of optimizations, parallelism, and SMT calls.

6 Related Work

6.1 Verifying Distributed Systems

The research community has long recognized the challenges of designing correct distributed systems. Manually written proofs [25, 31] and model checking [23, 26, 56] increase assurance, but struggle with practical distributed systems [4].

Recent work applies general-purpose software-verification tools to the verification of distributed systems [21, 46, 55]. These tools offer flexibility at the price of substantial human effort. For example, verifying Raft required over 50,000 lines of Coq proof for the protocol and its 520-line implementation [55], and Hawblitzel et al. used 12,000 lines of proof for the safety and liveness proofs of their Paxos protocol [21].

These human costs motivate the search for domain-specific languages (DSLs) and tools that reduce proof effort by restricting the class of encodable systems [12, 13, 36, 53]. For example, tools based on the heard-of model [12, 13, 36] need a run-time system to bridge the gap between an asynchronous network and the synchronous semantics assumed by the verification tool, which can lead to performance bottlenecks. Similarly, pretend synchrony [53] precludes classic optimizations, e.g., request batching in Paxos implementations.

All the works above, even the DSLs, rely on the developer to intuit invariants, which can be hard even for experienced researchers [34]. Recent work tries to reduce this cost via a restricted logic (EPR – §2.4) which makes invariant checking decidable; i.e., given a correct invariant, no further human work is needed. Even within EPR, *finding* the invariant remains undecidable [40], so Padon et al.'s IVy tool [42] interactively aids the developer: IVy iteratively checks if a candidate invariant is inductive. If not, IVy presents a concrete counterexample, and the developer strengthens the candidate to eliminate it. This repeats until she derives an inductive invariant.

In contrast, Ma et al.'s I4 tool [34] aims to be fully automatic. I4 first runs a custom model checker [19] on an artificially small example of the protocol (e.g., with two nodes) to produce an invariant for the small system. I4 then attempts to generalize the invariant to the unbounded setting. When it succeeds, I4 requires no human intervention. In practice, Ma et al. report manually specifying concrete bounds for all of their benchmarks (to avoid exhaustive parameter searches) and concretizations of certain variables for several benchmarks. Ultimately, I4 is limited by the abilities of its model checker, which does not support existential quantifiers (which §5.2 shows rules out a wide swath of protocols), and is unable to scale to more complex protocols like Paxos. In such cases, the developer is left with little recourse. However, I4 is frequently faster than SWISS and able to synthesize larger invariants for protocols where universally quantified invariants exist.

In recent work, Koenig et al. (FOL) [27] develop an algorithm capable of synthesizing invariants containing existentials. Their algorithm relies on the IC3/PDR algorithm [6, 14] for constructing invariants incrementally. Like SWISS, it iteratively produces counterexamples, but it uses those counterexamples as constraints in a SAT encoding of predicates to be synthesized. SWISS verifies protocols, like Paxos, that FOL does not and verifies some faster than FOL. The reverse is also true, suggesting some complementarity of the approaches.

6.2 General-Purpose Invariant Synthesis

Extensive research [7–9, 15, 17, 18, 20, 37, 39, 49, 57] studies loop-invariant inference for proving program correctness, but this remains challenging. Most approaches are limited to single-loop programs; only a few handle multiple loops or existential invariants. Approaches include abstract interpretation [8], interpolation [37], IC3 [16, 17], templates and constraint solvers [20], counterexample-guided invariant generation (CEGIR) [18, 39, 47], trace analysis [9, 15], and machine learning [48, 49, 57].

More closely related work uses templates [7, 20] to restrict the search to invariants of a given shape. In contrast to these approaches, we automatically construct a large set of templates and search for invariants of larger sizes. CEGIR approaches [18, 39, 47] use enumeration and exploit the fact that guessing a candidate and checking if it is invariant is easier than inferring a loop invariant directly from code. They often employ dynamic analyses to infer candidates from execution traces and use a verifier to check invariant validity. The idea of learning from counterexamples has also been applied to program synthesis in the form of counterexample-guided inductive synthesis (CEGIS) [51], where the synthesizer generates a candidate program and the verifier uses the failed cases to prune the search space. SWISS's approach is inspired by techniques from search-based program synthesis [1] and the CEGIS framework. Although CEGIS is a general framework, it cannot be used as a black-box since it requires a custom synthesizer, verifier, and learner for each domain. SWISS differs from prior approaches in how it uses the counterexamples to prune the search space (§4.2) and how it applies a CEGIS-style approach to infer more complex invariants than prior work.

Program sketching [50] allows a programmer to sketch a program, i.e., write a program with "holes." A synthesizer fills the holes such that a specification is satisfied. In SWISS, the user can similarly provide templates to restrict the search, but even if such a template is not provided, SWISS can automatically generate a set of templates and search all of them.

7 Conclusions and Future Work

We explore the hypothesis that the safety of most distributed systems can be proven via relatively small invariants (or conjunctions thereof), using our system SWISS, which incorporates novel optimizations to efficiently search the space of candidate invariants. We find that in many cases our hypothesis holds, and SWISS is able to automatically prove their safety, including several, such as Paxos, beyond the reach of prior work. Our results leave open the question of inferring large, mutually inductive invariants. They also illustrate that SWISS and its most recent predecessors often have complementary coverage of the benchmarks. Exploring ways to combine the strengths of each is an intriguing direction for future work.

8 Acknowledgements

We thank Jay Lorch, Jon Howell, the NSDI reviewers, and our shepherd, Siddhartha Sen, for useful paper feedback. We also thank Jean Yang and Mickaël Laurent for many helpful conversations and early explorations. We thank the Pittsburgh Supercomputing Center for providing computation resources.

This work was funded in part by the Alfred P. Sloan Foundation, a Google Faculty Fellowship, the NSF/VMware Partnership on Software Defined Infrastructure as a Foundation for Clean-Slate Computing Security (SDI-CSCS) program under Award No. CNS-1700521, and the NSF under grant CCF-2015445.

References

[1] Alur, R., Singh, R., Fisman, D., and Solar-Lezama, A. Search-based program synthesis. *Commun. ACM 61*, 12 (2018), 84–93.

[2] Barrett, C., Conway, C. L., Deters, M., Hadarean, L., Jovanovi'c, D., King, T., Reynolds, A., and Tinelli, C. CVC4. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806 of *Lecture Notes in Computer Science*, Springer, pp. 171–177.

[3] Bolosky, W. J., Douceur, J. R., and Howell, J. The Farsite project: A retrospective. *SIGOPS Oper. Syst. Rev. 41*, 2 (Apr. 2007), 17–26.

[4] Bolosky, W. J., Douceur, J. R., and Howell, J. The Farsite project: a retrospective. *ACM SIGOPS Operating Systems Review 41 (2)* (April 2007).

[5] Bornholt, J., and Torlak, E. Synthesizing memory models from framework sketches and litmus tests. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)* (2017).

[6] Bradley, A. R. SAT-based model checking without unrolling. In *Proceedings of Verification, Model Checking, and Abstract Interpretation (VMCAI)* (2011).

[7] Colón, M., Sankaranarayanan, S., and Sipma, H. Linear invariant generation using non-linear constraint solving. In *Proceedings of the International Conference on Computer Aided Verification (CAV)* (2003), W. A. H. Jr. and F. Somenzi, Eds., vol. 2725 of *Lecture Notes in Computer Science*, Springer, pp. 420–432.

[8] Cousot, P., and Halbwachs, N. Automatic discovery of linear restraints among variables of a program. In *Proceedings on the ACM Symposium on Principles of Programming Languages (POPL)* (1978), A. V. Aho, S. N. Zilles, and T. G. Szymanski, Eds., ACM Press, pp. 84–96.

[9] Danese, A., Piccolboni, L., and Pravadelli, G. A parallelizable approach for mining likely invariants. In *Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis* (2015), CODES '15, IEEE Press, p. 193–201.

[10] De La Briandais, R. File searching using variable length keys. In *Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference* (New York, NY, USA, 1959), IRE-AIEE-ACM '59 (Western), Association for Computing Machinery, p. 295–298.

[11] De Moura, L., and Bjørner, N. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (2008), Springer, pp. 337–340.

[12] DrăźGoi, C., Henzinger, T. A., Veith, H., Widder, J., and Zufferey, D. A logic-based framework for verifying consensus algorithms. In *Proceedings of Computer Aided Verification (CAV)* (2014).

[13] DrăźGoi, C., Henzinger, T. A., and Zufferey, D. PSync: A partially synchronous language for fault-tolerant distributed algorithms. In *ACM Symposium on Principles of Programming Languages (POPL)* (2016).

[14] Eén, N., Mishchenko, A., and Brayton, R. Efficient implementation of property directed reachability. In *Proceedings of the Conference on Formal Methods in Computer-Aided Design (FMCAD)* (2011).

[15] Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S., and Xiao, C. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program. 69*, 1–3 (Dec. 2007), 35–45.

[16] Ezudheen, P., Neider, D., D'Souza, D., Garg, P., and Madhusudan, P. Horn-ICE learning for synthesizing invariants and contracts. *Proc. ACM Program. Lang. 2*, OOPSLA (2018), 131:1–131:25.

[17] Garg, P., Löding, C., Madhusudan, P., and Neider, D. ICE: A robust framework for learning invariants. In *Proceedings of the International Conference on Computer Aided Verification (CAV)* (2014), A. Biere and R. Bloem, Eds., vol. 8559 of *Lecture Notes in Computer Science*, Springer, pp. 69–87.

[18] Garg, P., Neider, D., Madhusudan, P., and Roth, D. Learning invariants using decision trees and implication counterexamples. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (2016), R. Bodík and R. Majumdar, Eds., ACM, pp. 499–512.

[19] Goel, A., and Sakallah, K. A. Model checking of verilog RTL using IC3 with syntax-guided abstraction. In *Proceedings of the NASA Formal Methods Symposium (NFM)* (2019), J. M. Badger and K. Y. Rozier, Eds., vol. 11460 of *Lecture Notes in Computer Science*, Springer, pp. 166–185.

[20] Gupta, A., and Rybalchenko, A. InvGen: An efficient invariant generator. In *Proceedings of the International Conference on Computer Aided Verification (CAV)* (2009), A. Bouajjani and O. Maler, Eds., vol. 5643 of *Lecture Notes in Computer Science*, Springer, pp. 634–640.

[21] Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J. R., Parno, B., Roberts, M. L., Setty, S., and Zill, B. IronFleet: Proving safety and liveness of practical distributed systems. *Commun. ACM 60*, 7 (June 2017), 83–92.

[22] Howard, H., Malkhi, D., and Spiegelman, A. Flexible Paxos: Quorum intersection revisited, 2016.

[23] Jones, E. Model checking a Paxos implementation. http://www.evanjones.ca/model-checking-paxos.html, 2009.

[24] Joshi, R., Lamport, L., Matthews, J., Tasiran, S., Tuttle, M., and Yu, Y. Checking cache-coherence protocols with TLA+. *Journal of Formal Methods in System Design 22* (March 2003), 125–131.

[25] Kellomäki, P. An annotated specification of the consensus protocol of Paxos using superposition in PVS. Tech. Rep. 36, Tampere University of Technology, 2004.

[26] Killian, C. E., Anderson, J. W., Braud, R., Jhala, R., and Vahdat, A. M. Mace: Language support for building distributed systems. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)* (2007).

[27] Koenig, J. R., Padon, O., Immerman, N., and Aiken, A. First-order quantified separators. In *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)* (2020), A. F. Donaldson and E. Torlak, Eds., ACM, pp. 703–717.

[28] Lamport, L. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems 16*, 3 (May 1994).

[29] Lamport, L. The part-time parliament. *ACM Trans. Comput. Syst. 16*, 2 (May 1998), 133–169.

[30] Lamport, L. Fast Paxos. *Distributed Computing 19* (October 2006), 79–103.

[31] Lamport, L. Byzantizing Paxos by refinement. In *Proceedings of the International Conference on Distributed Computing (DISC)* (2011).

[32] Lamport, L., Malkhi, D., and Zhou, L. Vertical paxos and primary-backup replication. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)* (2009), S. Tirthapura and L. Alvisi, Eds., ACM, pp. 312–313.

[33] Lewis, H. R. Complexity results for classes of quantificational formulas. *J. Comput. Syst. Sci. 21* (1980), 317–353.

[34] Ma, H., Goel, A., Jeannin, J., Kapritsos, M., Kasikci, B., and Sakallah, K. A. I4: incremental inference of inductive invariants for verification of distributed protocols. In *Proceedings of the ACM Symposium on Operating Systems Principles, (SOSP)* (2019), T. Brecht and C. Williamson, Eds., ACM, pp. 370–384.

[35] Malkhi, D., Lamport, L., and Zhou, L. Stoppable Paxos. Tech. Rep. MSR-TR-2008-192, April 2008.

[36] Maric, O., Sprenger, C., and Basin, D. A. Cutoff bounds for consensus algorithms. In *Proceedings of Computer Aided Verification (CAV)* (2017).

[37] McMillan, K. L. Quantified invariant generation using an interpolating saturation prover. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (2008), C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963 of *Lecture Notes in Computer Science*, Springer, pp. 413–427.

[38] Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., and Deardeuff, M. How Amazon web services uses formal methods. *Commun. ACM 58*, 4 (Mar. 2015), 66–73.

[39] Nguyen, T., Antonopoulos, T., Ruef, A., and Hicks, M. Counterexample-guided approach to finding numerical invariants. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (FSE)* (2017), E. Bodden, W. Schäfer, A. van Deursen, and A. Zisman, Eds., ACM, pp. 605–615.

[40] Padon, O., Immerman, N., Shoham, S., Karbyshev, A., and Sagiv, M. Decidability of inferring inductive invariants. *SIGPLAN Not. 51*, 1 (Jan. 2016), 217–231.

[41] Padon, O., Losa, G., Sagiv, M., and Shoham, S. Paxos made EPR: Decidable reasoning about distributed protocols. *Proc. ACM Program. Lang. 1*, OOPSLA (Oct. 2017).

[42] Padon, O., McMillan, K. L., Panda, A., Sagiv, M., and Shoham, S. Ivy: Safety verification by interactive generalization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2016), Association for Computing Machinery, p. 614–630.

[43] Piskac, R., de Moura, L., and Bjørner, N. Deciding effectively propositional logic with equality. Tech. Rep. MSR-TR-2008-181, December 2008.

[44] Reynolds, A., Tinelli, C., Goel, A., and Krstic, S. Finite model finding in SMT. In *Proceedings of the International Conference on Computer Aided Verification (CAV)* (2013), Springer, pp. 640–655.

[45] Schiper, N., Rahli, V., Renesse, R. V., Bickford, M., and Constable, R. L. Developing correctly replicated databases using formal tools. In *Proceedings of the Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2014), IEEE Computer Society, p. 395–406.

[46] Schiper, N., Rahli, V., van Renesse, R., Bickford, M., and Constable, R. Developing correctly replicated databases using formal tools. In *Proceedings of the IEEE/IFIP Conference on Dependable Systems and Networks (DSN)* (June 2014).

[47] Sharma, R., Gupta, S., Hariharan, B., Aiken, A., Liang, P., and Nori, A. V. A data driven approach for algebraic loop invariants. In *Proceedings of the European Symposium on Programming Languages and Systems (ESOP)* (2013), M. Felleisen and P. Gardner, Eds., vol. 7792 of *Lecture Notes in Computer Science*, Springer, pp. 574–592.

[48] Si, X., Dai, H., Raghothaman, M., Naik, M., and Song, L. Learning loop invariants for program verification. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada* (2018), S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., pp. 7762–7773.

[49] Si, X., Naik, A., Dai, H., Naik, M., and Song, L. Code2Inv: A deep learning framework for program verification. In *Proceedings of the International Conference on Computer Aided Verification (CAV)* (2020), S. K. Lahiri and C. Wang, Eds., vol. 12225 of *Lecture Notes in Computer Science*, Springer, pp. 151–164.

[50] Solar-Lezama, A. Program sketching. *Int. J. Softw. Tools Technol. Transf. 15*, 5-6 (2013), 475–495.

[51] Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S. A., and Saraswat, V. A. Combinatorial sketching for finite programs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2006), J. P. Shen and M. Martonosi, Eds., ACM, pp. 404–415.

[52] Trakhtenbrot, B. The impossibility of an algorithm for the decidability problem on finite classes. In *Proceedings of the USSR Academy of Sciences* (1950).

[53] v. Gleissenthall, K., Kici, R. G., Bakst, A., Stefan, D., and Jhala, R. Pretend synchrony: Synchronous verification of asynchronous distributed programs. In *ACM Symposium on Principles of Programming Languages (POPL)* (2019).

[54] Wilcox, J. R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M. D., and Anderson, T. Verdi: A framework for implementing and formally verifying distributed systems. *SIGPLAN Not. 50*, 6 (June 2015), 357–368.

[55] Woos, D., Wilcox, J. R., Anton, S., Tatlock, Z., Ernst, M. D., and Anderson, T. Planning for change in a formal verification of the raft consensus protocol. In *ACM Conference on Certified Programs and Proofs (CPP)* (Jan. 2016).

[56] Yang, J., Chen, T., Wu, M., Xu, Z., Liu, X., Lin, H., Yang, M., Long, F., Zhang, L., and Zhou, L. MODIST: Transparent model checking of unmodified distributed systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (April 2009).

[57] Yao, J., Ryan, G., Wong, J., Jana, S., and Gu, R. Learning nonlinear loop invariants with gated continuous logic networks. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2020), PLDI 2020, Association for Computing Machinery, p. 106–120.

[58] Yu, Y. Using formal specifications to monitor and guide simulation: Verifying the cache coherence engine of the Alpha 21364 microprocessor. In *Proceedings of the IEEE International Workshop on Microprocessor Test and Verification (MTV)* (2002), Institute of Electrical and Electronics Engineers, Inc.

A Failed Optimizations

In the interest of full disclosure, and to save others unnecessary work, we briefly summarize three optimizations we implemented and evaluated, only to find that they provide little benefit or actively hurt performance.

A.1 Formula Synthesis

SWISS's current implementation explicitly enumerates and evaluates candidate invariants. Initially, however, we adopted a strategy from a prior system, MemSynth [5], which synthesizes memory models from a small collection of examples Instead of explicitly enumerating formulas, MemSynth encodes the desired shape of candidate formulas as constraints on its SMT queries. In our context, this means creating an SMT query that says, "Find a formula that satisfies this template and complies with our accumulated counterexamples."

However, our early experiments showed that synthesizing the formula via a solver was considerably slower than our combination of a custom enumerator and counterexample filters.

A.2 Bounded Model Checking

Prior work on finding invariants for distributed systems [42], found some benefit from using bounded model checking to try to quickly rule out candidate invariants. In our context, this means checking not just the condition $INIT \implies P$, but whether there are any violations of P in states reachable after taking a fixed number of steps from an initial state. If such a violation existed on a model M, we can use the counterexample filter True(M). Likewise, to build False filters, we consider states a fixed number of steps away from violating safety.

The hypothesis for this optimization is that it would produce more and "higher quality" filters to rule out future candidates. To pay off, these savings must offset the cost of the additional SMT calls that compute the bounded model checks. Sadly, this optimization rarely boosts performance significantly (§5.3.5).

A.3 Aggressively Accumulating Invariants

As it executes, Breadth finds predicates that are invariant with respect to the input invariants $I_1, \ldots, I_n$; these new invariants are fed into the next iteration of Breadth. This suggests an obvious improvement: treat newly found invariants as input invariants immediately in order to uncover even more invariants than the ones Breadth is guaranteed to find, leading to fewer total loops. We call this variation BreadthAccumulative.

However, this variant introduces several complications. Most critically, it interferes with the FastImplies optimization. For example, suppose we process predicates $f, g, h_1, \ldots, h_n$ in order and **(i)** g is invariant; **(ii)** f is invariant with respect to g; and **(iii)** FastImplies(f, h_i) holds for all i. BreadthAccumulative, would pass over f (not invariant), then find and add g, causing all h_i to become invariant. Since the h_i are not filtered out by the FastImplies check, BreadthAccumulative's aggressive addition of the h_i causes the number of invariants to explode. In contrast, Breadth would only add g at the end; then in the next call to Breadth, f would be added to allInv, excluding the h_i via FastImplies.

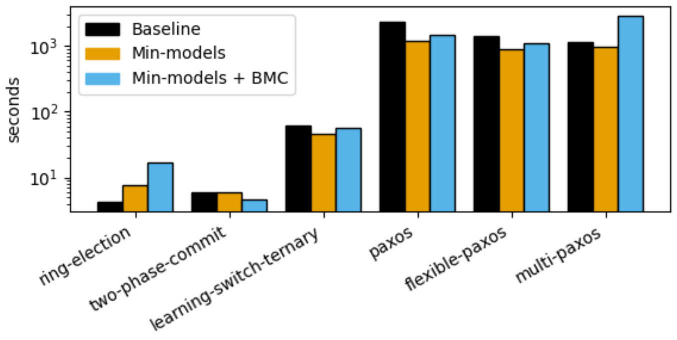


Figure 5: **Optimization Impact**. We evaluate our model-minimization and bounded-model-checking optimizations. To save time, these experiments use pre-specified templates rather than template auto-generation. Note the log scale of the y-axis.

To prevent BreadthAccumulative from adding such spurious invariants, we added a *strengthening* step, where the first h_i found would be strengthened to f. However, strengthening comes at a cost, and in the end, we found the BreadthAccumulative optimization to be unhelpful (Figure B.2).

B Additional Evaluation

In this section, we provide some additional measurements of the impact of SWISS's design decisions.

B.1 Impact of Optimizations

Model Minimization. To evaluate the effectiveness of model minimization (§A.2), we measure several benchmarks with and without it (Figure 5). While it adds some overhead for simple protocols, it helps significantly for more complex protocols; in the best case, we found that it improved the Flexible-Paxos experiment by 1.6×.

Bounded Model Checking. For our BMC variant (§A.2), we again ran several experiments with and without it (Figure 5). However, the results show that the cost of the required SMT calls was not sufficiently offset by gains from "higher quality" filters. Hence we disable it in SWISS's default configuration.

B.2 Impact of Parallelism

To evaluate our parallelism strategy (§4.5.2), we measure runtimes with varying numbers of threads, studying Breadth and Finisher independently. For each run, we break down the running time of the longest-running thread in each iteration to see which components of the algorithm parallelize well. We run our experiments on the Paxos protocol.

For the purposes of this experiment, unlike in standard runs, we do not terminate the Finisher algorithm when it finds an invariant which proves the safety condition; instead, we let it search the entire search space $\mathcal{F}$. This removes variance

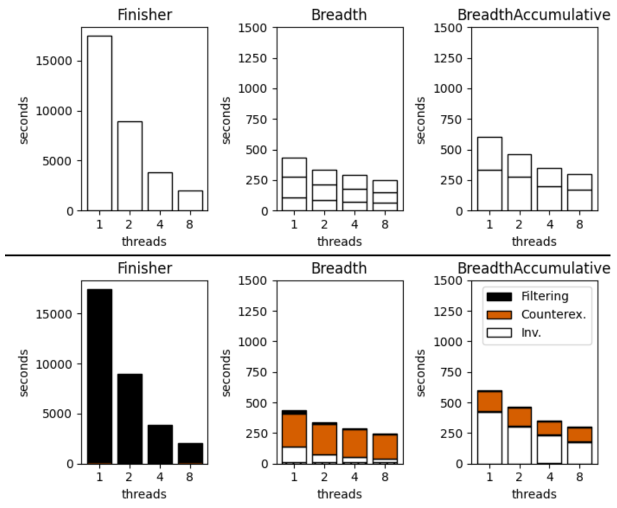


Figure 6: **Parallelism**. We measure time while varying the number of threads for Finisher, Breadth and BreadthAccumulative (§A.3) on Paxos. The top row shows the overall time, with Breadth broken down by iterations. Breadth terminates when the last (top) iteration fails to find any invariant, so it takes one less iteration than shown to find all invariants Breadth can find. In the bottom row, the runtimes are broken down into **(i)** filtering candidates with counterexamples, **(ii)** computing counterexamples of non-invariants, and **(iii)** processing candidates that are invariant.

from the random ordering of the search space, leading to more controlled experimental data.

Our results are shown in Figure 6. Finisher's runtime is dominated by enumerating and filtering, which splits fairly evenly across threads. Overall, SWISS on a single template is 2.0× faster with 2 threads than with 1 thread, and it is 8.6× faster with 8 threads than with 1.

Breadth, meanwhile, does not parallelize as well, since its runtime is dominated by time spent constructing counterexample filters via SMT calls, which is essentially a fixed cost per thread. At best, we saw a speedup of 1.7× with 8 threads.

We also evaluated BreadthAccumulative (§A.3) while varying the number of threads (Figure 6). Our hypothesis that BreadthAccumulative would require fewer iterations was confirmed: each run required only one iteration to find all invariants in $\mathcal{B}$ (plus one iteration to confirm no further invariants exist). By contrast, Breadth requires two iterations (plus one) on the same benchmark. However, BreadthAccumulative still performs worse than Breadth due its other costs (e.g., strengthening – §A.3).

B.3 Hard SMT instances

As described in §4.3, SMT queries are often rapid, but there are occasional outliers that slow down execution. To measure how problematic these outliers are, we measured the prevalence of hard instances, defined as any instance exceeding forty-five seconds and triggering our retry strategy. In particular, we measured the fraction of total computation time spent on these hard instances.

Among all protocols that SWISS was able to solve, this fraction was greatest for the paxos benchmark, which spent 10.2% of its computation time on hard instances, which accounted for 0.17% of its SMT instances.

However, among all protocols that SWISS was *not* able to solve, this fraction was greatest for the fast-paxos benchmark, which spent 98.7% of its computation time on hard instances, which accounted for 3.7% of its SMT instances. We currently do not have a good understanding of what makes this protocol's SMT instances difficult for SMT solvers, but the numbers suggest that improvement to SWISS's SMT strategy could make it much faster on harder protocols.

Avenir: Managing Data Plane Diversity with Control Plane Synthesis

Eric Hayden Campbell
Cornell

William T. Hallahan
Yale

Priya Srikumar
Cornell

Carmelo Cascone
ONF

Jed Liu
Intel

Vignesh Ramamurthy
Infosys

Hossein Hojjat*
Tehran & TeIAS

Ruzica Piskac
Yale

Robert Soulé
Yale

Nate Foster
Cornell

Abstract

The classical conception of software-defined networking (SDN) is based on an attractive myth: a logically centralized controller manages a collection of homogeneous data planes. In reality, however, SDN control planes must deal with significant diversity in hardware, drivers, interfaces, and protocols, all of which contribute to idiosyncratic differences in forwarding behavior that must be dealt with by hand.

To manage this heterogeneity, we propose Avenir, a synthesis tool that automatically generates control-plane operations to ensure uniform behavior across a variety of data planes. Our approach uses counter-example guided inductive synthesis and sketching, adding network-specific optimizations that exploit domain insights to accelerate the search. We prove that Avenir's synthesis algorithm generates correct solutions and always finds a solution, if one exists. We have built a prototype implementation of Avenir using OCaml and Z3 and evaluated its performance on realistic scenarios for the ONOS SDN controller and on a collection of benchmarks that illustrate the cost of retargeting a control plane from one pipeline to another. Our evaluation demonstrates that Avenir can manage data plane heterogeneity with modest overheads.

1 Introduction

The network control plane plays a similar role in modern systems as a classical OS kernel. It manages resources such as end-to-end forwarding paths, maps incoming traffic onto those paths, and enforces policy such as ensuring isolation between tenants in a public cloud.

One challenge that complicates the design of the control plane is dealing with data plane heterogeneity. Much as an OS kernel manages hardware resources for a variety of peripherals, the network control plane manages hardware resources for a variety of data planes. Most network operators purchase equipment from multiple manufacturers to avoid lock-in, which results in devices with heterogeneous feature sets, and even devices manufactured by the same vendor tend to evolve over time. This heterogeneity manifests as complexity throughout the control plane, appearing in low-level drivers and SDKs, device OSes (e.g., SONiC [42], FBOSS [5], Stratum [45]), higher-level APIs (e.g., OpenFlow [23], OpenConfig [30], P4Runtime [6]), and even network applications.

As an example, switches based on Broadcom ASICs such as Trident2, Tomahawk and Qumran-MX all expose an OpenFlow-like API to SDN controllers (or more precisely, the OF-DPA [32] abstraction). However, due to differences in the chips, the API behaves in subtly different ways on various devices. For instance, the Termination MAC table, which determines whether to route packets or bridge them, appears in all three devices but behaves differently on Trident2/Tomahawk versus Qumran-MX—the former supports matching on the ingress port while the latter does not. This discrepancy has led to bugs: before a special case was added to the ONOS controller, multicast traffic on Qumran-MX devices was flooded out on all ports rather than being forwarded to the proper multicast groups [34].

This anecdote is just one example of a more pervasive problem. The OF-DPA API specification [32] is more than 150 pages of English prose. The ONOS development team took two years to validate Qumran-MX switches and certify them as production-ready. This effort included multiple iterations of testing and bug fixing to port the Tomahawk driver to Qumran-MX, even though the devices come from the same vendor, implement the same protocols, and expose the same control plane abstractions. In practice, the problem of mapping abstract specifications of forwarding behavior down to real-world targets seems too hard to solve by hand.

Control Plane Synthesis. This paper presents a different approach to managing data plane diversity. Rather than relying on careful engineers to manually craft bug-free mappings from high-level abstractions to low-level targets, we show how to automate this task using program synthesis. More precisely, we develop Avenir, a system that automatically translates control plane operations written against an abstract forwarding specification (e.g., OF-DPA), into lower-level operations for a physical target (e.g., Qumran-MX).

*Work performed at Cornell.

Our approach proceeds in two steps. First, we use the P4 language [4] to model the behavior of the abstract and target devices. Although P4 was originally designed as a domain-specific language for programming devices like Barefoot's Tofino switch, it is also being used as a specification language for fixed-function devices (e.g., at Google [47]). For our purposes, what matters is that P4 provides a precise, bit-level specification of data plane behavior that can be mechanized using an SMT solver [21]. Hence, when P4 is not sufficiently expressive to model the pipelines' behavior, our approach should still be applicable. For example, one could work with other packet-processing languages like NPL, eBPF, or vendor SDKs. Second, we use *counterexample guided inductive synthesis* (CEGIS) [41] to translate the abstract control plane operations, such as inserting entries into a match-action table, into equivalent physical operations. Our synthesis algorithm is provably *sound* (i.e., if it succeeds, the abstract and target behaviors are guaranteed to be equivalent) and *complete* (i.e., if there a translation for a given operation, Avenir finds it).

At a technical level, we exploit the insight that data plane devices are fundamentally simple. When modeled as programs, they lack complex features like pointers and loops (parser state machines and uses of recirculation can be finitely unrolled in practice). Although data planes exhibit complexity in other dimensions, such as the number of protocols or table entries they support, the amount of processing they perform on any given packet is limited. Hence, it is possible to model their behavior using simple, loop-free programs that are amenable to analysis using automated solvers. In particular, P4's match-action tables can be treated as *program sketches*—i.e., programs populated with unknown variables called *holes*. The CEGIS loop synthesizes table operations by inductively filling in the program's holes. The controller interacts with these tables incrementally: table entries are usually not changed wholesale, but in small batches. We incrementally synthesize individual control plane operations rather than full tables, which greatly improves Avenir's efficiency.

However, even if one does synthesis incrementally, scaling up to real-world programs remains a significant challenge. Program synthesis has often been used in offline settings, where performance is not a critical concern. However, a typical control plane might modify a table every few milliseconds. To enable online operation, Avenir incorporates heuristic optimizations such as ignoring existing table rules (when possible), and learning "templates" that cache repeated patterns and avoid unnecessary calls to the SMT solver.

Implementation and Evaluation. We have built an implementation of Avenir in OCaml and Z3, and evaluated its effectiveness and scalability. In particular, we used Avenir to perform a "reboot" load test from the ONOS controller with moderate overhead: ONOS takes 15 minutes to generate 40k abstract IPv6 forwarding rules while our tool translates the insertions to a Broadcom pipeline in about 12 minutes. We conducted a series of experiments in which we retarget control

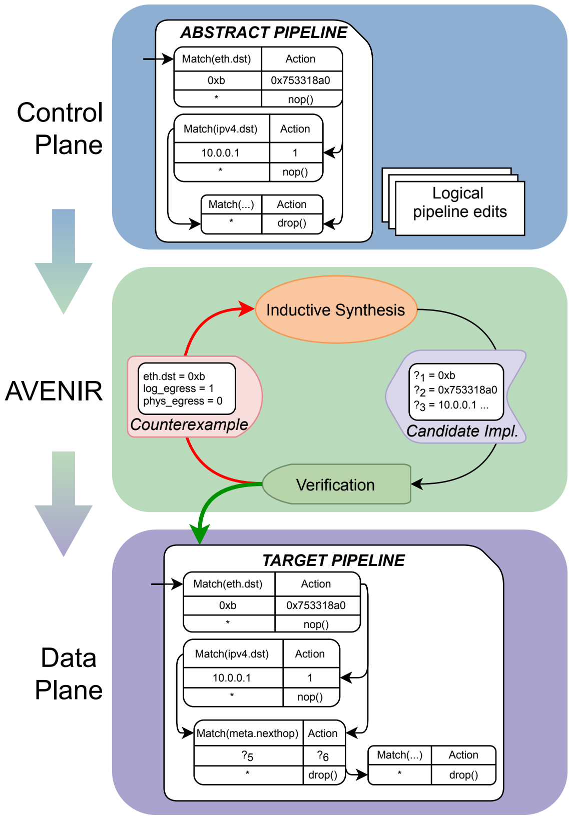


Figure 1: Avenir maps control plane operations for an abstract pipeline into corresponding operations for a target using sketch-based synthesis. The synthesis loop alternates between verifying the correctness of a candidate implementation and learning from counterexamples to generate a better one; the holes (e.g., $?_5$) in the target sketch denote missing values that are filled in using an SMT solver.

planes from one pipeline to another, and show that generated rules successfully forward packets on the Bmv2 software switch. Finally, to assess Avenir's scalability, we ran experiments on synthetic microbenchmarks.

Contributions. This paper presents Avenir, a practical control plane synthesis tool based on the following contributions:

- We present synthesis algorithm that incrementally computes changes to data plane operations, motivated by examples in real-world control planes.
- We formalize our synthesis algorithm and prove (in the appendix) that it is sound and complete.
- We present optimizations that leverage incrementality and domain insights to accelerate synthesis.
- We discuss an implementation and show through case studies and microbenchmarks that Avenir synthesizes control plane operations correctly with modest overheads.

L2_fwd	
eth.dst	**Action**
ABB28FC	set_out(5)

L3_fwd	
ipv4.dst	**Action**
10.0.0.1	set_out(8)

(a) Pipeline 1, with L2 and L3 forwarding for the original, homogeneous network.

L2_fwd	
eth.dst	**Action**
$?_1$	$?_2$

L3_fwd	
ipv4.dst	**Action**
10.0.0.1	set_meta(8)
$?_5$	$?_6$

LAG	
meta	**Action**
8	set_out(8)
$?_3$	$?_4$

(b) Pipeline 2, with a level of metadata indirection, and "holes" filled in. During synthesis, Avenir solves for these unknowns and concludes that $?_1 = \mathsf{ABB28FC}$, $?_2 = \mathsf{set\_meta}(5)$, $?_3 = 5$, $?_4 = \mathsf{set\_out}(5)$.

L2_fwd	
eth.dst	**Action**
ABB28FC	set_m2(5)

L3_fwd	
ipv4.dst	**Action**
10.0.0.1	set_m3(8)

LAG	
m2, m3	**Action**
5, 8	set_out(8)
5, *	set_out(5)
*, 8	set_out(8)

(c) Pipeline 3, which introduces two additional metadata fields.

fwd_table	
eth.dst, ipv4.dst	**Action**
*, 10.0.0.1	set_port(8)
ABB28FC, *	set_port(5)

(d) The second abstract pipeline which implements a "one big table."

$\boxed{(Pipe_1 \Rightarrow Pipe_2)}$

$e \mapsto \; e.\mathsf{table}\;\; e.\mathsf{keys}\;\; \mathsf{set\_meta}(e.\mathsf{out})$
$\qquad \mathsf{LAG}\;\; e.\mathsf{out}\;\; \mathsf{set\_out}(e.\mathsf{out})$

$\boxed{(Pipe_1 \Rightarrow Pipe_3)}$

$e \text{ in } \mathsf{L2} \mapsto \; \mathsf{L2}\;\; e.\mathsf{keys}\;\; \mathsf{set\_meta}(e.\mathsf{out})$
$\qquad \mathsf{LAG}\;\; (e.\mathsf{out}, *)\;\; \mathsf{set\_out}(e.\mathsf{out})$
$e \text{ in } \mathsf{L3} \mapsto \; \mathsf{L3}\;\; e.\mathsf{keys}\;\; \mathsf{set\_meta}(e.\mathsf{out})$
$\qquad \mathsf{LAG}\;\; (*, e.\mathsf{out})\;\; \mathsf{set\_out}(e.\mathsf{out})$
$\qquad \mathsf{L3}\;\; (r.\mathsf{m1}, e.\mathsf{out})\;\; \mathsf{set\_out}(e.out)$
$\qquad\quad$ for every existing row r in LAG

(e) Translations from Pipeline 1 to Pipelines 2 and 3.

$\boxed{(OBT \Rightarrow Pipe_1)}$

$e \text{ if eth.dst} = * \mapsto \mathsf{L2}\;\; e.\mathsf{ipv4.dst}\;\; \mathsf{set\_out}(e.\mathsf{out})$
$e \text{ if eth.ipv4} = * \mapsto \mathsf{L3}\;\; e.\mathsf{eth.dst}\;\; \mathsf{set\_out}(e.\mathsf{out})$
$\quad e \text{ otherwise} \mapsto \mathsf{Failure}$

$\boxed{(OBT \Rightarrow Pipe_2)}$

$e \text{ if eth.dst} = * \mapsto \mathsf{L2}\;\; e.\mathsf{ipv4.dst}\;\; \mathsf{set\_out}(e.\mathsf{out})$
$e \text{ if eth.ipv4} = * \mapsto \; \mathsf{L3}\;\; e.\mathsf{eth.dst}\;\; \mathsf{set\_meta}(e.\mathsf{out}),$
$\qquad \mathsf{LAG}\;\; (e.\mathsf{out}, *)\;\; \mathsf{set\_out}(e.\mathsf{out})$
$e \text{ otherwise} \mapsto \mathsf{Failure}$

$\boxed{(OBT \Rightarrow Pipe_3)}$

$e \text{ if eth.dst} = * \mapsto \mathsf{L2}\;\; e.\mathsf{ipv4.dst}\;\; \mathsf{set\_out}(e.\mathsf{out})$
$e \text{ if eth.ipv4} = * \mapsto \; \mathsf{L3}\;\; e.\mathsf{eth.dst}\;\; \mathsf{set\_meta}(e.\mathsf{out}),$
$\qquad \mathsf{LAG}\;\; (*, e.\mathsf{out})\;\; \mathsf{set\_out}(e.\mathsf{out})$
$\qquad \mathsf{LAG}\;\; (r.\mathsf{m1}, e.\mathsf{out})\;\; \mathsf{set\_out}(e.\mathsf{out})$
$\qquad\quad$ for every existing row r in LAG
$e \text{ otherwise} \mapsto \mathsf{Failure}$

(f) Translations from "one big table" to Pipelines 1 to 3.

Figure 2: Pipelines used in example scenario.

2 Background and Motivation

As shown in Figure 1, Avenir sits between the controller and the data plane, exposing an interface based on an abstract pipeline to the SDN control plane. It intercepts the control operations, translates them to the target pipeline, and passes results to the switch agent to install on the target device. Note that because Avenir works with an abstract notion of a pipeline, it could be used at multiple levels of abstraction—e.g., to implement a driver for a given switch, an abstraction layer like SAI, or even at higher layers of the SDN controller. Likewise, because Avenir operates on switch-by-switch granularity, it can expose different abstract pipelines for different targets. Avenir's synthesis algorithm is sound and its solutions are formally verified, which eliminates the potential for subtle bugs caused by the inherent complexity of the problem, assuming the specifications are correct. Avenir's algorithm is also complete—i.e., given sufficient time, it is guaranteed to find a correct sequence of target operations if it exists.

Status Quo: Manual Control Plane Mappings. Consider a simple running example based on ONOS that illustrates the need for a control plane synthesis tool. Suppose that each switch implements the simple L2-L3 pipeline in Figure 2a. In this pipeline, the output port is set based on the Ethernet and IPv4 destination addresses in the corresponding tables.

As the network matures, its engineers decide to add additional physical data planes—e.g., to incorporate a new generation of hardware or to avoid vendor lock-in. For instance, the pipeline, shown in Figure 2b, adds a layer of metadata indirection to the physical device to support link aggregation.

To avoid disrupting the control plane, which likely consists of hundreds of thousands of lines of code,[1] the engineers write a *driver* that translates operations written for Pipeline 1 into operations for Pipeline 2. In this case, the driver, shown in Figure 2e, is relatively simple: for each rule, it simply copies the output port into meta and inserts a row into the LAG table effectively copying the value of meta into the output port.

Now, suppose the engineers decide to support a third pipeline (Figure 2c), which sets a separate metadata field in each table. The translation (Figure 2e), is also simple, but requires some care to write—in particular, the L3 table's forwarding decision must always be preferred in the LAG table.

Finally, suppose the engineers want to migrate their original pipeline to a *one big table* abstraction (Figure 2d), similar to OpenFlow. Now, the engineers need to make code changes to all three translations (Figure 2f).

[1]ONOS has currently about 611k lines of Java code [28, 37].

Of course, the ONOS engineers could compose the translations from the one big table to the first pipeline, and on to the other pipelines. However as more and more logical and physical tables are added, managing a complex cascade of translations would become unwieldy, and hard to maintain.

Control Plane Synthesis with Avenir. Avenir improves upon the state of the art—i.e., writing manual translations—by automating the translation of rules from an abstract pipeline to a target pipeline. Of course, the programmer still needs to write programs that capture the behavior of both pipelines, and that's a non-trivial task. But we believe this should be less challenging than actually writing the translations—akin to describing source and target languages vs. writing a compiler.

To see how this is done, let's explore how Avenir translates abstract Pipeline 1 L2 insertions into Pipeline 2 insertions. First, assume, as shown in Figure 2b, that the L3 table is populated with rules that match on the IPv4 address (10.0.0.1) and set the metadata to (8), and the LAG table matches on that metadata and forwards out port 8. Consider inserting a single rule into the abstract Pipeline 1 L2 table that matches on $\mathsf{eth.dst} = \mathsf{ABB28FC}$ and sets the outport to 5. To reflect this update in Pipeline 2, we then need to solve for the unknowns, written as (?) in Figure 2b. These unknowns model the answers to questions like "Which tables need modification?" and "What should the matches/actions/action data be?"

More formally, the unknowns (?) represent a special kind of variable we instrument our program with, called a *hole*. Programs instrumented with holes are called *sketches*. We heuristically search for a valuation of these holes that makes the behaviors of the two pipelines equivalent. In this example, we could set $?_1 = \mathsf{ABB28FC}$, $?_2 = \mathsf{set\_meta}(5)$, $?_3 = 5$, and $?_4 = \mathsf{set\_out}(5)$. Since we do not need to insert a rule in the L3 table, we do not need to find values for these holes. In practice, holes can only be assigned values, not code snippets, like we are doing here for $?_2$ and $?_4$. We will see how to construct these sketches in detail in Section 3.2, and we will introduce our synthesis algorithm in Sections 3.3, 4.1 and 5.2.

As a strawman, we might consider an offline approach, where we synthesize the driver code once-and-for-all that translates any abstract operation into equivalent target operations. However, there are many cases (e.g., Figure 2f) where there is no translation that works for all abstract operations, this synthesis algorithm would fail to produce any solution in many cases where Avenir would succeed. Avenir's online solution allows for a more dynamic and flexible approach.

Incrementality and Optimizations. The key challenge in making Avenir practical is scaling up to handle real-world programs, which typically have at least dozens of tables with thousands of rules. Avenir needs to potentially compute a translation on every abstract control plane operation, so it must be responsive. As another strawman, imagine an approach that computes a full set of table rules on every control plane operation. This strategy might be workable when the tables have only a few rules, e.g., recomputing the existing match in Pipeline 2's L3 table, but it would quickly become a bottleneck if there were say, tens of thousands of rules in L3. Hence, we employ an incremental approach in which we synthesize "deltas" consisting of small batches of control plane operations rather than full tables. By only considering the most recent insertion or deletion into a table, we can often reuse previous solutions and avoid redundant recomputation.

Going a step further, we can cache "templates" derived from previous solutions to help translate future operations. For example, on the next insertion into L2, we can try to reuse the same stucture by inserting into L2_fwd and LAG, with actions set_meta and set_out, forcing the argument to set_meta to equal the LAG table match.

3 Control Plane Synthesis

Our synthesis algorithm is based on CEGIS [40]. The core of CEGIS is a loop with two main components: verification and inductive synthesis. In each iteration of the loop, a candidate implementation is run through the verification component to check correctness. If verification fails, a counterexample trace is produced, allowing the inductive synthesis component to learn from this failure to generate a better candidate. The loop terminates when verification ultimately succeeds.

In our setting, the CEGIS loop is run for each insertion into the abstract pipeline. Inductive synthesis produces candidate control plane implementations for the target pipeline, and verification checks whether the behavior of the two pipelines are equivalent. The rest of this section discusses the CEGIS components in detail. Section 4 discusses optimizations that make this approach efficient and scalable.

3.1 Basic Definitions and Verification

The *verification* component of the CEGIS loop determines whether the synthesized control plane operations implement the same packet-processing behavior on the target pipeline as on the abstract pipeline. We model packets as finite maps from a fixed set of header and metadata fields to bit vectors, and say two packets are equal and write $pkt = pkt'$ if they agree on all header fields. Packets have a direct interpretation as a boolean formula: for headers Hdr and a list $\vec{x} \subseteq \mathsf{Hdr}$, we write $pkt[\vec{x}]$ to mean $\bigwedge_{x \in \vec{x}} x = pkt.x$.

Syntax and Semantics. In Figure 3, we define the syntax of pipelines. A pipeline program is a just a command $c \in \mathsf{Cmd}$, that denotes a packet processing function, which we write $[\![c]\!] : \mathsf{Pkt} \to \mathsf{Pkt}$. Pipeline programs can contain bitvector expressions $e \in \mathsf{Expr}$ and boolean expressions $b \in \mathsf{Bool}$. Given a bitvector $[n]_s$ of length s, we use "wraparound" semantics when values n larger than $2^s - 1$ overflow. We often omit subscripts when s is clear from context or use evocative notation.

There are a few ways to compositionally build a pipeline program. First, fields f can be assigned values via the command $f := e$, which updates the packet pkt to $pkt\{f \mapsto n\}$, where e evaluates to n in pkt. Further, commands can be sequenced, $c_1; c_2$, which first executes c_1 then c_2. We can also

$$\begin{array}{rcll}
c & ::= & & (c \in \mathsf{Cmd}) \\
& | & f := e & \textit{Assignment(*)} \\
& | & c;c & \textit{Sequence(*)}\\
& | & \mathsf{if}\ \overrightarrow{b \to c}\ \mathsf{fi} & \textit{Guarded Commands(*)}\\
& | & \mathsf{apply}\ t & \textit{Table Application}\\
a & ::= & & (a \in \mathsf{Act})\\
& | & \lambda\,(\vec{x}).\ c(*) & \textit{Function}\\
t & ::= & \left\{\begin{array}{l} name : \mathsf{Name};\\ keys : \mathsf{Name}^+;\\ actions : \mathsf{Act}^+;\\ default : \mathsf{Act}\end{array}\right\} & \textit{Table Definition}\\
\delta,\varepsilon & ::= & & (\delta \in \mathsf{Edit})\\
& | & \mathsf{A}_x(\rho) & \quad\textit{Insertion}\\
& | & \mathsf{D}_x(n) & \quad\textit{Deletion}\\
\rho & \in & \mathsf{Row} = \mathsf{List}[\mathsf{BitVec}] \times \mathsf{List}[\mathsf{BitVec}] \times \mathbb{N} & \\
\tau,\sigma & \in & \mathsf{Inst} = \mathsf{Name} \to \mathsf{List}[\mathsf{Row}] & \\
v & ::= & [n]_n & \textit{Bitvector}\ (v \in \mathsf{BitVec})\\
h & ::= & \left\{\begin{array}{l} name : \mathsf{Name};\\ width : \mathbb{N}\end{array}\right\} & \textit{Header Field}\ (h \in \mathsf{Hdr})\\
m & ::= & \left\{\begin{array}{l} name : \mathsf{Name};\\ width : \mathbb{N}\end{array}\right\} & \textit{Metadata Field}\ (h \in \mathsf{Meta})\\
f & \in & \mathsf{Hdr} \cup \mathsf{Meta} & \\
x & \in & \mathsf{Name} & \\
n & \in & \mathbb{N} &
\end{array}$$

Figure 3: Pipeline syntax. Actions vary under starred variants

use conditional control flow, written if $b_1 \to c_1\ \ldots\ b_n \to c_n$ fi, which executes command c_i on the incoming packet *pkt* for the smallest-indexed b_i that evaluates to tt on *pkt*. These conditionals are similar to Dijkstra-style guarded commands [9]. Finally, table application $\mathsf{apply}(t)$ executes match-action table t. Tables are represented as records, where $t.name$ is table's name; $t.keys$ is a list of packet headers referred to by name; $t.actions$ is the list of actions (which are lexically-scoped anonymous functions $\lambda(\vec{x}).c$); and $t.default$ is the command that is executed when the table is missed. Only certain commands c may occur inside an action (denoted with a $(*)$ in Figure 3)—e.g., table application is not allowed.

Notice that tables have no way of referring to their entries. To represent entries in a table t, we maintain a *table instantiation* $\tau : \mathsf{Name} \to \mathsf{List}[\mathsf{Row}]$, alongside the syntactic pipeline, which maps table names to their row lists. We write Inst for the set of all instantiations. We refer to the pair (c,τ) as the *pipeline state*. A row $\rho \in \mathsf{Row}$ is a triple $\rho = (\vec{m}, \vec{d}, a)$, where $\vec{m}$ are matches, a is the action index and $\vec{d}$ is the action data.

We can define a source-to-source syntactic transformation $\mathsf{subst}(c,\tau)$ that replaces every occurence of $\mathsf{apply}(t)$ in c with a guarded command encoding the rows of the table $\vec{\rho} = \tau(t.name)$, as follows, where the ith row $\rho_i = (\vec{m_i}, \vec{d_i}, a_i)$:

$$\mathsf{if}\ \left(\begin{array}{ccc} t.keys = \vec{m_0} & \to & t.action[a_0](\vec{d_0}) \\ & \cdots & \\ t.keys = \vec{m_n} & \to & t.action[a_n](\vec{d_n}) \\ \mathsf{tt} & \to & t.default \end{array}\right)\ \mathsf{fi}$$

Hole	Description
$?\mathsf{Del}_{t,i} = 1$	Delete row i in table t
$?\mathsf{Add}_{t,j} = 1$	Add j rows to table t
$?\mathsf{Act}_{t,j} = i$	New Row j in table t (if added), has action i
$?\mathsf{k}_{t,j} = v$	New Row j in table t (if added), matches header k with v
$?\mathsf{d}_{t,j,i} = v$	New Row j in table t (if added with action i) has action data for parameter d set to v

Figure 4: Summary of holes used in sketching.

We say that a row $(\vec{m}, \vec{d}, a)$ is well-defined for a table t when $|\vec{m}| = |t.keys|$, $a < |t.actions|$, and for the parameters $\vec{x}$ of $t.actions[a]$, $|\vec{d}| = |\vec{x}|$. Further, an instance is well-defined when all of its rows are well-defined for their tables, and a command is well-defined when no two tables have the same name. We assume that commands and instantiations are well-defined, and that there are no bit-width mismatches: both are easy to check statically.

Finally, we have control plane edits ($\delta \in \mathsf{Edit}$), which are operations that allow the control plane to modify table instantiations. We interpret them as functions, i.e., $\delta(\tau) \in \mathsf{Inst}$. There are two kinds of edits: insertions and deletions. For a given instance τ, an insertion $\mathsf{A}_x(\rho)(\tau)$ appends ρ to the end of $\tau(x)$ (meaning it has the lowest priority). If $\tau(x)$ has a row ρ' with the same matches as ρ, the inserted row is dropped. A deletion $\mathsf{D}_x(i)(\tau)$ removes the ith element from $\tau(x)$.

Now that we know how to interpret pipelines as functions, we say $c_1 = c_2$ when they are functionally equivalent. To check this condition, we use predicate transformer semantics to generate a verification condition [13], written $c_1 \equiv c_2$, which we check using an SMT solver, by running $\mathsf{CheckSat}(c_1 \not\equiv c_2)$. If the solver returns UNSAT, we conclude the programs are equivalent. Otherwise, it returns SAT as well as a model that encodes a counterexample χ—i.e., an input and output packet pair χ that demonstrates different behavior in the abstract and physical programs, writing χ_0 and χ_1 for the input and output packets respectively. It is easy to prove that this validity check implies functional equivalence.

Theorem 1. *For every pair of pipelines* c_1, c_2*, if* $c_1 = c_2$ *then* $\mathsf{CheckSat}(c_1 \not\equiv c_2) = \mathsf{UNSAT}$*, and if* $c_1 \neq c_2$ *then* $\mathsf{CheckSat}(c_1 \not\equiv c_2) = \mathsf{Sat}\ \chi$.

Proof. By soundness of verification conditions with respect to the denotational semantics of guarded commands [9, 13]. □

3.2 Synthesizing Candidates via Sketches

To propose new candidate programs for verification, we use a technique called Sketching [41]. A *sketch* is a command containing special variables called *holes*. Aside from holes for values (i.e., ?k for match keys and ?d for action data), which we introduced in Section 2, we also need holes for table entries, corresponding to deletions (?Del), insertions (?Add) and action choice (?Act). The meaning of these holes is described in Figure 4.

$$\mathsf{fix\_cex}(p,\sigma,\chi,n,\vec{x}) \triangleq \chi_0[\vec{x}] \Rightarrow wp\,(\mathsf{instr}(p,\sigma,\cdot,n),\chi_1[\vec{x}])$$
$$\mathsf{model}(p,\sigma,X) \triangleq \mathsf{CheckSat}\left(\forall \vec{x}.\, \textstyle\bigwedge_{\chi\in X} \mathsf{fix\_cex}(p,\sigma,\chi,|X|,\vec{x})\right)$$

Figure 5: The model function. In the above, the vector $\vec{x}$ is all of the non-hole variables that occur in the formula.

To compute a candidate solution in our CEGIS loop, we first instrument the program with holes. We write $\mathsf{instr}(c,\tau,\vec{\delta},n)$ to describe the program $\vec{\delta}(\tau(c))$ with deletion holes for every row in τ, and holes for n row insertions. We do not add deletion holes for insertions in $\vec{\delta}$, which is crucial for the completeness theorem (Section 4). We lift this function from tables to programs in the obvious way.

Consider the L2 table from pipelines 1 and 2. To instrument it with holes, allowing for a single insertion, we would insert a deletion hole for the existing rule and a single row of insertion holes, yielding the following sketch:

	Match(eth.dst)	**Action**
?Del = 0	ABB28FC	set_out(5)

?Add = 1	?eth.dst	if ?Act = 0 → set_out(?p)
		?Act = 1 → drop()
		fi

A possible model for these holes that matches the destination MAC address with 00 : 00 : 00 : 00 : 00 and drops the packet, is $\{?\mathsf{Del} \mapsto 0, ?\mathsf{eth.dst} \mapsto 0, ?\mathsf{Act} \mapsto 1\}$. Note that ?p is irrelevant, so we omit it from the model.

Of course, sketches represent a vast search space of edits: every existing table row can be deleted, and up to n rows can be inserted. Blindly searching through this space would not scale in practice. Instead, we learn from counterexamples to help guide the search toward a solution.

3.3 Counterexample-Guided Search

When the solver determines that a proposed candidate pipeline is not equivalent to the abstract pipeline, it generates a counterexample χ that encodes an input-output packet pair. This pair corresponds to a behavior of the abstract switch that is not replicated in the candidate or vice versa. We can use this counterexample to guide our search. More formally, we use the weakest precondition $wp(c,\varphi)$ whose satisfying models are inputs that, after executing c, yield outputs satisfying φ.

The fix_cex function constructs the formula $\chi_0[\vec{x}] \Rightarrow wp(s,\chi_1[\vec{x}])$ for the sketch $s = \mathsf{instr}(p,\sigma,\cdot,|X|)$. The formula identifies edits that when applied to the physical pipeline state (p,σ) produce the input-output behavior indicated by χ.

The function model in Figure 5 lifts fix_cex over all counterexamples X that have been seen so far. Notice that we only instrument the physical pipeline with $|X|$ insertion holes since each counterexample hits at most one rule in each table.

3.4 Synthesis Algorithm

The full synthesis algorithm is presented in Figure 6. Given a abstract pipeline l, a target pipeline p, an abstract table instan-

```
$\mathsf{cegis}(l,p,\tau,\sigma,\vec{\delta},X) \triangleq$
  match $\mathsf{CheckSat}(\mathsf{subst}(l,\tau) \not\equiv \mathsf{subst}(p,\vec{\delta}(\sigma)))$ with
   | UNSAT $\to$ Ok $\vec{\delta}$
   | SAT $\to$
      match $\mathsf{model}(p,\vec{\delta}(\sigma),\{\chi\}\cup X)$ with
        | UNSAT $\to$ Fail
        | SAT $\vec{\delta}'$ $\to$ $\mathsf{cegis}(l,p,\tau,\sigma,\vec{\delta}',\{\chi\}\cup X)$
```

Figure 6: Simple Algorithm for Control Plane Synthesis.

tiation τ, a target table instantiation σ, a sequence of physical edits $\vec{\delta}$, and a set of counterexamples X, $\mathsf{cegis}(l,p,\tau,\sigma,\vec{\delta},X)$ produces a sequence of edits $\vec{\varepsilon}$ such that $\mathsf{subst}(l,\tau) = \mathsf{subst}(p,\vec{\varepsilon}(\sigma))$ if one exists. We initially call the algorithm with $\vec{\delta} = []$ and $X = \{\}$. First, we call the SMT solver to check for equality. If the programs are equal, we are done, and return $\vec{\delta}$. Otherwise, we get a counterexample χ and solve for new edits by augmenting X with χ, applying the edits to the target pipeline and calling model. If it returns UNSAT, there is no way to make the programs equivalent and we fail. Otherwise, we get a new sequence of edits and keep searching.

3.5 Formal Properties

Next we establish two formal properties for our synthesis algorithm: soundness and completeness. Soundness means that synthesized target operations produce equivalent behavior.

Theorem 2 (Soundness). *For every* $l,p \in \mathsf{Cmd}$, $\tau,\sigma \in \mathsf{Inst}$, $\vec{\delta} \in \mathsf{List}[\mathsf{Edit}]$, *and* $X \subseteq [\![\mathsf{subst}(l,\tau)]\!] \cap [\![\mathsf{subst}(p,\vec{\delta}(\sigma))]\!]$ *if* $\mathsf{cegis}(l,p,\tau,\sigma,\vec{\delta},X) = \mathsf{Ok}\,\vec{\varepsilon}$ *then* $\mathsf{subst}(l,\tau) = \mathsf{subst}(p,\vec{\varepsilon}(\sigma))$.

Proof. Follows from Theorem 1. ☐

Completeness says that if a solution exists, then our synthesis algorithm will (eventually) find it.

Theorem 3 (Completeness). *For every* $l,p \in \mathsf{Cmd}$, $\tau,\sigma \in \mathsf{Inst}$, $\vec{\delta} \in \mathsf{List}[\mathsf{Edit}]$, *and* $X \subseteq [\![\mathsf{subst}(l,\tau)]\!] \cap [\![\mathsf{subst}(p,\vec{\delta}(\sigma))]\!]$, *if* $\exists \vec{\delta}' \in \mathsf{List}[\mathsf{Edit}].\ \mathsf{subst}(l,\tau) = \mathsf{subst}(p,\vec{\delta}'(\sigma))$ *then* $\exists \vec{\delta}'' \in \mathsf{List}[\mathsf{Edit}].\ \mathsf{cegis}(l,p,\tau,\sigma,\vec{\delta},X) = \mathsf{Ok}\ \vec{\delta}''$ *and* $\mathsf{subst}(l,\tau) = \mathsf{subst}(p,\vec{\delta}''(\sigma))$.

Proof. By induction on the size of $\mathsf{Pkt} \setminus \pi_1(X)$. ☐

Limitations The main limitation of this first synthesis algorithm is that the number of queries is bounded by the number of counterexamples—i.e., every possible packet. Given an MTU of n, there could be as many as 2^n packets.

4 A Scalable Solution: Incremental Synthesis

To obtain a scalable synthesis algorithm, we first exploit the insight that the control plane operates in an incremental fashion—i.e., before each control plane operation, the data

	Match(eth.dst)	**Action**
$?\mathsf{Del}_0 = 0$	ABB28FC	set_out(5)

$?\mathsf{Add}_0 = 1$	$?\mathsf{eth.dst}_0$	if $?\mathsf{Act}_0 = 0 \rightarrow$ set_out($?\mathsf{p}_0$)
		$?\mathsf{Act}_0 = 1 \rightarrow$ drop()
		fi

$?\mathsf{Add}_1 = 1$	$?\mathsf{eth.dst}_1$	if $?\mathsf{Act}_1 = 0 \rightarrow$ set_out($?\mathsf{p}_1$)
		$?\mathsf{Act}_1 = 1 \rightarrow$ drop()
		fi

	Match(ip.dst)	**Action**
$?\mathsf{Del}_2 = 0$	10.0.0.1	set_out(8)
$?\mathsf{Del}_1 = 0$	8.8.8.8	set_out(47)

$?\mathsf{Add}_3 = 1$	$?\mathsf{ipv4.dst}_3$	if $?\mathsf{Act}_2 = 0 \rightarrow$ set_out($?\mathsf{p}_2$)
		$?\mathsf{Act}_2 = 1 \rightarrow$ drop()
		fi

$?\mathsf{Add}_3 = 1$	$?\mathsf{ipv4.dst}_3$	if $?\mathsf{Act}_3 = 0 \rightarrow$ set_out($?\mathsf{p}_4$)
		$?\mathsf{Act}_3 = 1 \rightarrow$ drop()
		fi

(a) Basic Sketch: Satisfiable for packets that hit L2's first row and L3's second.

	Match(eth.dst)	**Action**
$?\mathsf{Del}_0 = 0$	ABB28FC	set_out(5)

$?\mathsf{Add}_0 = 1$	$?\mathsf{eth.dst}_0$	if $?\mathsf{Act}_0 = 0 \rightarrow$ set_out($?\mathsf{p}_0$)
		$?\mathsf{Act}_0 = 1 \rightarrow$ drop()
		fi

	Match(ip.dst)	**Action**
$?\mathsf{Del}_2 = 0$	10.0.0.1	set_out(8)
	8.8.8.8	set_out(47)

$?\mathsf{Add}_1 = 1$	$?\mathsf{ipv4.dst}_1$	if $?\mathsf{Act}_1 = 0 \rightarrow$ set_out($?\mathsf{p}_1$)
		$?\mathsf{Act}_1 = 1 \rightarrow$ drop()
		fi

(b) Incremental Sketch: Unsatisfiable for packets that hit L2's first row and L3's second, which triggers backtracking, remembering that the previously-synthesized edit was incorrect.

Figure 7: Examples of basic and incremental sketches for Pipeline 1.

planes are already equivalent, so we only need to handle incremental changes to the abstract program, such as adding or deleting a rule. In the common case, we do not have to resynthesize all of the previously generated rules. However, some care is needed as certain control plane operations do require deleting previously installed rules.

4.1 Single Counterexample-Guided Search

Our first enhancement to the basic synthesis algorithm is to only add insertion holes to solve for the most recent counterexample, and only add deletion holes for state that existed before synthesis began, which greatly reduces the number of holes we need to produce as we explore the space. Instead of instrumenting the program with insertion holes for every counterexample, we only do it for the most recent one.

Consider again the L2 and L3 tables from pipelines 1 with the initial state depicted in Figure 2a. We want to synthesize edits that send Ethernet packets that miss in the L2_fwd with destination DECAFBAD out on port 47. Suppose the first counterexample has input packet $\chi_0 = \{\mathsf{eth.dst} \mapsto \mathsf{DECAFBAD}, \mathsf{ipv4.dst} \mapsto 8.8.8.8\}$, and output packet $\chi_1 = \chi_0\{\mathsf{out} \mapsto 47\}$. Let's say on the first iteration we produce the (incorrect) edit to L2_fwd that maps ipv4.dst = 8.8.8.8 to set_out(47), and the verification step will provide a new counterexample.

Suppose the next counterexample has input packet $\chi_0' = \{\mathsf{eth.dst} \mapsto \mathsf{ABB28FC}, \mathsf{ipv4.dst} \mapsto 8.8.8.8\}$, and output packet $\chi_1' = \chi_0'\{\mathsf{out} \mapsto 5\}$. Now the simple algorithm will produce the sketch in Figure 7a, which can be solved by deleting the already inserted row ($?\mathsf{Del}_1 = 1$) and adding the single required row to the L2 table ($?\mathsf{Add}_0 = 1$, $?\mathsf{eth.dst}_0 =$ DECAFBAD, $?\mathsf{Act}_0 = 0$, $?\mathsf{p}_0 = 47$, and remaining Add/Del holes disabled).

In contrast, the incremental search will first create the unsatisfiable sketch shown in Figure 7b. There is no way to fill its holes to satisfy the above counterexample. We backtrack with the knowledge that $?\mathsf{ipv4.dst} \neq 8.8.8.8$ and attempt to solve the original sketch with respect to the original counterexample, and the only remaining solution is correct.

First, notice that the final simple sketch uses 21 holes, whereas each incremental sketch uses only 10. On the other hand, the incremental search sends 3 sketches to the solver as opposed to the simple search, which only sends 2. Why do we want to send *more* queries to Z3 instead of less? This is a result of the NP-completeness of SAT/SMT solving. Solving more formulae with fewer variables is often faster than solving fewer formulae with more variables. Here, the search space size for the 3 incremental sketches is approximately $3 \cdot |B|^{10}$, whereas for "simple" query it is approximately $|B|^{21}$, where $|B|$ is the size of the bitvector domain.

Further, observe that the incremental sketches we send will *always* have 10 holes, independent of the number of counterexamples, whereas the simple sketch will continue to add holes as the number of counterexamples grows.

We formalize this new incremental model-finding function model' in Figure 8. It is defined in term of a satisfiability check for a conjunction of three sub-formulas. The first conjunct uses a modified fix_cex function that instruments the program with one addition hole per table. The second conjunct, φ, limits the search by preventing models from reoccurring. The final conjunct is a search oracle HEURISTIC() that computes restrictions on the search space. The only constraints on HEURISTIC() are that it must not add covered rules or previously-deleted rules (to avoid looping), and it must not permanently preclude any solution (to ensure completeness).

$$\mathsf{fix\_cex}(p,\sigma,\vec{\delta},\vec{x},\chi) \triangleq pkt[\vec{x}] \Rightarrow wp(\mathsf{instr}(p,\sigma,\vec{\delta},1),pkt'[\vec{x}])$$

$$\mathsf{model}'(p,\sigma,\vec{\delta},\chi,\varphi) \triangleq \mathsf{SAT}\left(\begin{array}{l}\forall\vec{x}.\mathsf{fix\_cex}(p,\sigma,\vec{\delta},\vec{x},\chi))\\ \wedge\,\varphi\wedge\mathsf{HEURISTIC}()\end{array}\right)$$

Figure 8: The model' function computes edits to physical state (p,σ) to accomodate the counterexample χ. The oracle soundly restricts the search space.

```
$\mathsf{cegis}(l,p,\tau,\delta,\sigma) \triangleq \mathsf{verify}(l,p,\delta(\tau),\sigma,[\,])$
$\mathsf{verify}(l,p,\tau,\sigma,\vec{\delta}) \triangleq$
  match $\mathsf{CheckSat}(\mathsf{subst}(l,\tau) \not\equiv \mathsf{subst}(p,\vec{\delta}(\sigma)))$ with
    | $\mathsf{UNSAT} \to \mathsf{Ok}\ \vec{\delta}$
    | $\mathsf{SAT}\ \chi \to \mathsf{solve}(l,p,\tau,\sigma,\vec{\delta},\chi,\mathsf{tt})$
$\mathsf{solve}(l,p,\tau,\sigma,\vec{\delta},\chi,\varphi) \triangleq$
  match $\mathsf{model}'(p,\sigma,\vec{\delta},\chi,\varphi)$ with
    | $\mathsf{UNSAT} \to \mathsf{Fail}$
    | $\mathsf{SAT}\ \vec{\delta}' \to$
      match $\mathsf{verify}(l,p,\tau,\sigma,\vec{\delta}\circ\vec{\delta}')$ with
        | $\mathsf{Ok}\ \vec{\delta}'' \to \mathsf{Ok}\ \vec{\delta}''$
        | $\mathsf{Fail} \to \mathsf{solve}(l,p,\tau,\sigma,\vec{\delta},\chi,\varphi\wedge\neg\vec{\delta}')$
```

Figure 9: The incremental backtracking CEGIS algorithm.

4.2 Incremental Synthesis Algorithm

We present our incremental synthesis algorithm in Figure 9. It comprises two mutually recursive functions: verify, which checks the verification condition and solve, which generates new models. Both functions take the same arguments: the abstract and target programs and instantiations ((l,τ) and (p,σ) respectively), and a sequence of edits to the target program $\vec{\delta}$. They either return $\mathsf{Ok}\ \vec{\delta}'$, where $\vec{\delta}$ is the prefix of $\vec{\delta}'$ and $\mathsf{CheckSat}(\mathsf{subst}(l,\tau) \not\equiv \mathsf{subst}(p,\vec{\delta}'(\sigma))) = \mathsf{UNSAT}$, or Fail, if there is no such $\vec{\delta}'$. The cegis function is the "main" method. It takes the abstract and target pipelines (l and p) and instantiations (τ and σ) as arguments, as well as the abstract edit δ. It then applies δ to τ and invokes verify with no target edits.

The verify function resembles the cegis function of Section 3. It first checks whether the programs are equal, and if so returns $\mathsf{Ok}\ \vec{\delta}$. Otherwise it calls solve with an initial counterexample χ and an unrestricted model, which searches for an edit to make the programs equivalent.

The solve function takes the standard arguments, with the addition of a counterexample χ and the model space restriction formula φ, which keeps track of failed solutions for χ, to prevent repetition. First, model' searches for a target edit that corrects the behavior for the counterexample. If none exists, we return Fail, indicating that there is no sequence of equivalent target edits with the prefix $\vec{\delta}$. Otherwise, model' provides a model $\vec{\delta}'$. In this case we extend the running sequence of edits to $\vec{\delta}\circ\vec{\delta}'$ and call back to verify. If successful, we return the result, otherwise we preclude $\vec{\delta}'$ from the space of possible models φ (writing $\neg\vec{\delta}'$ for the negation of valuations that produce $\vec{\delta}'$.) Then we recursively call solve and continue searching within this restricted space of models.

4.3 Formal Properties

We prove that the incremental algorithm is also sound and complete. As with the simpler algorithm, the proof of soundness follows by the correctness of the verification condition.

Theorem 4 (Incremental Soundness)**.** *For every* $l,p \in \mathsf{Cmd}$, $\tau,\sigma \in \mathsf{Inst}$, $\delta \in \mathsf{Edit}$, $X \subseteq [\![\mathsf{subst}(l,\tau)]\!] \cap [\![\mathsf{subst}(p,\vec{\delta}(\sigma))]\!]$, *if* $\mathsf{cegis}(l,p,\tau,\sigma,\delta,X) = \mathsf{Ok}\ \vec{\varepsilon}$, *then* $\mathsf{subst}(l,\tau) = \mathsf{subst}(p,\vec{\varepsilon}(\sigma))$.

Proof. Again, the result follows from Theorem 1. ☐

As in the simple synthesis algorithm, incremental completeness relies on the finite domain, which here is the product of two finite domains: (1) sequences of reachable edits that do not redundantly add and delete a rule, and (2) the number of valuations for the holes introduced by the instr function.

Theorem 5 (Incremental Completeness)**.** *For every abstract program* l*, target program* p*, abstract instantiation* τ*, target instantiation* σ *and abstract edit* δ *if* $\exists\vec{\varepsilon} \in \mathsf{List}[\mathsf{Edit}]$. $\mathsf{subst}(l,\delta(\tau)) = \mathsf{subst}(p,\vec{\varepsilon}(\sigma))$ *then* $\exists\vec{\delta}' \in \mathsf{List}[\mathsf{Edit}]$. $\mathsf{cegis}(l,p,\tau,\delta,\sigma,[\,]) = \mathsf{Ok}\ \vec{\delta}'$ *and* $\mathsf{subst}(l,\delta(\tau)) = \mathsf{subst}(p,\vec{\delta}'(\sigma))$.

Proof. By strong outer induction on the size of the reachable non-deleting edit sequences, and strong inner induction on the (lexicographically ordered) size of the counterexample set and the number of models in each model space. ☐

Theorem 5 proves that Avenir translates abstract operations given unbounded resources. In practice, Avenir's effectiveness relies on heuristics and optimizations.

5 Heuristics and Optimizations

Avenir offers a number of heuristic optimizations designed to help it scale to larger networks. Interestingly, these optimizations need not be sound. We introduce a run-time check for soundness and revert the optimization if it fails. We focus on two classes of optimizations: verification and model finding.

5.1 Exploiting Incrementality

The key to scalable synthesis is to adopt an incremental approach and focus on edits, while incorporating further optimizations within the verification and synthesis steps.

Fast Counterexamples. In the incremental setting, we know that a new abstract insertion δ must be the cause of any semantic difference with the target pipeline. We symbolically compute packets that hit δ via an SMT query that gives us a potential counterexample packet *pkt*. We use the denotational semantics to check whether *pkt* is a real counterexample. If *pkt* doesn't induce different behavior we retry the query (in practice 10 times) until we either obtain a true counterexample, or resort back to the standard equivalence check.

Program Slicing. We leverage the incrementality assumption to use program slicing to verify only the part of the program that changes. This isn't always sound, so we check that the abstract edits are reachable iff the target edits are. We also have a faster and stronger constraint that checks that the abstract and target matches are disjoint from the extant rules. If both conditions fail, we run the full equivalence check. In practice, slicing composes with constant propagation and dead code elimination to normalize the queries.

Query Templates. The queries produced using program slicing are often syntactically similar. So when we see two validity queries that only differ in their specific concrete values, we try to abstract those concrete values into a universally quantified variable. We then check whether that more-general query is valid. If it is, we add it to a cache of templates, otherwise we continue in a CEGIS loop by negating the valuation of the quantified variables and trying again. Whenever we get future queries that are instances of the template, we can return VALID without having to consult the SMT solver.

Translation Templates. As with queries, we can cache translations of operations by generalizing over their concrete values to obtain a template. The template observes the way that concrete values are mapped from previously-seen abstract insertions into their equivalent target insertions, and structurally replicates that mapping on the new abstract insertion. It also observes the cache of translations for differing constants and generates unused constants for new rules which optimizes for metadata patterns like in Figures 2b and 2c. Note that before adding a solution to the cache, Avenir optionally reduces its size, by heuristically removing superfluous target edits, which improves the generality of the solution. When no template applies, Avenir relies on a heuristic-guided search.

5.2 Model-Finding Heuristics

Now we describe the implementation of the HEURISTIC() oracle, which abstracts a combination of heuristics. In our formalization, we assume that the heuristics are always complete. However in practice, many of Avenir's individual heuristics are not; when a given combination fails, we disable some and try again with a different combination. This search through the heuristics is currently hard-coded, but we plan to support user control of the search strategy and custom heuristics. We describe the heuristics useful in our experiments here.

Ternary and Optional Matching. In the previous sections, we only inserted holes to generate exact matches. We can generate ternary matches for a match key k, which allows us to represent, say, a wild-carded IPv4 source address in only a single row (rather than 2^{32} exact-match rows). To do this, we generate a pair of holes ?k and $?\mathsf{k}_{\mathtt{mask}}$ and encode the match as $\mathsf{k}\&?\mathsf{k}_{\mathtt{mask}} = ?\mathsf{k}$. To eliminate duplicate keys we also enforce the constraint $?\mathsf{k}\&?\mathsf{k}_{\mathtt{mask}} = ?\mathsf{k}$. For optional matches, we restrict the masks to be all 1s or all 0s.

Exact and Mask Hints. When a row is inserted into the abstract pipeline, the non-wildcarded keys K of that row are likely relevant in classifying packets. So, we force the relevance of matches on fields in K, either by copying the abstract match values into the target edits (which is very optimistic), or by forcing their masks (if masking is enabled) to be all 1s.

Action Hints. Given a counterexample (pkt_0, pkt_1), we can observe the variables that change in the abstract program, i.e., $\Delta = \{x \mid pkt_0.x \neq pkt_1.x\}$, and ensure that all edits have actions that can influence the value of some variable in Δ.

Other Optimizations. Our final collection of optimizations are based on intuitive heuristics that arise often in practice.

- **Reachable Adds.** We force synthesized models to be reachable using the counterexample driving the search.
- **Prefer Adds.** We try to find a solution that does not require deleting existing rules.
- **Prefer Non Zero Models.** We enforce $?\mathsf{k} \neq 0 \neq ?\mathsf{d}$ for all key and data holes, unless they are wildcarded.
- **Bounded Edits.** We restrict the search space so that backtracking is triggered beyond specified limits.
- **Previous Counterexamples.** We try to synthesize rules that do not violate previously-seen counterexamples.

6 Implementation

We implemented Avenir in approximately 11K lines [37] of OCaml code that interfaces with Z3 [8]. Our implementation accepts a description of an abstract and a target pipeline, sequences of insertions to both programs (to construct the initial state), as well as a sequence of abstract edits to synthesize. Avenir then produces a sequence of edits to the target program (or fails if no such sequence exists). All of the optimizations described in 5 are configurable as command line flags. In our implemention, we use an efficient encoding of the weakest precondition [13], which has linear size for the programs in our internal syntax.

P4 Program Encoding. The front-end of our implementation supports a large subset of P4, via an encoding from P4's control blocks into Avenir's internal syntax. This translation resembles previous work on verifying P4 programs [21]. Of course, P4 is a larger language than Avenir's syntax. We support more complex P4 language constructs by desugaring them into sequences of internal commands.

We currently assume that all of the data plane programs use the same parser and headers. Hence, in cases where a mapping only exists due to invariants enforced by the parser—e.g., that a packet cannot simultaneously have IPv4 and IPv6 headers—these assumptions must be manually encoded as annotations. We also ignore match kinds and assume all matches are either exact, ternary or optional, depending on command line flags. Finally, we manually encode certain device-specific behaviors

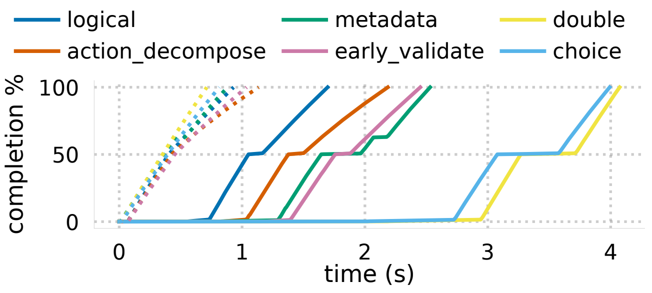


Figure 10: Retargeting case study: solid lines show cold-start completion %; dotted lines show hot-start completion %.

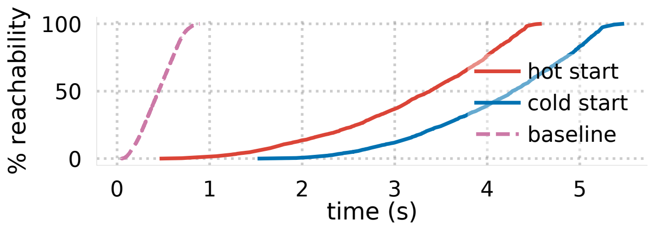


Figure 11: Proportion of all pairs of 64 hosts connected in a star topology that have completed a successful IPv4 ping.

such as the initial value fields and the drop port value. Our implementation is on GitHub[2] under an open-source license.

7 Evaluation

To evaluate Avenir, we demonstrate its functionality under a variety of synthetic and realistic scenarios, and measured its performance against hand-written baselines. First, we show how Avenir can automatically retarget a given abstract pipeline to multiple target pipelines (Section 7.1). Second, we pass packets through the Bmv2 software switch using the generated rules, which both shows they are correct and quantifies Avenir's performance when installing rules for multiple hosts (Section 7.2). Third, we present a case study consisting of a realistic workload drawn from the Trellis data center fabric, running on top of the ONOS SDN controller [2, 29] (Section 7.3). Finally, we study Avenir's scalability via a suite of microbenchmarks (Section 7.4). Our evaluation pays particular attention to the caches, as these are particularly important to obtain good performance.

Summary of Results. Overall, our evaluation shows that, in a variety of cases, Avenir can translate large numbers of rules efficiently. The retargeting, emulation, and ONOS experiments show that Avenir is effective at mapping to and from a variety of programs, and demonstrate that the caching optimizations are highly effective at reducing overheads.

7.1 Retargeting Study

Avenir allows operators to expose a single pipeline abstraction to the control plane, while implementing the forwarding logic over a myriad of physical devices. We demonstrate this use case via a retargeting study, where we retarget an initial program onto a variety of different target pipelines.

The logical program logical.p4 is a simple L2-L3 pipeline followed by a PUNT table that performs packet validation on all headers and metadata. We describe 5 additional target pipelines in terms of the changes to logical.p4:

(early_validate.p4) Replaces the PUNT table of logical.p4 with an ACL that can only match on addresses. Adds a validation table prior to the L2 table that matches on the validity of IPv4 and the TTL field and conditionally applies the rest of the pipeline.

(action_decomp.p4) Decomposes the L3 table into two tables, (1) a forward table that matches on the IPv4 destination and sets the output port, (2) a rewrite table that matches on the IPv4 destination and performs MAC rewriting.

(metadata.p4) Instead of setting the output port, the L2 and L3 tables set a metadata field. This metadata field is mapped to the output port in the nexthop table, which is applied between the L2 and L3 tables.

(double.p4) Applies all three tables in the pipeline twice.

(choice.p4) Introduces a staging table that sets a metadata variable to select between copies of the abstract pipeline.

We used Avenir to translate 1,001 logical.p4 insertions (1 into PUNT for TTL checking, 500 into L2 for Ethernet destination forwarding, and 500 into L3 for IPv4 destination forwarding and MAC rewriting). We show completion graphs for each target in Figure 10.

There are a few things to notice. Every line has an "elbow" at the 50% mark on the y-axis, after which the slope decreases. This represents the transition between parts of the workload. The L3 insertions are slower, because the L2 table is already populated with 500 rules, and slicing has to deal with larger tables. Further, these rules may cause the query template cache to miss: the second "elbow" on the metadata line indicates where the query cache's synthesis engine was able to successfully abstract.

To further demonstrate the power of our template caches, we compare our "cold-start" synthesis (solid lines), where the caches are empty, with "hot-start" synthesis (dotted lines), where the caches are fully populated. We achieve this by running Avenir on the same data twice, without resetting the caches, and logging performance for the second run. The massive performance increase is seen in Figure 10. Network operators concerned with nondeterministic runtimes associated with synthesis can manually populate their caches.

7.2 Network Emulation

We use Avenir to program the entries of a programmable software switch (bmv2) running in a network emulator (mininet). We configure 64 hosts in a star topology connected to a single

[2] Available at https://github.com/cornell-netlab/avenir

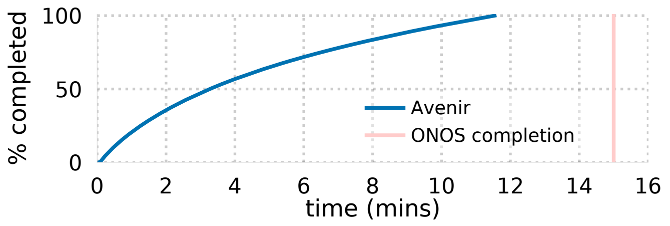


Figure 12: Completion graph for mapping 40k fabric.p4 IPv6 route insertions onto bcm.p4; ONOS takes around 15 min.

switch, and install rules to establish all-pairs ping connectivity. The P4 program running on the software switch is the simple_router.p4 program from the Bmv2 repository. The abstract program is a modified version that joins together the L3 rewriting and forwarding tables into one.

We generate rules required to establish all-pairs connectivity into the logical program and use Avenir to synthesize the equivalent edits into simple_router.p4. We then report the time of the first successful ping between each pair of hosts. We compare Avenir cold-cache run with a manually generated sequence of rule insertions and a pre-populated hot-cache, the results are depicted in Figure 11.

7.3 Case Study: Trellis & ONOS

Trellis [46] is a set of production-grade SDN apps running on ONOS [2,29] to provide control plane logic for multi-purpose L2/L3 leaf-spine fabrics of OF-DPA Broadcom switches. Internally, Trellis uses an ONOS API called FlowObjective, designed to allow portability of apps across different switches by abstracting common L2/L3 functionalities. Trellis controls switches by writing FlowObjectives, which are translated by an ONOS driver into OpenFlow messages for OF-DPA tables. Finally, OF-DPA translates OpenFlow messages to Broadcom SDK calls to populate ASIC-specific tables.

We evaluated Avenir on real-world P4 programs that represent the outermost layers of the architecture described above. The fabric.p4 [12] P4 program was created by the ONOS developers to support Trellis on programmable switches. It is designed to simplify control plane operations, and for this reason it closely resembles the FlowObjective API. Likewise, bcm.p4 [27] abstracts tables from the Broadcom SDK, and was created for Stratum [45], an open source switch agent that uses P4 to model control APIs.

We then collected 40k IPv6 route insertions into fabric.p4 corresponding to a switch reboot load test designed by ONOS engineers. Avenir synthesized insertions into bcm.p4 that equivalently process the IPv6 header and egress specification.

Since Avenir does not process the parser, we simulated its behavior by manually setting the validity bit of the IPv6 header to true, and the IPv4 and MPLS headers to false. Further, the P4 specification [7] leaves the initial values of metadata headers undefined; we manually zero-initialize the metadata fields (a behavior that can be specified for many P4

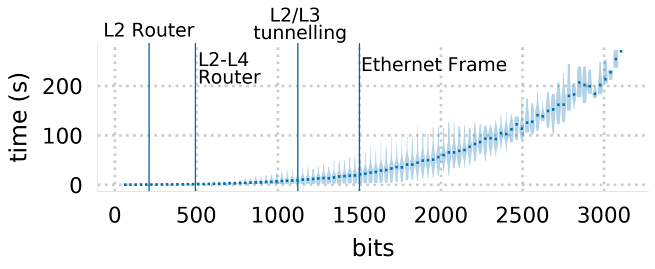


Figure 13: Program bits vs time to translate 100 edits. The vertical lines estimate the sizes of common router programs.

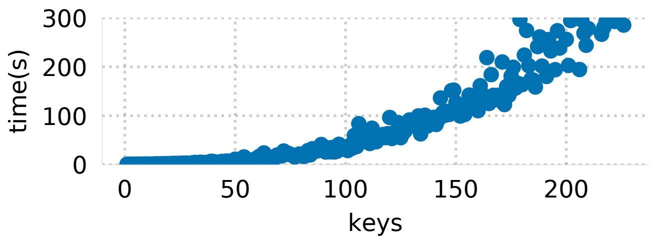


Figure 14: Classifier Scaling. We fixed the number of 32-bit output variables to 8, and varied the number of keys.

targets via a compile time flag).

Further, we modified the l3_fwd in bcm.p4 by swapping the IPv6 matches for IPv4 matches; otherwise there wouldn't have been a valid translation. Finally since Avenir works with parsed headers, we systematically renamed headers in bcm.p4 to match fabric.p4.

The results are shown in Figure 12. According to its engineers, ONOS computes and installs these 40k IPv6 routes over a period of about 15 minutes. This figure includes Trellis' route computation logic, the translation itself and the installation of rules onto the physical target devices. Figure 12 shows that Avenir translates these 40k routes into bcm.p4 pipeline in just under 12 minutes. However, it is unclear what conclusions to draw about overhead, because we don't know how ONOS' translation logic performs. In the (unlikely) best case, we would have no overhead. In the (also unlikely) worst case, we would nearly double the runtime. The real performance would likely be somewhere between these extremes.

7.4 Microbenchmarks

To assess Avenir's scalability, we procedurally generated a collection of microbenchmarks that explore two independent variables, the number of 32-bit input variables I and the number of 32-bit output variables O. For simplicity, the input and output variables sets are distinct.

The abstract pipeline has one table that matches on all of the input variables, and assigns one of the output variables. The target pipeline first matches on all output variables and assigns a metadata value m. This initial staging table is followed by a sequence of O output tables. Table i in this sequence matches solely on m and optionally assigns an output variable.

The results are shown in Figure 13. The x-axis shows the number of bits in the abstract program (i.e., $32(I+O)$) and the y axis shows the time in seconds to translate 100 random abstract edits. The violins show the timing distribution marked with median value. The variation comes from the random generation and from the variation in I and O.

Since networking programs are usually classifier-heavy, we also fixed the number of 32-bit output variables to 8, and varied the size of the classifier. The results are in Figure 14.

Of course, it's difficult to make general claims about the scalability of Avenir's approach, which incorporates numerous heuristics. Nevertheless, it does seem that the complexity increases exponentially with the number of bits, as is expected for a tool that relies on a black-box solver. Target pipelines with different structure than the regular, repeated structure in our microbenchmarks may behave differently.

8 Limitations and Future Work

We discuss two limitations to Avenir's methodology: the cost of formally specificying the abstract and target pipelines, and the run-time overheads of our heuristic search.

The biggest threat to Avenir's use is the requirement that pipelines be formally specified. The work required to develop a formal specification can be significant, and there is no guarantee that a given specification of a pipeline will accurately describe its run-time behavior. Of course, these concerns can be side-stepped if the pipelines are already programmed in P4. But more generally we would need tools for generating specifications and testing conformance. We plan to explore such tools in future work.

Another limitation is our use of heuristic search. The evaluation shows many situations in which Avenir works efficiently, but there are also situations in which it fails to terminate in a reasonable time. For example, to translate from $Pipe_1$ to *OBT* in Figure 2, Avenir maintains a cross product of L2_fwd and L3_fwd, which requires quadratic operations, and causes incremental heursitics to fail. Expanding the effective scope of Avenir's search is future work. We also plan to explore optimal notions of synthesis—e.g., finding the smallest solution.

9 Related Work

Synthesis. Avenir is based on Sketching [39], wherein the programmer is allowed to insert unknown "holes" into a program that are filled using CEGIS [40]. Sketching has been used to build a code generator for packet-processing switch pipelines [16]. NetComplete [10] allows network operators to express their intent by sketching parts of the intended configuration for refactoring or updating purposes. Our novelty is to use sketching to synthesize control plane mappings.

Another use of synthesis is to generate implementations from high-level specifications, e.g., stratified Datalog [11], regular expressions with uninterpreted functions [35], first-order logic constraints [3], and LTL [22].

P4 Verification. There are several recent projects on verifying P4 program properties. Lopes et al. developed an operational semantics for P4 and developed a verification tool based on Datalog which can check program equivalence [24]. P4K presented an operational semantics for P4 using the K framework [19]. p4pktgen uses symbolic execution to generate test cases for P4 programs [26]. ASSERT-P4 translates P4 to a C-like representation, and then symbolically executes the program [15]. Vera [43] uses SymNet [44] as a symbolic execution framework to verify P4 programs. p4v [21] uses symbolic techniques to avoid run-time source traversals.

Network Virtualization. There are many SDN controllers, such as POX [33], NOX [17], and Open Daylight [31]. A few of them specifically target the problem of flow rule composition, including the Frenetic language and controller [14] and Pyretic [25]. Other efforts have focused on network virtualization, i.e., mapping abstract specifications down to target realizations, such as ONIX [20]. FlowVisor [36], CoVisor [18] and the NetKAT compiler [38]. Among this work, Avenir is unique in developing an approach to managing heterogeneous abstract and target pipelines.

10 Conclusion

This paper presented Avenir, a tool that automatically synthesizes control plane operations to ensure uniform behavior across a variety of physical data planes. Avenir uses a counterexample guided inductive synthesis algorithm based on a novel application of sketches to data plane programs. Our evaluation demonstrates that Avenir correctly synthesizes control plane operations with modest overheads.

Acknowledgments

The authors wish to thank Andy Fingerhut for suggesting that we explore the idea of control plane synthesis, and for many helpful discussions as we developed Avenir. Thanks also to Charles Chan and Pier Luigi Ventre from ONF for the discussion on their experience supporting OF-DPA switches in ONOS and Trellis. This work is supported by NSF GRFP grant number DGE-1650441, NSF CCF-1553168, NSF FMiTF-1918396, DARPA HR001120C0107, and gifts from Alibaba, Fujitsu, Infosys, and VMware.

References

[1] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: Semantic Foundations for Networks. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA (POPL)*, pages 113–126, January 2014.

[2] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, and Guru Parulkar. ONOS: Towards an open, distributed SDN OS. In *ACM Workshop on Hot Topics in Software Defined Networking, Chicago, Illinois (HotSDN)*, pages 1–6, August 2014.

[3] Rüdiger Birkner, Dana Drachsler-Cohen, Laurent Vanbever, and Martin T. Vechev. Config2Spec: Mining network specifications from network configurations. In Ranjita Bhagwan and George Porter, editors, *In USENIX Symposium on Networked Systems Design and Implementation, Santa Clara, CA (NSDI)*, pages 969–984, February 2020.

[4] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review (CCR)*, 44(3):87–95, July 2014.

[5] Sean Choi, Boris Burkov, Alex Eckert, Tian Fang, Saman Kazemkhani, Rob Sherwood, Ying Zhang, and Hongyi Zeng. FBOSS: Building switch software at scale. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, Budapest, Hungary (SIGCOMM)*, pages 342–356, August 2018.

[6] P4 Language Consortium. P4Runtime. `https://p4.org/p4-runtime/`, October 2017. Accessed March, 2021.

[7] P4 Language Consortium. P416 Language Specification. `https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.pdf`, 2018. Accessed March, 2021.

[8] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems, Budapest, Hungary (TACAS)*, pages 337–340. Springer, March 2008.

[9] Edsger W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Communications of the ACM (CACM)*, 18(8):453–457, August 1975.

[10] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. NetComplete: Practical Network-Wide Configuration Synthesis with Autocompletion. In *USENIX Symposium on Networked Systems Design and Implementation, Renton, WA (NSDI)*, pages 579–594, April 2018.

[11] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin T. Vechev. Network-Wide Configuration Synthesis. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification, Heidelberg, Germany (CAV)*, volume 10427 of *Lecture Notes in Computer Science*, pages 261–281, 2017.

[12] fabric.p4 source code from ONOS v2.2.2. `https://github.com/opennetworkinglab/onos/blob/2.2.2/pipelines/fabric/impl/src/main/resources/fabric.p4`, 2020. Accessed March, 2021.

[13] Cormac Flanagan and James B Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK (POPL)*, pages 193–205, 2001.

[14] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A Network Programming Language. In *ACM SIGPLAN International Conference on Functional Programming, Tokyo, Japan (ICFP)*, pages 279–291, September 2011.

[15] Lucas Freire, Miguel Neves, Lucas Leal, Kirill Levchenko, Alberto Schaeffer-Filho, and Marinho Barcellos. Uncovering Bugs in P4 Programs with Assertion-based Verification. In *ACM SIGCOMM Symposium on Software Defined Networking Research, Los Angeles, CA (SOSR)*, pages 4:1–4:7, March 2018.

[16] Xiangyu Gao, Taegyun Kim, Michael Dean Wong, Divya Raghunathan, Aatish Kishan Varma, Pravein Govindan Kannan, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. Switch Code Generation using Program Synthesis. In *ACM Special Interest Group on Data Communication, Virtual Event, USA (SIGCOMM)*, pages 44–61, August 2020.

[17] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. NOX: Towards an Operating System for Networks. *ACM SIGCOMM Computer Communication Review (CCR)*, 38(3):105–110, July 2008.

[18] Xin Jin, Jennifer Gossels, Jennifer Rexford, and David Walker. CoVisor: A Compositional Hypervisor for Software-Defined Networks. In *USENIX Symposium on Networked Systems Design and Implementation, Oakland, CA (NSDI)*, pages 87–101, May 2015.

[19] Ali Kheradmand and Grigore Rosu. P4K: A Formal Semantics of P4 and Applications. *Computing Research Repository (CoRR)*, abs/1804.01468, 2018.

[20] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: A Distributed Control Platform for Large-Scale Production Networks. In *USENIX Conference on Operating Systems Design and Implementation, Vancouver, BC (OSDI)*, pages 351–364, October 2010.

[21] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. P4V: Practical Verification for Programmable Data Planes. In *ACM Special Interest Group on Data Communication, Budapest, Hungary (SIGCOMM)*, pages 490–503, August 2018.

[22] Jedidiah McClurg, Hossein Hojjat, Nate Foster, and Pavol Cerný. Event-driven network programming. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, Santa Barbara, CA (PLDI)*, pages 369–385, 2016.

[23] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Computer Communication Review (CCR)*, 38(2):69–74, March 2008.

[24] Nick McKeown, Dan Talayco, George Varghese, Nuno Lopes, Nikolaj Bjørner, and Andrey Rybalchenko. Automatically verifying reachability and well-formedness in P4 Networks. Technical Report MSR-TR-2016-65, September 2016.

[25] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing Software-Defined Networks. In *USENIX Symposium on Networked Systems Design and Implementation, Lombard, IL (NSDI)*, pages 1–14, April 2013.

[26] Andres Nötzli, Jehandad Khan, Andy Fingerhut, Clark Barrett, and Peter Athanas. p4pktgen: Automated Test Case Generation for P4 Programs. In *ACM SIGCOMM Symposium on SDN Research, Los Angeles, CA (SOSR)*, pages 5:1–5:7, March 2018.

[27] Brian O'Connor, Yi Tseng, Maximilian Pudelko, Carmelo Cascone, Abhilash Endurthi, You Wang, Alireza Ghaffarkhah, Devjit Gopalpur, Tom Everman, Tomek Madejski, et al. Using P4 on Fixed-Pipeline and Programmable Stratum Switches. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems, Los Angeles, CA (ANCS)*, pages 1–2, October 2019.

[28] ONOS : Open Network Operating System. https://github.com/opennetworkinglab/onos/commit/b7b79af9702f03c1286b8f2f9d98e6b87b29c467. Accessed March, 2021.

[29] Open Network Operating System (ONOS) SDN controller for SDN/NFV solutions. https://www.opennetworking.org/onos. Accessed March, 2021.

[30] OpenConfig. https://www.openconfig.net. Accessed March, 2021.

[31] OpenDaylight. https://www.opendaylight.org. Accessed March, 2021.

[32] OpenFlow-Data Plane Abstraction Networking Software. https://www.broadcom.com/products/ethernet-connectivity/software/of-dpa. Accessed March, 2021.

[33] The POX OpenFlow Controller. https://github.com/noxrepo/pox/. Accessed March, 2021.

[34] QMX switches require the unicast flow being installed before multicast flow in TMAC table. https://github.com/opennetworkinglab/onos/commit/45b69ab951915a4211a. Accessed March, 2021.

[35] Shambwaditya Saha, Santhosh Prabhu, and P. Madhusudan. NetGen: synthesizing data-plane configurations for network policies. In Jennifer Rexford and Amin Vahdat, editors, *ACM SIGCOMM Symposium on Software Defined Networking Research, Santa Clara, CA (SOSR)*, pages 17:1–17:6, June 2015.

[36] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. Can the Production Network Be the Testbed? In *USENIX Conference on Operating Systems Design and Implementation, Vancouver, BC (OSDI)*, pages 365–378, October 2010.

[37] SLOCCount. https://dwheeler.com/sloccount/. Accessed March, 2021.

[38] Steffen Smolka, Spiridon Eliopoulos, Nate Foster, and Arjun Guha. A Fast Compiler for NetKAT. In *ACM SIGPLAN International Conference on Functional Programming, Vancouver, BV (2015)*, pages 328–341, August 2015.

[39] Armando Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, 2008.

[40] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodík. Sketching Concurrent Data Structures. In *ACM SIGPLAN Notices*, pages 136–148, June 2008.

[41] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial Sketching for Finite Programs. In *ACM Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA (ASPLOS)*, pages 404–415, 2006.

[42] SONiC. https://azure.github.io/SONiC/. Accessed March, 2021.

[43] Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Debugging P4 Programs with Vera. In *ACM Special Interest Group on Data Communication, Budapest, Hungary (SIGCOMM)*, pages 518–532, 2018.

[44] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. SymNet: Scalable Symbolic Execution for Modern Networks. In *ACM Special Interest Group on Data Communication, Florianopolis, Brazil (SIGCOMM)*, pages 314–327, August 2016.

[45] Stratum: enabling the era of next generation SDN. https://www.opennetworking.org/stratum/. Accessed March, 2021.

[46] Trellis platform brief. https://www.opennetworking.org/wp-content/uploads/2019/09/TrellisPlatformBrief.pdf. Accessed March, 2021.

[47] Konstantin Weitz, Stefan Heule, Waqar Mohsin, Lorenzo Vicisano, and Amin Vahdat. Leveraging P4 for Fixed-Function Switches. In *P4 Workshop 2019*, 2019.

A Formal Details

The grammar of bit-vector expressions and boolean formulae is described in Figure 15.

$$
\begin{array}{rcll}
e & ::= & & (e \in \mathsf{Expr}) \\
 & \mid & v & \textit{Bit Vector} \\
 & \mid & f & \textit{Field} \\
 & \mid & e - e & \textit{Subtraction} \\
 & \mid & e + e & \textit{Addition} \\
 & \mid & e \,\&\, e & \textit{Bitwise And} \\
 \\
b & ::= & & (b \in \mathsf{Bool}) \\
 & \mid & \mathsf{tt} & \textit{Truth} \\
 & \mid & \mathsf{ff} & \textit{Falsehood} \\
 & \mid & e = e & \textit{Equality} \\
 & \mid & e < e & \textit{Inequality} \\
 & \mid & \neg b & \textit{Negation} \\
 & \mid & b \wedge b & \textit{Conjuction} \\
 & \mid & b \vee b & \textit{Disjunction}
\end{array}
$$

Figure 15: Syntax of BitVector expressions and boolean formulae.

The semantics of edits are defined below

$$
\begin{array}{rcl}
\mathsf{A}_x(\rho)\ \tau & \triangleq & \begin{cases} \tau\{x \mapsto \tau(x) @ [\rho]\}, & \forall \rho \in \tau(x), \\ & \pi_1(\rho) \neq \pi_1(\rho') \\ \tau, & \textit{otherwise} \end{cases} \\
\mathsf{D}_x(i)\ \tau & \triangleq & \tau\{x \mapsto \tau(x)[0:i] @ \tau(x)[i+1:]\} \\
\vec{\delta}\ \tau & \triangleq & \delta_1 \circ \cdots \circ \delta_n\ \ \tau
\end{array}
$$

Instantiations. A table is populated by rows in Row. A single row $\rho = (\vec{m}, \vec{d}, a) \in \mathsf{Row}$ is a tuple comprising a sequence of match values $\vec{m}$, a sequence of action data $\vec{d}$, and an action index i. We write $\pi_1(\rho) = \vec{m}$, $\pi_2(\rho)$ for $\vec{d}$, and $\pi_3(\rho)$ for a.

Note that our instantiations only allow exact matches on data. This does not affect the generality of our formal results, since exact matches can easily encode ternary and lpm matches (with untenable blowup, of course). Using these more compact matches is an optimization we describe in Section 5.

A row is *well-formed* for a table t if $|\vec{m}| = t.keys$, $i < |t.actions|$, and when $t.actions[i] = \lambda\vec{x}.\ c$, then $|\vec{x}| = |\vec{d}|$. In practice, there are additional typing constraints regarding the sizes of the bitvectors, but as we've abstracted bitvectors to naturals in this exposition, we can set that bookkeeping aside. We assume henceforth that all rows are well-formed for the tables into which they are being installed.

An instantiation $\tau \in \mathsf{Inst}$ is a function from table name to sequences of rows that describes the given state of the tables in a pipeline, i.e., given a table t, $\tau(t.name)$ gives us the sequence of rows in the table. We often write $\tau(t)$ as convenient shorthand. Moving forwards we will use τ to describe instantiations for abstract programs and σ to describe instances for physical programs.

An instantiation is well-formed if every row in every table is well-formed

Now that we have pipelines c and instantiations τ, we can define how to combine them via the function $\mathsf{subst}(c,\tau)$, which produces another command with no tables in it. Effectively we replace a table t with a guarded command that checks each row $(\vec{m}, \vec{d}, i)$ in sequence. A single row is translated to the guarded command: $\mathsf{encKeys}(t.keys, \vec{m}) \to t.actions[i](\vec{d})$, where the encKeys function We present this formally in Figure 16. Since instances τ are total functions, $\tau(t)$ will always be defined.

We call the command $c' = \mathsf{subst}(c,\tau)$ an *instantiated pipeline*. Note that all of the table applications have been encoded away and we have a simple loop-free command.

Interpretation. We can interpret instantiated pipelines as functions on packets. A packet comes in and then a (possibly) different packet goes out. Similar to other formalisms of packet processing functions we define packets to be valuations on the headers and metadata [1]: packets are defined as finite maps $\mathsf{Hdr} \cup \mathsf{Meta} \rightharpoonup (\mathsf{BitVec})$. Operations on packets pkt are the standard ones: the empty packet is written $\{\}$; to update or set the value of $h \in \mathsf{Hdr}$ to $[n]_s \in \mathsf{BitVec}$ with $h.size = s$, write $pkt\{h \mapsto [n]_s\}$, otherwise the update is undefined; to access the value of x, write $pkt.x$. The set of defined names x in a packet pkt is denoted $\mathrm{dom}(pkt)$. A packet is *well-formed* when it can be constructed by a series of defined updates. In what follows, we assume all updates are well-formed and all packets well-defined: specifically, that Pkt the set of well-defined elements of $\mathsf{Hdr} \rightharpoonup (\mathsf{BitVec})$.

The semantics of commands on a packet pkt are straightforward. The assignment operation $x := e$ first evaluates e to a value n in the environment defined by the packet pkt, and then returns the packet $pkt\{x \mapsto n\}$. The sequence operator $(c_1; c_2)$ is simply interpreted as functional composition: the output of $[\![c_1]\!]\ pkt$ is passed into $[\![c_2]\!]$.

The semantics of guarded commands are broken into two cases. First, if there are no rows in the selection, i.e., if fi, the command is interpreted as the identity function, otherwise, if there is at least one row in the selection (i.e., if $(b \to c)\ \overrightarrow{b \to c}$ fi), then b is evaluated in the packet environment. If it evaluates to tt, then execute c, i.e., $[\![c]\!]\ pkt$, otherwise, if b evaluates to ff, we simply check the remaining rows in the guarded command, i.e $[\![\text{if } \overrightarrow{b \to c} \text{ fi}]\!]\ pkt$.

Finally, the denotation of a table is simply its default action.

We leave the semantics of our expressions ($\mathcal{E}[\![-]\!]$) and booleans ($\mathcal{B}[\![-]\!]$) undefined, as their definitions are standard.

The logical encoding uses the predicate transformer semantics of GCL programs [9]. Our semantics are identical to Dijkstra's, with the exception of our guarded commands. Ours are definitionally mutually exclusive, his are nondeterministic. To remedy this we simply negate the preceding guards, and then apply the weakest precondition function, *wp*.

$$\begin{array}{rcl}
\Phi & \triangleq & \bigwedge\{h.name = x' \mid h \in x' \text{ fresh}\} \\
c_1 \equiv c_2 & \triangleq & wp(c_1, \Phi) \Leftrightarrow wp(c_2, \Phi) \\
\mathsf{ISVALID}(\varphi) & \triangleq & \mathsf{SAT}(\neg\varphi)
\end{array}$$

Figure 18: The verification condition for determining when two programs implement the same function

$$\begin{array}{lcl}
\mathsf{encKeys} :: \mathsf{List}[\mathsf{Hdr}] \times \mathsf{List}[\mathsf{BitVec}] \to \mathsf{Bool} & & \\
\mathsf{encKeys}(\vec{k}, \vec{m}) & \triangleq & \bigwedge_{0 \le i < |t.keys|} k_i = m_i \\
 & & \\
\mathsf{subst}(c, \tau) :: \mathsf{Cmd} & & \\
\mathsf{subst}(x := e, \tau) & \triangleq & x := e \\
\mathsf{subst}(c; c, \tau) & \triangleq & \mathsf{subst}(c, \tau); \mathsf{subst}(c, \tau) \\
\mathsf{subst}(\mathsf{if}\ \overrightarrow{b \to c}\ \mathsf{fi}, \tau) & \triangleq & \mathsf{if}\ \overrightarrow{b \to \mathsf{subst}(c, \tau)}\ \mathsf{fi} \\
\mathsf{subst}(\mathsf{apply}\ t, \tau) & \triangleq &
\end{array}$$

```
    if
      $\mathsf{encKeys}(t.keys, \vec{m}_1) \to t.actions[a_1](\vec{d}_1)$
                    $\vdots$
      $\mathsf{encKeys}(t.keys, \vec{m}_n) \to t.actions[a_n](\vec{d}_n)$
    fi

    where $(\vec{m}_1, \vec{d}_1, a_1), \ldots, (\vec{m}_n, \vec{d}_n, a_n) = \tau(t)$
```

Figure 16: Semantics of Table Instances

$$\begin{array}{lcl}
[\![c]\!] :: \mathsf{Pkt} \to \mathsf{Pkt} & & \\
[\![x := e]\!]\ pkt & \triangleq & pkt\{x \mapsto \mathcal{E}[\![e]\!]\ pkt\} \\
[\![c_1; c_2]\!]\ pkt & \triangleq & [\![c_2]\!] \circ [\![c_1]\!]\ pkt \\
[\![\mathsf{if}\ \ \mathsf{fi}]\!]\ pkt & \triangleq & pkt \\
[\![\mathsf{if}\ (b \to c)(\overrightarrow{b \to c})\mathsf{fi}]\!]\ pkt & & \\
 & \triangleq & \begin{cases} [\![c]\!]\ pkt & \mathcal{B}[\![b]\!]\ pkt = \mathrm{tt} \\ [\![\mathsf{if}\ \overrightarrow{b \to c}\ \mathsf{fi}]\!]\ pkt & \text{otherwise} \end{cases} \\
[\![\mathsf{apply}(t)]\!]\ pkt & \triangleq & [\![t.default]\!]\ pkt
\end{array}$$

Figure 17: Semantics of programs

The verification condition is defined in Figure 18. Leveraging the connection between weakest preconditions and denotational semantics [9], we can define equivalence pipelines c_1 and c_2 by taking the weakest precondition with respect to a particular formula Φ, which equates every header variable to a fresh symbolic variable. The free variable will then capture the input value of a program, and the symbolic variable will capture the output value of the program.

As an example, consider the following command c:

```
if   $x = 1$   $\to$   $x := 5$
       tt   $\to$   $x := 9$
fi
```

Then the Φ corresponding to this command is $x = x'$. Then $wp(c, \Phi)$ is equivalent to

$$\begin{array}{l}
(x = 1 \Rightarrow 5 = x') \\
\wedge (x \neq 1 \Rightarrow 9 = x')
\end{array}$$

The variable x captures the input conditions (either $x = 1$ or not), and the x's capture the output varibles: when the input x value is one, the output will be 5; otherwise, it will be 9.

Instrumentation The instr function formalizes how to add holes to a table. We write else if to help delineate elements in the selection list.

```
instr$(t, \tau, \vec{\delta}, n)$   $\triangleq$
  if $t.keys = \vec{m}_i \wedge ?\mathtt{Del}_{t.name,i} = 0 \to t.actions[a_i](\vec{d}_i)$
                    for $(\vec{m}_i, \vec{d}_i, a_i) \in \tau(t)$ if $\mathsf{D}(t.name, i) \notin \vec{\delta}$
  else if $t.keys = \vec{m} \to t.actions[a_i](\vec{d})$
                              for $\mathsf{A}(t.name, (\vec{m}_i, \vec{d}_i, a_i)) \in \vec{\delta}$
  else if
    $\left(\begin{array}{l} \bigwedge_{k \in t.keys} k = ?\mathtt{k}_{t.name,j} \\ \wedge ?\mathtt{Add}_{t.name,j} = 1 \\ \wedge \bigwedge_{k \le j} ?\mathtt{Add}_{t.name,k} = 1 \\ \wedge ?\mathtt{ActId}_{t.name,j} = i \end{array}\right) \to t.action[i](\overrightarrow{?d_{t.name,j,i}})$
                    for $0 \le i < |t.actions|$ and $0 \le j < n$
  fi
```

Figure 19: The instr function formalizes how to add holes to a tables

B Proof of Completeness

Lemma 1 (Hits Restrict)**.** *For every program $p \in \mathsf{Cmd}$, instance $\sigma \in \mathsf{Inst}$, and set of counter examples X such that for every $(\chi_0, \chi_1) \in X$, $[\![\mathsf{subst}(p,\sigma)]\!]\ \chi_0 = \chi_1$, then there exists σ' such that $X \subseteq [\![\mathsf{subst}(p,\sigma')]\!]$ and $|\sigma'(t)| \leq |X|$ for every $t \in \mathsf{Tables}(p)$.*

Proof. Proceed by induction on the structure of p:

[$(p = x := e)$] Trivial because $\mathsf{Tables}(x := e) = \emptyset$ and for every σ, $\mathsf{subst}(x := e, \sigma) = x := e$.

[$p = (p_1; p_2)$] Assume $p = p_1; p_2$. Introduce X and σ as above.

Decompose X across p_1 and p_2 such that

$$\begin{array}{lll} X_1 & \triangleq & \{(\chi_0,\chi_1) \mid [\![\mathsf{subst}(p_1,\sigma)]\!]\ \chi_0 = \chi_1, \chi_0 \in \pi_1(X)\} \\ X_2 & \triangleq & \{(\chi_1,\chi_2) \mid [\![\mathsf{subst}(p_2,\sigma)]\!]\ \chi_1 = \chi_2, \chi_1 \in \pi_2(X_1)\} \end{array}$$

Notice that $X_2 \circ X_1 = X$.

Now by the IH on p_1, σ and X_1 we get σ_1 such that every rule is hit by a counterexample in X_1,

$$\forall(\chi_0,\chi_1) \in X_1, [\![\mathsf{subst}(p_1,\sigma_1)]\!]\ \chi_0 = \chi_1$$

and $|\sigma_1(t)| \leq |X_1|$ for all $t \in \mathsf{Tables}(p_1)$. Similarly, by the IH on p_2, σ and X_2, we get σ_2 such that every rule is hit by a counterexample in X_2.

$$\forall(\chi_1,\chi_2) \in X_2, [\![\mathsf{subst}(p_2,\sigma_2)]\!]\ \chi_1 = \chi_2$$

and $|\sigma_2(t)| \leq |X_1|$ for all $t \in \mathsf{Tables}(p_2)$.

Now we can construct σ' such that

$$\sigma'(t) = \begin{cases} \sigma_1(t), & \text{if } t \in \mathsf{Tables}(p_1) \\ \sigma_2(t), & \text{if } t \in \mathsf{Tables}(p_2) \end{cases}$$

Observe that $\sigma|_{\mathsf{Tables}(p_i)} = \sigma_i$ for $i = 1,2$. Now we show the two remaining properties.

- Since $X = X_2 \circ X_1$ is a function, we know $\forall t \in \mathsf{Tables}(p)$, $|\sigma_1(t)| \leq |X_1| \leq |X|$ and $|\sigma_2(t)| \leq |X_2| \leq |X|$, and the table names in p_1 and p_2 are disjoint. Conclude that $\forall t \in \mathsf{Tables}(p), |\sigma'(t)| \leq |X|$.
- Show $X \subseteq [\![\mathsf{subst}(p_1;p_2,\sigma')]\!]$. Our IHs give

  $$\begin{array}{l} X_1 \subseteq [\![\mathsf{subst}(p_1,\sigma_1)]\!] \\ X_2 \subseteq [\![\mathsf{subst}(p_2,\sigma_2)]\!] \end{array}$$

  By definition,

  $$\begin{array}{c} [\![\mathsf{subst}(p_1;p_2,\sigma')]\!] \\ = \\ [\![\mathsf{subst}(p_2,\sigma')]\!] \circ [\![\mathsf{subst}(p_1,\sigma')]\!] \end{array}$$

  Then,

  $$\begin{array}{c} [\![\mathsf{subst}(p_2,\sigma')]\!] \circ [\![\mathsf{subst}(p_1,\sigma')]\!] \\ = \\ [\![\mathsf{subst}(p_2,\sigma_2)]\!] \circ [\![\mathsf{subst}(p_1,\sigma_1)]\!] \end{array}$$

  by the disjointness of table names. The result follows.

[$p = \mathsf{if}\ \overrightarrow{b \to p}\ \mathsf{fi}$] Similar, but n-ary.

[$p = \mathsf{apply}(t)$] Assume that $p = \mathsf{apply}(t)$ for some table t. The corresponding rows for the table are $\sigma(t) = \vec{\rho}$. Compute a subsequence $\vec{\rho}'$ of $\vec{\rho}$ such that $\vec{\rho}'$ contains ρ_i iff there is some input packet in X that hits ρ_i. Create an instantiation $\sigma' = \sigma\{t \mapsto \vec{\rho}'\}$. Since each the rules are equality matches (and hence disjoint), $[\![\mathsf{subst}(\mathsf{apply}(t),\sigma')]\!]\ \chi_0 = \chi_1$ for every $(\chi_0,\chi_1) \in X$. Since some $\chi_0 \in \pi_1(X)$ may miss, conclude that $|\sigma'(t)| \leq |X|$. □

Lemma 2 (Model Solution)**.** *For every $p \in \mathsf{Cmd}$, every $\sigma \in \mathsf{Inst}$, and every $X \in \mathsf{Pkt}^2$, if there exists $\vec{\delta} \in \mathsf{List[Edit]}$ such that $X \subseteq [\![\mathsf{subst}(p,\vec{\delta}(\sigma))]\!]$, then*

$$\mathsf{SAT}\ \vec{\delta}' = \mathsf{model}(p,\sigma,X)$$

and

$$X \subseteq [\![\mathsf{subst}(p,\vec{\delta}'(\sigma))]\!]$$

Proof. Let $\vec{\delta}$ be such that $X \subseteq [\![\mathsf{subst}(p,\vec{\delta}(\sigma))]\!]$.

Lemma 1 gives us σ' from $\vec{\delta}(\sigma)$ such that for every table $t \in \mathrm{dom}(\sigma')$, $|\sigma'(t)| \leq |X|$, and $X \subseteq [\![\mathsf{subst}(p,\sigma')]\!]$.

Construct the following witness to the query in Figure 5

$$\left\{ \begin{array}{l|l} ?\mathtt{Del}_{x,i} \mapsto 1, & t \in \mathsf{Tables}(p), \\ ?\mathtt{Add}_{x,j} \mapsto 1, & x = t.name \\ ?\mathtt{ActId}_{x,j} \mapsto a, & 0 \leq i < |\sigma(t)|, \\ \overrightarrow{?k_{x,j} \mapsto m}, & 0 \leq j < |\sigma'(t)| \\ \overrightarrow{?d_{x,j,i} \mapsto d} & (\vec{m},\vec{d},a) = \sigma'(t)[j] \end{array} \right\}$$

which corresponds to the rules

$$\begin{array}{llll} d_{t,i} & \triangleq & \mathsf{D}(t,i), & \forall t \in \mathsf{Tables}(p), |\sigma(t)| > i \geq 0 \\ a_{t,j} & \triangleq & \mathsf{A}(t,\sigma'(t)[j]), & \forall t \in \mathsf{Tables}(p), 0 \leq j < |\sigma'(t)| \\ \vec{\delta}' & \triangleq & \vec{d} \cdot \vec{a} & \end{array}$$

Note that $\vec{d}_t$ is sorted from highest index to lowest index. Observe that $\vec{\delta}'(\sigma) = \vec{a}(\vec{d}(\sigma))\sigma = \vec{a}(\cdot) = \sigma'$, so we can conclude that $\forall(\chi_0,\chi_1) \in X, [\![\mathsf{subst}(p,\vec{\delta}'(\sigma))]\!]\ \chi_0 = \chi_1$, and we're done. □

Theorem 6 (Completeness)**.** *For every logical program $l \in$ Cmd, physical program $p \in \mathsf{Cmd}$, logical instantiation $\tau \in$ Inst, physical instantiation $\sigma \in \mathsf{Inst}$, sequence of physical edits $\vec{\delta} \in \mathsf{List[Edit]}$, and every set X s.t.*

$$X \subseteq [\![\mathsf{subst}(l,\tau)]\!] \cap [\![\mathsf{subst}(p,\vec{\delta}(\sigma))]\!],$$

if

$$\exists \vec{\varepsilon} \in \mathsf{List[Edit]}.\mathsf{subst}(l,\tau) = \mathsf{subst}(p,\vec{\varepsilon}(\sigma))$$

then

$$\mathsf{Ok}\ \vec{\delta}' = \mathsf{cegis}(l,p,\tau,\sigma,\vec{\delta},X)$$

and

$$\mathsf{subst}(l,\tau) = \mathsf{subst}(p,\vec{\delta}'(\sigma))$$

Proof. Proceed by induction on $|\mathsf{Pkt} \setminus \pi_1(X)|$.

Base Case. $(\pi_1(X) = \mathsf{Pkt})$. The definition of functional subset gives us

$$\forall(\chi_1,\chi_2) \in X.[\![\mathsf{subst}(l,\tau)]\!]\ \chi_1 = \chi_2 = [\![\mathsf{subst}(p,\delta(\vec{\sigma}))]\!]\ \chi_1$$

which reduces to

$$\forall \chi_1 \in \pi_1(X).[\![\mathsf{subst}(l,\tau)]\!]\ \chi_1 = [\![\mathsf{subst}(p,\vec{\delta}(\sigma))]\!]\ \chi_1.$$

Then by the assumption that $\pi_1(X) = \mathsf{Pkt}$, this is just

$$\forall \chi_1 \in \mathsf{Pkt}.[\![\mathsf{subst}(l,\tau)]\!]\ \chi_1 = [\![\mathsf{subst}(p,\vec{\delta}(\sigma))]\!]\ \chi_1$$

Conclude, by definition, that

$$\mathsf{subst}(l,\tau) = \mathsf{subst}(l,\vec{\delta})$$

Inductive Step. $(\pi_1(X) \subseteq \mathsf{Pkt})$. If $\mathsf{subst}(l,\tau) = \mathsf{subst}(p,\vec{\delta}(\sigma))$, then we're done, so assume instead that we have a counterexample $\chi = (\chi_0,\chi_1)$ such that

$$[\![\mathsf{subst}(l,\tau)]\!]\ \chi_0 = \chi_1 \neq [\![\mathsf{subst}(p,\vec{\delta}(\sigma))]\!]$$

which tells us that that $\chi \notin X$.

Our main assumption tells us that there is some $\vec{\varepsilon}$ such that

$$\mathsf{subst}(l,\tau) = \mathsf{subst}(p,\vec{\varepsilon}(\sigma))$$

By definition we have

$$\forall pkt \in \mathsf{Pkt}.[\![\mathsf{subst}(l,\tau)]\!]\ pkt = [\![\mathsf{subst}(p,\vec{\varepsilon}(\sigma))]\!]\ pkt.$$

which, since $\pi_1(\{\chi\} \cup X) \subseteq \mathsf{Pkt}$, means that

$$\forall pkt \in \{\chi\} \cup X.[\![\mathsf{subst}(l,\tau)]\!]\ pkt = [\![\mathsf{subst}(p,\vec{\varepsilon}(\sigma))]\!]\ pkt,$$

Then Lemma 2 gives us a model $\vec{\delta}'$ such that

$$\forall pkt \in \{\chi\} \cup X.[\![\mathsf{subst}(l,\tau)]\!]\ pkt = [\![\mathsf{subst}(p,\vec{\delta}'(\sigma))]\!]\ pkt$$

The result follows by IH on $\{\chi\} \cup X$ as $|\{\chi\} \cup X| > |X|$. □

C Proof of Completeness for Incremental Synthesis

Definition 1 (Minimal Sequence)**.** *Given an instantiation τ, a sequence of edits $\vec{\delta}$ is minimal iff for every table $t \in \mathrm{dom}(\tau)$, for every $\vec{\delta}'$ s.t. $\vec{\delta}(\sigma) = \vec{\delta}'(\sigma)$, $|\vec{\delta}| \leq |\vec{\delta}'|$.*

Definition 2 (Minimal Extension)**.** *Given a command c, an instantiation τ, and a sequence of edits $\vec{\delta}$, another sequence $\vec{\delta}'$ is called a* minimal extension *of $\vec{\delta}$ when $\vec{\delta} \circ \vec{\delta}'$ is minimal.*

Lemma 3 (Finite Minimal Sequences)**.** *Given a command c, and an instantiation τ, the set of minimal sequences is finite.*

Proof. Proceed by induction on the structure of c.

$(h := e)$ There there is one minimal instatiation: $[]$.

$(c_1; c_2)$ By IHs, the set of minimal sequences corresponding to c_1 and τ is a finite D_1, and the set of minimal sequences corresponding to c_1 and τ is a finite D_2. The set of minimal sequences corresponding to $c_1; c_2$ is the set of all interleaving of all sequences in D_1 and sequences of D_2, which is finite.

$(\mathsf{if}\ \overrightarrow{b \to c}\ \mathsf{fi})$ This is similar to the previous case except $n$ary.

$(\mathsf{apply}(t))$ First we show that the space of functions that t can represent is finite. Since $t.keys$ is finite, and each header in $t.keys$ has a finite domain of values, so the domain of matches is finite. A similar argument shows that the domain of actions is finite, and so the domain of functions is finite. Further for each of these functions, there are finitely many ways of representing them (all of the permutations of the rules).

This means that there are finitely many minimal sequences when $\tau(t) = []$, simply install each of these, with no duplicates, since removing the duplicates would result in a smaller sequence of edits.

When $\tau(t) \neq []$ there are also finitely many minimal sequences. They are those that delete as few rules in $\tau(t)$ as necessary and then install the missing rules (if any are needed): they never delete a rule and then reinstall the same rule, and then never install a rule only to delete it later. □

Lemma 4 (Finite Minimal Extensions)**.** *For a command c and an instantiation τ, and a sequence of edits $\vec{\delta}$, there are finitely many minimal extensions of $\vec{\delta}$.*

Proof. There are finitely many minimal extensions of c and τ. Some of those have $\vec{\delta}$ as a prefix; there are finitely many of them. □

Definition 3 (φ-Preclusion). *For a program c, instantiation τ, and a packet pair $\chi \notin [\![\mathsf{subst}(c,\tau)]\!]$, we say that φ precludes $\vec{\delta}$ when for every $\vec{\delta'} \models \varphi$ that is also a prefix of $\vec{\delta}$, $\chi \notin [\![\mathsf{subst}(p,\vec{\delta}(\tau))]\!]$. We say that* φ precludes a solution, *when it precludes every $\vec{\delta}$.*

Definition 4 (Zip Function). *Let* zip*($\vec{x}$, $\vec{y}$) be the function that simultaneously iterates through $\vec{x}$ and $\vec{y}$ and returns a sequence of the element-wise pairs, i.e., $\overrightarrow{(x,y)}$. The function is undefined if $|\vec{x}| \neq |\vec{y}|$.*

Lemma 5 (Model Finding). *For every logical program l, physical program p, logical instantiation τ, phsyical instantiation σ, sequence of physical edits $\vec{\delta}$, counterexample $\chi \in [\![\mathsf{subst}(l,\tau)]\!] \setminus [\![\mathsf{subst}(p,\vec{\delta}(\sigma))]\!]$, and formula $\varphi \in \mathsf{Bool}$ such that for every $\vec{\delta'} \models \neg\varphi$ st $\chi \in [\![\mathsf{subst}(p,\vec{\delta'}(\vec{\delta}(\sigma)))]\!]$, there is no extension $\vec{\delta''}$ such that $\mathsf{subst}(l,\tau) = \mathsf{subst}(l,\vec{\delta''}(\vec{\delta'}(\vec{\delta}(\sigma))))$, then if there exists $\vec{\varepsilon}$ not precluded by φ such that*

$$\mathsf{subst}(p,\tau) = \mathsf{subst}(l,\vec{\varepsilon}(\vec{\delta}(\sigma)))$$

, and $\vec{\delta} \circ \vec{\varepsilon}$ is minimal, then

$$\mathsf{Sat}\ \vec{\delta''} = \mathsf{model}'(p,\sigma,\delta,\chi,\varphi)$$

Proof. Let l, p, τ, σ, $\vec{\delta}$, χ, and φ be given. Construct the following model for the query in $\mathsf{model}'(p,\sigma,\delta,\chi,\varphi)$:

We accumulate a model for every table $t \in \mathsf{Tables}(p)$. If $\pi_1(\chi)$ in $\mathsf{subst}(p,\vec{\varepsilon}(\vec{\delta}(\sigma)))$ hits the ith row ρ_i, then there are several cases

MISS If $\pi_1(\chi)$ misses in $\mathsf{subst}(p,\vec{\delta}(\sigma))$, then add

$$\left\{\begin{array}{l} ?\mathsf{AddRowTo}_{t,1} \mapsto 1 \\ ?\mathsf{Act}_{t,1} \mapsto \rho_i.action \\ ?\vec{\mathsf{k}}_{t,1} \mapsto \rho_i.keys \\ \overrightarrow{?\mathsf{d}_{t,\rho.action,1}} \mapsto \rho_i.data \end{array}\right\}$$

HITCORRECT If $\pi_1(\chi)$ hits the jth row ρ_j of $(\vec{\delta}(\sigma))(t)$ in $\mathsf{subst}(p,\vec{\delta}(\sigma))$, and $\rho_i = \rho_j$ then do nothing.

HITWRONG If $\pi_1(\chi)$ hits the jth row of $(\vec{\delta}(\sigma))(t)$ in $\mathsf{subst}(p,\vec{\delta}(\sigma))$ and $\rho_j \neq \rho_i$,

$$\left\{\begin{array}{l} ?\mathsf{AddRowTo}_{t,1} \mapsto 1 \\ ?\mathsf{Act}_{t,1} \mapsto \rho_i.action \\ ?\vec{\mathsf{k}}_{t,1} \mapsto \rho_i.keys \\ \overrightarrow{?\mathsf{d}_{t,\rho.action,1}} \mapsto \rho_i.data \end{array}\right\}$$

From here we extract queries from the model as before, sorting the deletions by decreasing index to get $\vec{\delta''}$. Note that χ hits a syntactically equivalent rule in $\mathsf{subst}(p,\vec{\delta''}(\vec{\delta})(\sigma))$ as in $\mathsf{subst}(l,\vec{\varepsilon}(\vec{\delta}(\sigma)))$. We conclude $\chi \in [\![\mathsf{subst}(p,\vec{\delta''}(\vec{\delta}(\sigma)))]\!]$ □

Lemma 6 (Nontrivial Models). *For every physical program p, physical instance σ, sequence of edits $\vec{\delta}$, formula φ, and counterexample $\chi \in \mathsf{Pkt}^2$ such that $\chi \notin [\![\mathsf{subst}(p,\vec{\delta}(\sigma))]\!]$, if $\mathsf{Sat}\ \vec{\delta''} = \mathsf{model}'(p,\sigma,\vec{\delta},\chi,\varphi)$, then $\vec{\delta''} \neq []$.*

Proof. Let $p \in \mathsf{Cmd}$, $\sigma \in \mathsf{Inst}$, $\vec{\delta} \in \mathsf{List[Edit]}$ and φ and $\chi \in \mathsf{Pkt}^2$ be given. Assume $\mathsf{Sat}\ \vec{\delta''} = \mathsf{model}'(p,\sigma,\vec{\delta},\varphi)$.

Prove the contrapositive, that if $\vec{\delta''} = []$, then $\chi \in [\![\mathsf{subst}(p,\sigma)]\!]$. Assume $\vec{\delta''} = []$. Then the query in $\mathsf{model}(p,\sigma,\vec{\delta},\chi,\varphi)$ is satisfiable with all deletion and insertion holes set to zero. This means that the following query is also satisfiable (where $\chi = (pkt, pkt')$):

$$\mathsf{SAT}\left(\forall\vec{x}.\varphi \wedge \left(\overrightarrow{\chi_1.x = x}\right) \Rightarrow wp\left(\mathsf{subst}(p,\vec{\delta}(\sigma)), \left(\overrightarrow{pkt'.x = x}\right)\right)\right)$$

By the correspondence between wp and the denotational semantics, conclude that $\chi \in [\![\mathsf{subst}(p,\vec{\delta}(\sigma))]\!]$. □

Proposition 1 (Oracle Constraints). *For a given physical program p, edit sequence δ and instance σ,*

$$\mathsf{HEURISTIC}() \Rightarrow \bigwedge_{t \in \mathsf{Tables}(p)} ?\mathsf{Add}_{t,1} = 1 \Rightarrow \bigwedge_{(\vec{m},\vec{d},\vec{a}) \in \vec{\delta}(\sigma)(t)} \neg\left(\overrightarrow{?\mathsf{k}_{t,1}} = m\right)$$

and

$$\mathsf{HEURISTIC}() \Rightarrow \bigwedge_{\substack{t \in \mathsf{Tables}(p) \\ 0 \le i < |\sigma(t)| \\ (\vec{m},\vec{d},a) = \sigma(t)[i]}} ?\mathsf{Del}_{t,i} = 1 \Rightarrow \neg\left(\begin{array}{l} \overrightarrow{?\mathsf{k}_{t,1} = m} \\ \wedge ?\mathsf{Act}_{t,l} = a \\ \wedge \overrightarrow{?\mathsf{d}_{t,a,1} = d} \end{array}\right)$$

Lemma 7 (Reachable Edits). *For every program p, instantiation σ, edit sequence $\vec{\delta}$, counterexample χ and $\varphi \in \mathsf{Bool}$, if*

$$\mathsf{Sat}\ \vec{\delta'} = \mathsf{model}'(p,\sigma,\vec{\delta},\chi,\varphi)$$

then the insertions in $\vec{\delta}$ are reachable.

Proof. By Proposition 1. □

Lemma 8 (Deletes Not Resurrected). *For every program p, instantiation σ, edit sequence $\vec{\delta}$, counterexample χ and $\varphi \in \mathsf{Bool}$, if*

$$\mathsf{Sat}\ \vec{\delta'} = \mathsf{model}'(p,\sigma,\vec{\delta},\chi,\varphi)$$

then the insertions in $\vec{\delta}$ are reachable.

Proof. By Proposition 1. □

Lemma 9 (Minimal Models). *For every physical program p, physical instance σ, sequence of edits $\vec{\delta}$, formula φ, and counterexample $\chi \in \mathsf{Pkt}^2$ such that $\chi \notin [\![\mathsf{subst}(p,\sigma)]\!]$, if $\mathsf{Sat}\ \vec{\delta''} = \mathsf{model}'(p,\sigma,\vec{\delta},\varphi)$, then $\vec{\delta} \circ \vec{\delta''}$ is minimal.*

Proof. Let p, σ, $\vec{\delta}$, φ, χ be given. Assume $\mathsf{Sat}\ \vec{\delta}'' = \mathsf{model}'(p,\sigma,\vec{\delta},\varphi)$.

Consider another sequence of edits $\vec{\delta}'$ such that there is some σ' such that $\vec{\delta}''(\vec{\delta}(\sigma)) = \sigma' = \vec{\delta}'(\sigma)$. Show that $|\vec{\delta} \circ \vec{\delta}''| \leq |\vec{\delta}'|$.

We prove two propositions.

(ADD) Assume that there were some insertion $\delta_i = \mathsf{A}(t,\rho) \in \vec{\delta} \circ \vec{\delta}''$ that doesn't occur in $\vec{\delta}'$. If $\delta_i \in \vec{\delta}$, then $\rho \in \sigma'(t)$. By minimality of $\vec{\delta}$, and $\rho \notin \sigma'(t)$. So $\sigma' \neq \vec{\delta}(\sigma)$, which is a contradiction. If $\delta_i \in \vec{\delta}''$, then $\rho \in \sigma'(t)$, and $\rho \notin \sigma(t)$, because model' always produces reachable edits (Lemma 7). Consequently $\vec{\delta}'(\sigma) \neq \sigma'$ which is a contradiction.

So we know that δ_i has a corresponding edit $\delta'_j \in \vec{\delta}'$. Assume that there is another edit $\delta_k \in \vec{\delta} \circ \vec{\delta}$ that corresponds to δ'_j. This is impossible by Lemma 7 and by the minimality of $\vec{\delta}$.

(DEL) Assume that there were some edit $\delta_j = \mathsf{D}(t,i) \in \vec{\delta} \circ \vec{\delta}''$ deletes some row ρ that occurs in $\vec{\delta}'(\sigma)$: ie. $((\delta_{i-1} \circ \cdots \circ \delta_1)(\sigma))(t)[i] = \rho \notin \sigma'(t)$ and $\rho \in \vec{\delta}'(\sigma)$.

If $\delta_i \in \vec{\delta}''$, we know, by construction, that δ_i deletes a row in $\sigma(t)$. Further, Lemma 8 says $\vec{\delta}'(\sigma) \neq \sigma'$, which is a contradiction.

So we know that δ_i has a corresponding edit $\delta'_j \in \vec{\delta}'$. Assume that there is another edit $\delta_k \in \vec{\delta} \circ \vec{\delta}$ that corresponds to δ'_j. This is impossible because rows can only be deleted once.

Since every edit in $\vec{\delta} \circ \vec{\delta}''$ has a corresponding unique edit in δ'. Conclude that $\vec{\delta} \circ \vec{\delta}'' \subseteq \vec{\delta}'$. The result follows. ☐

Lemma 10 (Completeness)**.** *For every logical program l, every physical program p, every logical instantiation τ, every physical instantiation σ and every sequence of physical edits $\vec{\delta}$, then, the following properties hold:*

1. *if there exists a sequence of physical edits $\vec{\delta}'$ such that*
$$\mathsf{subst}(l,\tau) = \mathsf{subst}(p,\vec{\delta}'(\vec{\delta}(\sigma)))$$
then
$$\mathsf{Ok}\ \vec{\delta}'' = \mathsf{verify}(l,p,\tau,\sigma,\vec{\delta})$$
and
$$\mathsf{subst}(l,\tau) = \mathsf{subst}(p,\vec{\delta}''(\sigma))$$

2. *For every $\chi \in [\![\mathsf{subst}(l,\tau)]\!] \setminus [\![\mathsf{subst}(p,\vec{\delta}(\sigma))]\!]$, and every $\varphi \in \mathsf{Bool}$ such that if $\vec{\delta}' \models \neg\varphi$ and $[\![\mathsf{subst}(l,\tau)]\!]\ \pi_1(\chi) = \pi_2(\chi) = [\![\mathsf{subst}(p,\vec{\delta}'(\vec{\delta}(\sigma)))]\!]\ \pi_1(\chi)$, there is no extension $\vec{\delta}''$ such that $\mathsf{subst}(l,\tau) = \mathsf{subst}(l,\vec{\delta}''(\vec{\delta}'(\vec{\delta}(\sigma))))$ then if there exists a non-empty, minimal sequence of physical edits $\vec{\varepsilon}$ not precluded by φ such that*
$$\mathsf{subst}(l,\tau) = \mathsf{subst}(p,\vec{\varepsilon}(\vec{\delta}(\sigma)))$$
then
$$\mathsf{Ok}\ \vec{\delta}'' = \mathsf{solve}(l,p,\tau,\sigma,\vec{\delta},\chi,X,\varphi)$$
and
$$\mathsf{subst}(l,\tau) = \mathsf{subst}(p,\vec{\delta}''(\sigma))$$

Proof. First we justify the finiteness of our inner inductive measure. There is a finite number of models to every model' query, simply because there are finitely many holes, each of which has finite domain. Since φ is composed of the same set of variables, it is also finite.

Let l, p, τ and σ be given. Proceed by induction on the number of nonempty minimal extensions of $\vec{\delta}$. Lemma 4 shows this measure is well-formed.

BASE CASE There are no nonempty minimal extensions of $\vec{\delta}$. Consider each proposition separately

1. Let $\vec{\varepsilon}$ be a nonempty sequence of edits such that $\mathsf{subst}(l,\tau) = \mathsf{subst}(l,\vec{\varepsilon}(\sigma))$. However, $\vec{\delta}$ has no nonempty minimal extensions, so $\mathsf{subst}(l,\tau) = \mathsf{subst}(l,\vec{\delta}(\sigma)))$. Entering the verify function, observe that by Theorem 1 $\mathsf{CheckSat}(\mathsf{subst}(l,\tau) \not\equiv \mathsf{subst}(l,\vec{\delta}(\sigma)))$ will be UNSAT, and we're done by Theorem 4.
2. Vacuous, there is no such $\vec{\varepsilon}$.

INDUCTIVE STEP There are nonempty minimal extensions of $\vec{\delta}$.

First we prove proposition 2 by strong induction on the number of models for φ.

BASE CASE There are no models for φ, i.e., φ is unsatisfiable. Consequently, the call to model' fails to produce a model. This is a contradiction by Lemma 5.

INDUCTIVE STEP There are n models for φ Let X and χ be given. Assume $\vec{\varepsilon}$ exists such that $\mathsf{subst}(l,\tau) = \mathsf{subst}(p,\vec{\varepsilon}(\vec{\delta}(\sigma)))$.

Now, we get a model by calling $\mathsf{model}'(p,\sigma,\vec{\delta},\chi,\varphi)$, and there are two cases.

i. Assume the result is UNSAT. This is a contradiction by Lemma 5.
ii. Assume the result is $\mathsf{Sat}\ \vec{\delta}''$. By Lemma 6, $|\vec{\delta}''|$ is nonempty. Consider two cases:
Case 1. Assume there exists some extension of $\vec{\delta} \circ \vec{\delta}''$ that is a solution. Then the outer IH proves that the verify call produces a solution.
Case 2. Assume there is no extension of $\vec{\delta} \circ \vec{\delta}''$ that is a solution. Then $\mathsf{verify}(l,p,\tau,\sigma,\vec{\delta} \circ \vec{\delta}'')$

returns Fail. Then, since $\vec{\delta}'' \models \varphi$, and $\vec{\delta} \not\models \neg\vec{\delta}$, the number of models for $\varphi \wedge \neg\vec{\delta}''$ is strictly less than n. Further, we know that there is an way to extend $\vec{\delta}$ that is a solution, namely $\vec{\varepsilon}$. We also know that $\vec{\varepsilon}$ isn't precluded by φ, by assumption, finally we also know that $\vec{\varepsilon}$ isn't precluded by $\neg\vec{\delta}''$, because of our assumption that $\vec{\delta} \circ \vec{\delta}''$ cannot be extended to a solution. These final conditions witness the preconditions of the inner IH, which proves the result.

✓

Now prove proposition 1. Let $\vec{\varepsilon}$ be such that

$$\mathsf{subst}(l, \tau) = \mathsf{subst}(l, \vec{\varepsilon}(\vec{\delta}(\sigma)))$$

There are two cases, either $\mathsf{subst}(l, \tau) = \mathsf{subst}(l, \vec{\delta}\sigma)$ or not. In the former case, we are done by Theorem 1. In the latter, we will get a counterexample χ (by Theorem 1), such that

$$\chi \notin \mathsf{subst}(l, \tau) \cap \mathsf{subst}(l, \vec{\delta}(\sigma))$$

Then the result follows as a special case of the preceeding proof of proposition 2. □

Don't Yank My Chain: Auditable NF Service Chaining

Guyue Liu, Hugo Sadok, Anne Kohlbrenner,[*] *Bryan Parno, Vyas Sekar, Justine Sherry*
Carnegie Mellon University [*]*Princeton University*

Abstract

Auditing is a crucial component of network security practices in organizations with sensitive information, such as banks and hospitals. Unfortunately, network function virtualization (NFV) is viewed as incompatible with auditing practices which verify that security functions operate correctly. In this paper, we bring the benefits of NFV to security-sensitive environments with the design and implementation of AuditBox.

AuditBox not only makes NFV compatible with auditing, but also provides stronger guarantees than traditional auditing procedures. In traditional auditing, administrators test the system for correctness on a schedule, e.g., once per month. In contrast, AuditBox continuously self-monitors for correct behavior, proving runtime guarantees that the system remains in compliance with policy goals. Furthermore, AuditBox remains compatible with traditional auditing practices by providing sampled logs which still allow auditors to inspect system behavior manually. AuditBox achieves its goals by combining trusted execution environments with a lightweight verified routing protocol (VRP). Despite the complexity of routing policies for service-function chains relative to traditional routing, AuditBox's protocol introduces 72-80% fewer bytes of overhead per packet (in a 5-hop service chain) and provides 61-67% higher goodput than prior work on VRPs designed for the Internet.

1 Introduction

Modern networks contain a myriad of *network functions* (NFs) such as firewalls, intrusion detection systems, normalizers, exfiltration detectors, and proxies. Beyond security and performance benefits, a key driving factor for NFs is that they are mandated by legal and policy requirements; *e.g.,* HIPAA [8], FERPA [7], and PCI [13], among others.

While *network functions virtualization* (NFV) promises potential benefits in cost, elasticity, and richer policies [32, 58, 63], there is significant resistance to adoption of NFV [6] for such regulatory use cases. Conversations with industry experts suggest that this reluctance stems from the inability to audit NFV deployments to demonstrate *compliance* as mandated by standards [15, 41]; *i.e.,* show that the service function chains (SFC) are correctly implemented and packets traverse the intended sequences of NFs in the right order.

With legacy hardware NF deployments, administrators and auditors can simply look at static wiring and hardware placement to intuitively verify ('what you see is what you get') if the network meets intended requirements.[1] In contrast, NFV introduces new dimensions of dynamism, virtualization, and multiplexing in the environment. For instance, VMs running NFs may be ephemeral, virtual switches may multiplex several services, and servers may host multiple services. Enhanced dynamism and a larger attack surface make NFV systems harder to reason about and as such, regulators do not have suitable tools for auditing. This lack of auditing is a fundamental stumbling block for NFV adoption.

To this end, we propose (1) *formal models* of correct SFC routing which clarify 'correctness' in the context of dynamic NF scheduling and routing; and (2) a protocol which *provably* provides *continuous* assurance that packets follow the (formally specified) policy-mandated paths.

Realizing this vision in a practical system, however, is challenging. To see why, consider traditional *verified routing protocols* [46, 55] (VRPs). At a high level, VPRs cryptographically ensure that packets are not modified in flight and do not deviate from a traversal of a prespecified sequence of routers. Unfortunately, VRPs fail to provide the required capabilities for our setting. First, VRPs assume that packets will traverse their path unmodified, but NFs can legitimately modify packets. Second, VRPs assume that the correct route for packets is fixed and known a priori, but in SFC the correct route for a packet may only be revealed mid-flight. Third, these approaches focus on per-packet behavior, whereas NFV often involves stateful NFs whose semantics depend on cross-packet state. Furthermore, VRPs have prohibitively high performance overhead; *e.g.,* OPT [46] increases min-sized packets by 3× for a 4-NF chain, and even the most recent work EPIC [47] incurs 1.69× overhead for a strong attack model.

In this paper, we present the design and implementation of AuditBox, which (1) re-enables status-quo 'what you see is what you get' auditing practices, and (2) raises the bar by enforcing *at runtime* that the system operates correctly. AuditBox builds on four key ideas:

- *Using secure enclaves:* To ensure that the correct NF software is running, AuditBox runs them atop hardware enclaves (*e.g.,* Intel's SGX [25]). By changing the trust model, we reformulate verified routing to audit actions between *trusted* NFs, with an untrusted network in between.
- *Trusted PacketIDs:* To tackle dynamic packet modifications, *immutable packetIDs* are carried by an AuditBox packet trailer. This enables us to logically bind modified packets to incoming packets when creating audit trails.
- *NF-hop-by-hop protocols:* Given trusted NFs, we devise a simplified path attestation protocol that focuses on the packets at individual "NF hops". This hop-by-hop attestation has the dual benefit of addressing dynamic paths and reducing the size of attestation headers.

[1]As we argue later, this is indeed a weak guarantee, but today's NFV deployments lack tools even for this weak property.

- *Lightweight simplified operations:* The use of trusted NFs inside enclaves enables a number of simple-yet-effective cryptographic optimizations such as the use of symmetric keys and *updatable MAC* computations to significantly improve data plane performance.

To realize these ideas with minimal modifications to NF implementations, AuditBox embeds a trusted SFC routing shim in the enclave alongside the NF. The shim receives inbound/outbound packets to/from the NF and is simultaneously responsible for generating *audit trails* for traditional auditing practices and for our new goal of enforcing *runtime checks* that the untrusted components of the system behave as expected. Relative to verified routing protocols for the Internet, AuditBox introduces 72-80% less per-packet header overhead (assuming a 5-hop service chain) and offers 62-67% higher goodput in the dataplane.

Nonetheless, auditability in AuditBox (or any such framework) does come at a cost relative to an uninstrumented NFV cluster (*e.g.,* 3%-38% overhead for s single NF as shown in Figure 16). That said, for security-sensitive settings, regulatory compliance is a fundamental requirement for NFV deployment. Hence, AuditBox's essential advantage is in bringing the benefits of ease of management, lower-cost equipment, faster upgrades and security patches, and flexibility that are associated with NFV to new markets where it would not have been viable previously. Indeed, we estimate that AuditBox, despite its overheads relative to uninstrumented NFV, can still result in capital savings of 1.9-60× (depending on the appliance) relative to traditional hardware middlebox solutions.

2 Background and Motivation

We begin by discussing network compliance today (§2.1), our problem statement (§2.2) and threat model (§2.3), and finally discuss why compliance for NFV is challenging (§2.4).

2.1 Tussle Between Compliance and NFV

Modern organizations need to satisfy a number of security standards for compliance with government and industrial regulations (*e.g.,* HIPAA, FERPA, FISMA, GDPR, and PCI, among others) [66]. In this paper, we focus on requirements or *controls* related to network security under NIST 800-53 [41] in the United States:[2]

- *Middleboxes must be deployed to protect sensitive data and systems:* Control SC-7 mandates the need for protection devices (*e.g.,* proxies, gateways, firewalls, guards, encrypted tunnels) arranged in an effective architecture.
- *Administrators must periodically test that security infrastructure is running properly:* Control SI-6 demands that these inspections be performed periodically, with 'once a month' as an example acceptable frequency.
- *Systems must provide logs of anomalies and past behavior:* Control AU-2 requires systems to keep such records for later analysis for auditing.

[2]ISO 270001 [15] has similar international standards.

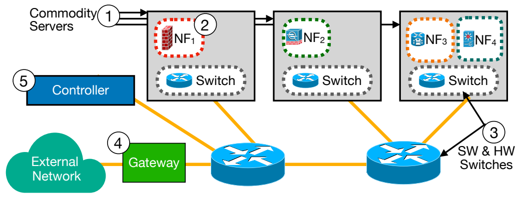


Figure 1: Components of a basic NFV cluster.

- *Independent 'auditors' must certify that security mechanisms are in place and running correctly:* Control CA-7 mandates that organizations be 'certified' by outside auditors (*e.g.,* third-party IT consulting companies) that the above requirements (among others) are being met.

Reluctance to adopt NFV: To meet the above compliance requirements, there is a large regulatory technology ('RegTech') industry [66] (expected to surpass \$55.28B by 2025). One might expect then that this RegTech market would be an early adopter of NFV to reduce capital and operating expenses. However, our conversations with representatives from NIST and a RegTech firm revealed that this industry is *hesitant* to adopt NFV. The key reason is that while NFV lowers the bar for some aspects of compliance (*e.g.,* SC-7), it makes other requirements such as SI-6, AU-2, and CA-7 difficult, if not impossible.

In hindsight, this reluctance is not surprising. NFV introduces new dimensions of dynamism and multiplexing (*e.g.,* shared hosts running VMs, dynamic overlay routing, dynamic load balancing). This makes it harder to reason about the deployment and introduces an increased attack surface for threats (and misconfigurations). In contrast, a simple statically-wired NF deployment with hardware boxes seems intuitively easy to test, audit, and demonstrate compliance to external auditors.

2.2 Problem Setup

We consider an NFV cluster managed by a framework such as E2 [58], AT&T Domain 2.0 [2], or Blue Planet [3]. At a high level, NFV clusters consist of five basic components (Figure 1): (1) *Commodity servers* on which containerized or virtualized NFs run; (2) *Network Functions* such as firewall, proxy, or IDS; (3) *Software and Hardware Switches* that steer traffic between a sequence of NFs; (4) *Gateways* where cluster traffic enters; and (5) a *Controller* responsible for provisioning NFs on the servers and defining routes through them. NF instances are composed to create *service function chains* (SFC) policies for specific traffic classes (*e.g.,* Gateway → Firewall → Proxy → IDS → Gateway).

To make these systems *auditable*, we need to verify that: (1) the correct and untampered NFs are running correctly and (2) packets traverse these NFs according to policy. We formally define what it means for traffic to traverse NFs 'according to policy' in §4. Furthermore, to meet the compliance requirements, operators must be able to inspect and demonstrate that the correct behavior is happening; *e.g.,* trigger tests to observe that packets follow policies (SI-6) and

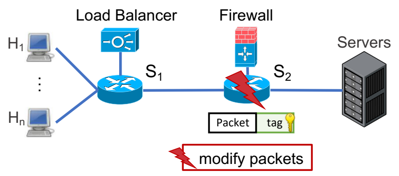


Figure 2: MACs for integrity are invalidated by rewritten packets.

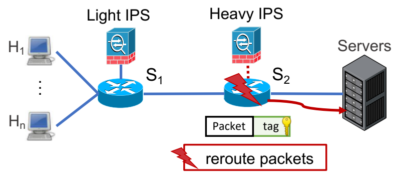


Figure 3: Rerouting traffic and bypassing the heavy IPS.

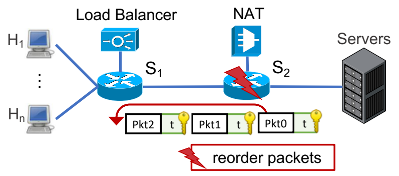


Figure 4: Reordering packets effects NF state and future packets.

produce *audit trails* for external inspectors.

To 'raise the bar' for auditability, we add two additional requirements. First, rather than merely enabling administrators to test on demand or post-facto whether or not the system is or was running correctly, we also want to enable the system to *audit itself, continuously, at runtime* for deviations from correct behavior, and to alert administrators if such a deviation is detected. Second, we also want to support rich *dynamic* SFC policies as opposed to traditional static service-chain policies; *e.g.,* steering packets tagged as suspicious by earlier NFs for deeper inspection [32].

In this context, we note that while auditability may seem related to *network verification* (*e.g.,* [44, 45, 60, 74], the requirements are fundamentally different on two fronts. First, most existing network verification efforts simply look at the configurations of the network elements such as NFs/switches and formally verify if the configuration meets intended policies. It does not typically provide any runtime guarantees about data plane actions. Second, network verification *can* be used to provide evidence of correct operation [45] but it does not provide 'what you see is what you get' audit trails; in this regard, verification could be coupled with audit trails to provide additional evidence of compliance.

2.3 Threat Model

Although most operators are primarily concerned with cluster misconfiguration rather than outright attacks, we target a stronger threat model as follows. First, what is trusted: the controller is the arbiter of correct policy and how NFs should be scheduled; we assume that the controller is trusted, and we do not consider attacks where a 'lead' administrator (that is, an administrator charged with configuring policies at the controller) provides invalid or malicious policies to the controller. In the case of the gateway and the NFs, we assume there exists vendor-certified code which is digitally signed. This code is privileged to drop or rewrite packets, and we exclude NFs that can inject packets.

Most other components of the network are untrusted. An attacker may attempt to corrupt server software (including the operating system and/or VMM), NF and gateway software, and the software and hardware of the switches. The attacker can cause one or more corrupted components to inject, drop, or rewrite packets.

Our solution builds on the 'abstract enclave assumption' defined by prior research [17, 18, 61]: the attacker cannot observe or modify any data or program code running within an enclave, and the enclave is trusted to attest to the integrity of the code running therein. While existing enclave solutions – such as SGX, which we build on – fall short of meeting the abstract enclave assumption perfectly [22, 24, 36, 54, 73], fixing the shortcomings of current enclave solutions is out of scope for this work, as such fixes are an active area of research in their own right [26, 34].

2.4 Challenges

At first glance, it would appear that we can borrow from prior work on *verified routing protocols* (VRPs) that cryptographically guarantee that a packet takes a pre-specified intended path and is not modified in flight (*e.g.,* [46, 49, 55, 77, 80–83]). We use OPT [46] as an exemplar state-of-art solution from this class. Specifically, OPT extends each packet with: (a) a cryptographic hash of the packet contents and (b) its expected switch-level path. Every router along the path verifies that the packet's current hash matches the header and also adds attestations to ensure the packet has matched the expected sequence. As we will see next, our NFV auditability problem introduces new dimensions outside the scope of these prior efforts.

Mutable Packets: Figure 2 shows two NFs: a load balancer that modifies the destination IP to distribute the load across multiple backend servers and a firewall configured to block packets from malicious IPs. Consider the scenario where switch S_2 (either adversarially or via misconfigurations) modifies the IP header to bypass the firewall. Because NFs can legitimately modify packet headers, it is difficult to distinguish whether this action was malicious or an intended NF action. OPT-like VRPs assume that most packet fields are immutable and perform crypto operations by excluding a few mutable packet fields (*e.g.,* TTL), and hence would generate a large number of false positives by flagging all legitimate NF modifications.

Dynamic Paths: Consider a dynamic SFC scenario in Figure 3, where a lightweight IPS performs basic detections and then routes suspicious packets to the heavy IPS for further processing. Again, an adversarial or misconfigured device could reroute all suspicious packets to bypass the heavy IPS, and it is hard to tell whether this was the result of the light IPS's action or a malicious switch. Because the intended path cannot be determined until the light IPS finishes processing, OPT-like VRPs – which must pre-specify the end-to-end route of the packet – are not applicable to this network.

Stateful Behavior: NFs' stateful semantics mean that per-packet auditability may not be sufficient. We may also need to ensure that all packets in a given flow follow the same path and that they arrive in order if we wish to ensure the stateful semantics are not compromised. To see why, consider Figure 4, which shows two stateful NFs: a layer 4 load balancer to distribute packets based on source IP and port and a NAT that maps public ports to private ports. Consider an adversary (or misconfiguration) that reorders packets and sends the `FIN` packet before the data packets. This could cause all following data packets to be discarded by the load balancer. This highlights a fundamental limitation of VRPs: because they reason about correctness of each packet independent of the others, there may be cross-packet policy violations which they cannot capture.

3 AuditBox Overview

Our goal in designing AuditBox is to provide auditing capabilities for NFV deployments. In practice, we also want: R1) *minimal modifications* to existing NFs and R2) *low overhead* on the data/control paths. In this section, we discuss some of the key ideas in AuditBox and its overall architecture.

3.1 Key Ideas

We first discuss the main ideas that enable AuditBox to tackle the challenges of packet modifications (C1), dynamic paths (C2), and stateful actions (C3) while meeting the practical requirements of minimal NF modifications (R1) and low overhead (R2).

A) Running NFs in enclaves atop a shim: Inspired by the trust guarantees provided by prior work [39,61,70], AuditBox runs NFs in trusted enclaves. This enables AuditBox to trust those modifications that are validly introduced by NFs (C1). To avoid modifying existing NFs (R1), we introduce a trusted shim in each enclave to perform auditing (§5.4). This shim also ensures that the next-hop NFs are chosen based on the intended policy (C2).

B) Trusted Packet ID: Mutable packets (C1) make it hard to generate audit trails as we cannot causally relate NF-modified packets to their inputs (§4). To tackle this, we introduce an *immutable packet ID*, carried by the packet header (§5). We envision a *trusted gateway* (running in an enclave) that generates and assigns this ID when the packet first arrives in the NFV cluster; the packet ID is carried through NFs even if the packet itself is modified or rewritten by NFs.

C) NF-Hop-by-hop updated attestations: We leverage the trusted shim in each enclave to develop a hop-by-hop attestation protocol in which each *pair* of shims attest that the packet was delivered, without improper modification between an NF and its policy-compliant successor. Compared to the end-to-end approach taken by traditional VRPs, hop-by-hop attestation has the dual benefit of supporting *dynamic paths* and also *reducing packet overheads*.

D) Efficient crypto mechanisms: By using trusted enclaves, we can use *one symmetric key* for all NFs in the same policy pipelet (§3.3), to simplify the cryptographic operations and also introduce new opportunities for efficiency. For example, we implement an efficient *updatable MAC* algorithm to improve the performance of repeated attestations to the packet at each hop (§6.1).

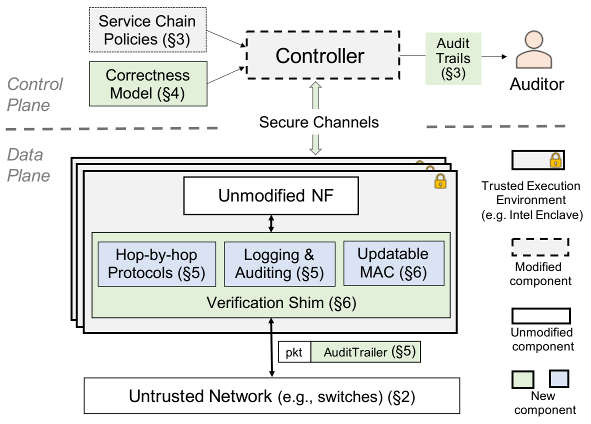


Figure 5: AuditBox Architecture.

3.2 AuditBox Architecture: Data Plane

The key components of the AuditBox architecture are illustrated in Figure 5.

In the data plane, AuditBox runs unmodified NFs in trusted execution environments (TEEs) to isolate them from other untrusted network components (*e.g.,* switches, OSes). Although our current implementation (§6) uses Intel SGX enclaves, our design in principle can be realized using other TEE technologies such as Arm TrustZone [1]. The key capabilities we leverage are attested memory isolation and integrity during program execution. In each enclave, we add a shim which intercepts the traffic entering/exiting the NF to serve three purposes. (1) The shim determines the correct next-hop NF according to a policy it received from the controller; this is no different than any other NFV policy manager such as FlowTags [32] or E2 [58].

The second two tasks for the shim are novel to AuditBox and form the entirety of sections 4, 5, and 6 and so we only introduce them briefly here. (2) The shim checks each incoming packet to verify that the packet has not been improperly routed or modified while traversing the untrusted network between enclaves; on egress the shim attaches a custom trailer (called AuditTrailer) along with a MAC attesting to the contents of the packet and trailer so that the next-hop NF can similarly verify the packet was routed correctly. (3) The shim logs any packets to local storage which *either* appear to violate policy, *or* are tagged via a secret 'log bit' for recording.

3.3 AuditBox Architecture: Control Plane

The controller serves two key purposes: NF deployment and management, and serving as an interface for an administrator to inspect logs and audit trails.

NF Deployment: NF deployment includes scheduling NFs

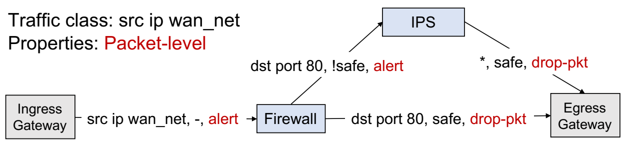


Figure 6: An annotated policy graph.

on servers, verifying enclave attestations that the correct NF software is running at each NF, distributing and updating the global symmetric key used for dataplane verification, and installing the correct *next-hop policies* at each enclave.

Policies are roughly similar to other DAG-based policy languages [31, 32, 58]: network operators specify *policy pipelets* where nodes represent NFs and edges are annotated with traffic classes and where they should be routed according to policy (illustrated in Figure 6). AuditBox augments this traditional policy with two new parameters: a *correctness model* which defines what behavior in the network constitutes a violation (presented in §4) and *violation annotations* on each edge to determine whether packets which violate the correctness model should be dropped or raise an alert. From this policy, the operator can define a function which, for a given packet egressing an NF, can determine what the correct next-hop NF to deliver the packet to is; this function is installed in the shim at each enclave along with the assigned violation action to take in case, *e.g.,* a packet is corrupted inflight between enclaves.

Logging and Audit Trails: Auditors and administrators can use the controller to query for *logs* (records of any anomalies or policy violations) and *audit trails* (the end to end path a packet takes through the cluster, and all intermediate rewritten states of the packet between NFs). By default, each enclave stores logs and audit trail records to local storage, encrypted with a symmetric key that is local to that NF; the controller sets up pairwise keys with each NF for log and audit storage. Because it is infeasible to log every packet through the system, the administrator configures a sampling policy; we discuss the sampling policy, the logging mechanism, and the security of the logs in §5.4.

4 Formalizing Correctness

We begin by formally defining what it means for a system to **(a)** obey correct routing, and **(b)** support auditing.

Correct Routing: Since networked forwarding elements are untrusted, we rely on a trusted shim in each SGX enclave to verify at runtime that the network has not deviated from correct behavior. §4.2 defines 'correctness' with regard to network forwarding behaviors, *i.e.,* those verified by the shim.

Auditability: Our verified routing approach operates on a hop-by-hop basis; as we discuss in §5 this simplifies our protocol and limits its size overhead. However, auditors expect *end-to-end* evidence (upon inspection) that the system is operating correctly. An 'audit trail' consists of a recorded sequence of *causally connected* operations across devices – *e.g.,* packet p entered at the gateway, was processed by NF_i resulting in p', which was processed by NF_j, and resulted in p'', which was released through the gateway. Due to space constraints, we defer our formal definition of an 'audit trail' to §B.2.

Prior to introducing our formal definitions (§4.2), we define our model of the network (§4.1). We make a few assumptions for the sake of simplifying our presentation. All can be relaxed at the cost of additional notational complexity. First, we assume that the cluster has a single ingress/egress 'gateway' where packets transit to/from the primary network. We similarly assume that each NF has only one ingress/egress port. Finally, we assume that forwarding within the cluster is performed at L2, and hence it is not necessary for network switches to update any TTL values or checksums, *i.e.,* there is no need for the network to modify packets.

4.1 Definitions

Time: We model time as an ordered sequence E, the set of all *events* in the system. E is initialized to $[]$, that is, empty.

As the (modeled) system runs, NFs in the system populate E with 4-tuples containing the following named values:

- $\mathbf{pkt}_{\mathrm{in}}$: a packet received ($\in P$, defined below).
- $\mathbf{pkt}_{\mathrm{out}}$: a packet sent ($\in P$).
- **NF**: the network function ($\in F$, defined below).
- **t**: the logical 'time' the packet was received or sent (*i.e.,* the index in E); represented as a positive integer ($\in \mathbb{N}$).

We may refer to members of $e \in E$ as $e.\mathrm{pkt}_{\mathrm{in}}$, $e.\mathrm{pkt}_{\mathrm{out}}$, $e.\mathrm{NF}$, or $e.\mathrm{t}$. We *index* E using array notation *e.g.,* $\mathrm{E}[43] = (*,*,*,43)$. We may also search E for events matching the specified value at a specified field using the function $\text{get-event}_E(\text{field, value}) \rightarrow [\mathrm{E}]$, for example, $\text{get-event}_E(\mathrm{pkt}_{\mathrm{in}}, p_i)$ returns the sequence of all events in E where p_i was received as input at an NF.

Note that, although AuditBox provides some logging mechanisms, *there is no globally ordered event log in the system implementation* – the event sequence defined here is merely an abstraction to help reason about time while modeling.

Packets: P is the set of all 64-9000 byte binary strings, that is, P is the set of all (up to jumbo framed) Ethernet packets. Each packet p contains an Ethernet header and an IP header. Depending upon the NFV framework, the packet may also contain a collection of 'metadata' fields such as FlowTags [32] or a Network Service Header (NSH) [64]. If the packet represents a TCP or UDP packet, we represent the classic flow 5-tuple (source IP address, destination IP address, source port, destination port, protocol) through the function $\textsc{flow}(p)$.

As a packet p traverses the network, it may be transformed into some p' or p'' through modifications to the payload or metadata fields. If the data *anywhere* in the packet – that is, the 64-9000 byte binary string it represents – has been changed, then $p \neq p'$. In the event log, $e.\mathrm{pkt}_{\mathrm{in}}, e.\mathrm{pkt}_{\mathrm{out}} \in P$.

NFs: There is a set of Network Functions F. Each $\mathrm{NF}_i \in F$ is a function,[3] $\mathrm{NF}_i : P \rightarrow P \cup \{\perp\}$. That is, it takes in a packet and produces another packet (or null).

[3]Called a 'transfer function' elsewhere in the literature [44, 60].

In our model, we assume NF_i encloses some NF_i-specific function $f_i : P \to P \cup \{\perp\}$, which may keep state, modify packet contents, etc. When NF_i takes in a packet and f_i returns null, this represents a drop. We exclude NFs that could inject new packets.

With respect to the event log E, we log each operation at NF_i as follows:

Algorithm 1 NF Model

```
1: function NF_i(input)
2:     output ← $f_i$(input)
3:     if output $\neq \perp \vee$ input $\neq \perp$ then
4:         $E$.append(input, output, NF_i, $E$.length + 1)
5:     return output
```

Switches: Like prior work in SDN and formal modeling [44], we consider the network as 'one big switch' or a 'fabric' and we model its behavior with a single function $\Phi : P \cup \{\perp\} \to P \cup \{\perp\}$. In our threat model, Φ is the untrusted part of the system. Φ is not considered an NF ($\Phi \notin F$), and Φ does not append to E.

Gateway: Packets enter and exit the cluster – and hence enter and exit the model – via a dedicated NF, GW which, like other NFs, also appends to the event sequence. We model the Gateway as GW_{in} when a packet enters the cluster via the gateway as follows:

Algorithm 2 Model of Packets Entering the Cluster

```
1: function $GW_{\text{IN}}$(input)
2:     if input $\neq \perp$ then
3:         $E$.append($\perp$, input, $\mathrm{GW}_{\text{in}}$, $E$.length + 1)
4:     return $f_{in}(input)$
```

Like normal NFs, the gateway applies a gateway-specific function f_{in} to the packet before transmitting it. We define GW_{out}, for when a packet exits the cluster similarly (§B.1).

Path: Packet processing occurs via traversal of a sequence of NFs and switches. Whenever an NF sends a packet, it is passed to the network which is expected to steer the packet to the next NF specified by the service-chain policy. When a new packet arrives at GW_{in}, we model traversal of the network via a nested function of Φ and elements of F, for example: $\mathrm{GW}_{\text{out}} \circ \Phi \circ \mathrm{NF}_i \circ \Phi \circ \mathrm{NF}_j \circ \Phi \circ \ldots \circ \Phi \circ \mathrm{GW}_{\text{in}}$.

Policy: There are many languages [32, 58] and services for specifying service-chain policy. Here we assume that for a packet and an NF which just produced that packet, that there exists some *policy function* ($\text{policy} : P \times F \to F$) that can determine the NF that should next process the packet. Importantly, we assume that the information necessary to determine the right next hop relies only in the packet fields (*e.g.,* IP header, metadata) and the departing NF.

4.2 Correctness

We give two possible definitions of "correct" forwarding below, based on properties of the modeled event log E.

4.2.1 Packet Correctness

Our first definition, packet correctness, is based on the UDP service model of packet delivery. Under the packet-correctness definition, the system is 'correct' even if packets are reordered, dropped, or duplicated between NFs. The behaviors which are incorrect under this model are limited to **(a)** injecting packets which were not sent by an end host, or **(b)** modifying/corrupting packets between sender and receiver.

Packet correctness holds iff Property 1 holds over E.

$$\forall e_b \in E \text{ s.t.}: e_b.\text{pkt}_{\text{in}} \neq \perp, \tag{1}$$
$$\exists e_a \in E \text{ s.t.}: e_a.\text{t} < e_b.\text{t} \wedge e_a.\text{pkt}_{\text{out}} = e_b.\text{pkt}_{\text{in}} \tag{2}$$
$$\wedge \text{policy}(e_a.\text{pkt}_{\text{out}}, e_a.\text{NF}) = e_b.\text{NF} \tag{3}$$

Property 1: No Injection or Modification

To summarize the above: for all events e_b in which an NF e_b.NF receives and processes a packet, there exists a prior event e_a in which e_a.NF sends the same packet to e_b.NF.

4.2.2 Flow Correctness

Flow correctness is based on the TCP service model. Like packet correctness, a network where packets are injected or corrupted by the network is not correct. To meet flow correctness, the network also must not drop packets, reorder packets within a flow, or duplicate them. Hence, the network obeys Flow Correctness iff it respects Property 1 and the following three properties.

The first additional property aims to verify that the network has not dropped any packets in flight. An absolute guarantee that no packets are dropped is impossible, as the NFs cannot force hostile network elements to deliver packets. Hence, we instead define our 'no drops' property to state that packets in a given flow that *do arrive* at their own destination may only be accepted if all packets previously transmitted from that flow to that destination have already arrived. To achieve this, Property 2 ('No Drops') says in English that if a packet $e_{a2}.\text{pkt}_{\text{in}}$ is sent by e_{a2}.NF and received at e_{b2}.NF, then any other earlier packet $e_{a1}.\text{pkt}_{\text{out}}$ belonging to the same flow and also destined for e_{b2}.NF should have already been received by e_{b2}.NF prior to $e_{b2}.\text{pkt}_{\text{in}}$.

$$\forall \mathrm{NF}_a, \mathrm{NF}_b \in F, \forall e_{a1}, e_{a2}, e_{b2} \in E \text{ s.t.}: \tag{4}$$
$$(e_{a1}.\text{t} < e_{a2}.\text{t} \wedge e_{a1}.\text{pkt}_{\text{out}} \neq \perp \wedge e_{a2}.\text{pkt}_{\text{out}} \neq \perp \tag{5}$$
$$\wedge e_{a2}.\text{pkt}_{\text{out}} = e_{b2}.\text{pkt}_{\text{in}} \tag{6}$$
$$\wedge \text{policy}(e_{a1}.\text{pkt}, e_{a1}.\text{NF}) = \text{policy}(e_{a2}.\text{pkt}, e_{a2}.\text{NF}) \tag{7}$$
$$\wedge e_{a1}.\text{NF} = e_{a2}.\text{NF} = \mathrm{NF}_a \wedge e_{b2}.\text{NF} = \mathrm{NF}_b \tag{8}$$
$$\wedge \text{flow}(e_{a1}) = \text{flow}(e_{a2})) \implies \tag{9}$$
$$(\exists e_{b1} \in E \text{ s.t.}: \tag{10}$$
$$(e_{b1}.\text{t} < e_{b2}.\text{t} \wedge e_{b1}.\text{NF} = \mathrm{NF}_b \wedge e_{b1}.\text{pkt}_{\text{in}} = e_{a1}.\text{pkt}_{\text{out}})) \tag{11}$$

Property 2: No Drops

Property 3 ensures that packets within a flow are not reordered between NFs. In English, the property specifies that if an NF_a transmits a packet $e_{a1}.\text{pkt}_{\text{out}}$ before a packet

e_{a2}.pkt$_{\text{out}}$ to the same NF$_b$, then e_{a1}.pkt$_{\text{out}}$ should also *be received* by NF$_b$ before e_{a2}.pkt$_{\text{out}}$.

$$\forall \text{NF}_a, \text{NF}_b \in F, \tag{12}$$
$$\forall e_{a1}, e_{b1}, e_{a2}, e_{b2} \in E \text{ s.t. :} \tag{13}$$
$$(e_{a1}.\text{pkt}_{\text{out}} = e_{b1}.\text{pkt}_{\text{in}} \wedge e_{a2}.\text{pkt}_{\text{out}} = e_{b2}.\text{pkt}_{\text{in}} \tag{14}$$
$$\wedge e_{a1}.\text{pkt}_{\text{out}} \neq \perp \wedge e_{a2}.\text{pkt}_{\text{out}} \neq \perp \tag{15}$$
$$\wedge e_{a1}.\text{NF} = e_{a2}.\text{NF} = \text{NF}_a \wedge e_{b1}.\text{NF} = e_{b2}.\text{NF} = \text{NF}_b \tag{16}$$
$$\wedge \text{flow}(e_{a1}) = \text{flow}(e_{a2})) \implies (e_{a1}.\text{t} < e_{a2}.\text{t} \iff e_{b1}.\text{t} < e_{b2}.\text{t}) \tag{17}$$

Property 3: No Reordering Within the Same Flow

$$\forall e_a \in E, \nexists e_b \in E \setminus \{e_a\} \text{ s.t. : } e_a.\text{pkt}_{\text{in}} = e_b.\text{pkt}_{\text{in}} \tag{18}$$

Property 4: No Duplication

Finally, we ensure that packets are not duplicated in the networks ('replay attacks') with Property 4. We note that the Property 4 formalization assumes that the same packet is never sent by any NF more than once. This seems to be a reasonable assumption. As we are forwarding under L2, an NF$_A$ and an NF$_B$ will always have different Ethernet headers even if the IP, TCP/UDP, and payload are identical. And it seems reasonable to assume that an NF$_A$ will never transmit the same packet twice either – even a re-transmitted TCP packet will come with a different IPID value. However, we discuss how to revise this formalism to allow NFs to duplicate packets in §A.

5 AuditBox Protocol

In this section, we focus on the dataplane protocol. We begin by providing a high-level view of the actions that each AuditBox-enabled node performs at each hop (§5.1). Then we discuss the detailed construction of the attestation check and update logic we use for packet- (§5.2) and flow-level (§5.3) correctness. We envision different NFV deployments can flexibly choose one of these levels of correctness as desired. Finally, in §5.4 we describe the secure logging mechanisms which allow auditors to observe audit trails demonstrating that the system is operating correctly.

5.1 High-level workflow

AuditTrailer: AuditBox adds an AuditTrailer, carried by the packet, to verify integrity. We discuss the contents of AuditTrailer in packet-level and flow-level form in detail in §5.2 and §5.3 respectively; here we briefly present the two foundational fields for auditing and runtime correctness. These fields are present in both the packet-level and flow-level versions of the AuditTrailer.

pktID is a unique, immutable ID assigned to a packet; when a packet enters an NF and is modified or rewritten, the *pktID* allows us to identify for each input packet which output packet (if any) is a result of its processing. This tracing is the key mechanism which enables us to generate audit trails – the causal sequence of events we present in §B.2.

tag is a message authentication code (MAC) which allows us to identify if a packet has been improperly modified while in flight between two (trusted) shims. Operationally, the tag is a Galois Message Authentication Code (GMAC), computed using a symmetric key over various packet fields (discussed below in §5.2 and §5.3). The *tag* field enables us to perform runtime correctness checks while running with untrusted operating systems, switches, *etc.*.

Data Plane Actions (Shim and Gateway): The AuditBox dataplane protocol is implemented at the gateway to/from the cluster and at the shims inserted into each NF enclave.

At the Gateway: For each incoming packet, the ingress gateway generates an AuditTrailer (including a unique pktID field) and appends it to the packet, and then forwards it to the next NF. At the end of the service chain, as packets leave the cluster, the egress gateway validates the AuditTrailer (assuming the validation passes), and then removes the AuditTrailer and forwards the packet out of the cluster.

At the Shim: At each hop, the shim receives the incoming packet and performs the following four operations: Check, Process, Update, and Log. *Check* validates the received packet, including verifying the MAC stored in the *tag* field, to verify that the received packet was not incorrectly modified by the network. *Process* hands the packet to the actual NF code; when the packet is returned from the NF, the shim then *Updates* the AuditTrailer (*e.g.,* computing the new *attestation* field). Finally, in certain cases the shim *Logs* the packet and certain metadata to produce an audit trail; the packet is then released to its next hop in the service chain.

5.2 Packet Correctness Protocol

As presented in §4.2.1, the goal of our packet correctness protocol is to ensure that packets are not injected or modified by the network, *i.e.,* that all packets received and processed by an NF were transmitted *to* that NF by another NF or the ingress gateway.

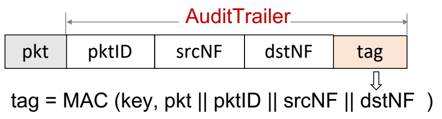


Figure 7: The AuditTrailer for packet-level integrity.

For packet correctness, the AuditTrailer (Figure 7) contains the following fields: pktID (6 bytes), tag (16 bytes), srcNF (2 bytes), dstNF (2 bytes). As discussed above, the pktID is required for generating audit trails so we do not discuss it further here other than to require that it be transmitted alongside other packet fields (IP header, payloads, *etc.*) uncorrupted by the network. Naïvely, one might simply compute the MAC over the packet and pktID to compute the tag. However, this would only meet a portion of the clauses from Property 1 – that there exists *some* valid NF which transmitted this packet. If a malicious or misconfigured switch delivers a valid packet, intended for some NF A, to NF B, the packet would appear to have a valid tag.

Adding the srcNF and dstNF fields explicitly to the header bind the packet to the *policy-compliant route*, meeting the final clause of Property 1. The sender explicitly encodes its own NF ID and the intended destination NF ID into the packet and computes the MAC over the packet, pktID, srcNF, and dstNF fields. The receiver, by validating the MAC stored in the tag, can be sure that the sending NF intended to send the packet to the receiver; the receiver can also (redundantly, as a sanity-check) re-compute the policy compliant next-hop NF for the received packet and the source NF to further verify that it is indeed the correct recipient.

We specify this protocol in Algorithm 5 and prove that it meets the requirements of Property 1 in Appendix D.

Theorem 1 (Packet Correctness) *Consider the game described in §D.3 with adversary $\mathcal{A}$ and instantiated with the AuditBox packet-correctness protocol. Specifically, looking at Algorithm 5, we instantiate f_{in} with the function* GENERATE *and f_i with* PROCESS$_i$*. The probability that the game outputs an event log E that violates Property 1 is negligible.*

5.3 Flow Correctness Protocol

We now discuss the implementation of the Flow Correctness Protocol, which in addition to Property 1 also meets Properties 2, 3, 4. We envision that networks that use only stateless or order-insensitive NFs will prefer the lighter packet-correct protocol, but those with stateful NFs whose operations are sensitive to packet ordering may use the stronger flow-correctness protocol.

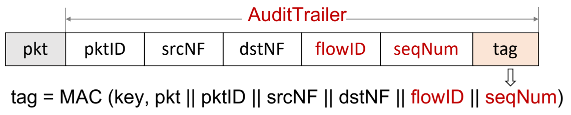

AuditTrailer
pkt
pktID
srcNF
dstNF
flowID
seqNum
tag
tag = MAC (key, pkt || pktID || srcNF || dstNF || flowID || seqNum)

Figure 8: The AuditTrailer for flow-level integrity.

As shown in Figure 8, to extend the AuditTrailer for flow-level semantics, we add two additional fields over the packet-level version: a flowID field (4 bytes), and a seqNum field (4 bytes). The tag is now computed over the packet and all fields including flowID and seqNum.

When packets enter the cluster through the gateway, the gateway hashes the classic '5-tuple' and looks this up in a flow table mapping 5-tuples to flowIDs. If a flowID already exists for this flow, the gateway appends the existing flowID to the AuditTrailer. If a flowID does not exist for this flow, the gateway assigns an unallocated ID number to the flow and inserts this into the flow table and the packet. For non-TCP and non-UDP packets, the flowID is simply set to 0. Like the pktID, the flowID is unmodified as packets flow through the network – even if header values and port numbers are rewritten, the flowID remains the same across NFs; as we will show, this is important for the creation of audit trails (§5.4).

seqNum values are maintained per-hop and represent the ordering of packets between a sending NF$_A$ and a receiving NF$_B$ for a given flow. The sending NF maintains an incrementing per-flow counter, and, for each packet from the same flow, appends the sequence number to the seqNum field in the AuditTrailer. At the receiver, there is a corresponding table of per-flow counters with the next sequence number expected. In our implementation, there is no reason for in-network reordering within a flow and hence the receiver raises an alert for any out-of-order packets; we could configure the receiver to maintain a small buffer to wait for reordered packets and put them back in order for processing in a cluster (and to only raise an alert after multiple out of order arrivals) where reordering were for some reason possible. Thus, packets are discarded at the receiver if they *either* fail the tag verification *or* fail the expected-next-sequence number test.

Because the protocol rejects out-of-order packets, it easily meets Property 3. Since duplicate packets will have the same sequence number, they are detected. Thus, the protocol also meets Property 4. Finally, if a packet is dropped, the arrival of subsequent packets will induce alerts due to out-of-order sequence numbers, meeting Property 2.[4] We specify this protocol in Algorithm 6 and formally prove its security in Appendix D.

Theorem 2 (Flow Correctness) *Consider the game described in §D.2 with adversary $\mathcal{A}$ and instantiated with the AuditBox flow-correctness protocol. Specifically, looking at Algorithm 6, we instantiate f_{in} with the function* GENERATE *and f_i with* PROCESS$_i$*. The probability that the game outputs an event log E that violates Properties 1-4 is negligible.*

Why not use existing TCP sequence numbers? Rather than embedding an additional sequence number, one might attempt to naïvely re-use the sequence number already embedded in TCP flows to save additional bytes in the header. However, packets may *already be missing* when they enter the cluster (leading to sequence number gaps which are not the result of misbehavior within the cluster) and NFs may choose to drop packets (also creating legitimate sequence number gaps). Hence we need a new sequence number whose role is only to detect gaps that are the result of drops *between NFs* in our cluster.

Why not one sequence number per pair of NFs, rather than per-flow? At first glance, it may seem simpler to keep a sequence number across all flows rather than a sequence number per-flow. However, many NF implementations use receive side scaling (RSS) at the receive NIC to fan out packets across multiple cores; in such an architecture a unified sequence number becomes a performance bottleneck; per-flow counters are more parallelizable and hence more scalable.

5.4 Logging & Auditing

Traditional logging (as mandated by AU-2 in §2) focuses on recording packets which resulted in alerts, policy violations,

[4]However, an attacker who blocks *all* packets from a flow may go undetected – to avoid this noncompliance, a receiver detecting a flow with over a minute without transmissions will query the sender for its sequence number to detect any dropping behavior.

and anomalies. At each shim, any packet which results in a violation is logged in its entirety, including all fields of the AuditTrailer; the log is written to local storage and encrypted with an NF-local symmetric logging key provided by the controller.

AuditBox also records additional logged events to produce an 'audit trail', which restores 'what you see is what you get' confidence in the correctness of the underlying system. As defined in §B.2, an audit trail consists of a hop-by-hop trace of a packet (or all packets in a flow) through a sequence of NFs as well as their intermediary rewritten states.

The basic idea is as follows. The gateway probabilistically samples incoming packets and tags them with a 'log bit' which follows the packet through the cluster along with the AuditTrailer. Any packet whose log bit is set to true is logged at each shim, including all metadata or AuditTrailer fields. The administrator or inspector specifies three parameters to define which packets are sampled: (1) a Berkeley Packet Filter (BPF) to match for selected packets, (2) a sampling rate (*e.g.,* 0.001%), and (3) whether to log at the packet- or flow- granularity. Logging at the flow granularity is only permitted if the policy is in flow-correctness mode.

Upon querying, the administrator can then inspect manually (or via an automated script) the path of packets or flows throughout the entire system. Even if packet headers or fields are changed, the packetID or flowID are preserved between NFs; even in traditional service chain deployments such tracing is not possible today when NFs modify packets in a way that they cannot be connected to their corresponding inputs.

There is only one 'trick' to this very simple design: in a truly malicious environment, the network could selectively treat logged packets according to correct policy and only manipulate unlogged packets. Although we would still expect our runtime checks at each shim to detect the error, auditors would no longer be able to retrieve an audit trail associated with the violation. Hence, it is important that we *hide* the logging bit at the gateway so that that the attacker cannot mount such an attack. In the following section, as we discuss our implementation, we describe how the logging bit is stored virtually in a way that is cryptographically secure while consuming 0 bits of overhead in the AuditTrailer.

6 AuditBox Implementation

We now discuss two performance optimizations to our protocol (§6.1) and our end-to-end prototype (§6.2).

6.1 Optimizing Verification

Updatable GMAC: To support mutable packets and dynamic paths, AuditBox computes the MAC twice for each packet at each hop: first to verify the packet when receiving it, and second to authenticate the packet before sending it out (§5). In our implementation, we use the GMAC algorithm [53]. While GMAC is one of the fastest authentication algorithms (thanks in part to acceleration from Intel's AES-NI instructions [40]), it still adds non-trivial overhead to packet processing.

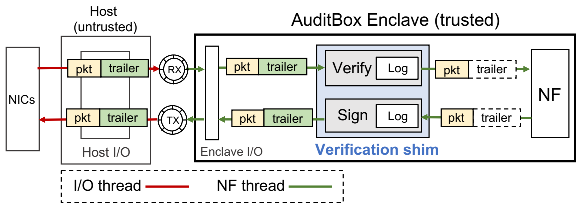


Figure 9: AuditBox software architecture (white boxes denote existing Safebricks components).

To reduce AuditBox's overhead, we implement a proven secure [52] *updatable* version of GMAC on top of EverCrypt [62], a formally verified cryptographic library. We use EverCrypt because it is easy to port into SGX and it is fast; its verified properties are a pleasant bonus. We defer to future work the extension of EverCrypt's formal verification to our updatable API.

The key optimization of updatable GMAC is to take advantage of GMAC's algebraic structure to securely *reuse* the first MAC when computing the second MAC [52]. With this optimization, the second MAC's cost is proportional to the number of modified data blocks, rather than the total packet length. In practice, this improves performance for NFs that only read the packet or that modify a small portion of it (§7).

In addition to a key and a message, GMAC requires as input an initialization vector (IV) that must be unique for each MAC invocation with a given key. For this we use the concatenation of the srcNF and pktID fields. This leads to a bound on how long we can use the same key K_σ. At 100Gbps, we would expect to overflow the pktID once every 22 days and hence we use we use a 14 day key rotation which ensures that the same IV is never used twice. When the pktID wraps around, one can use timestamps to differentiate entries in audit trails.

Secret Logging: In §5.4 we discussed that we must encrypt the bit used to mark which packets should be logged for auditing; we now describe how we introduce 0 bits of overhead and 0 computational overhead for unsampled packets. Our idea is to embed a *virtual logging bit* in the AuditTrailer. When the ingress gateway generates the *tag*, it appends this virtual logging bit as the last field when computing the MAC. For example, $tag = \text{MAC}_{K_\sigma}(pkt||pktID||srcNF||dstNF||\ 1)$ for a packet that should be logged. When the packet arrives at the verification shim, it verifies the MAC by appending a 0 (assuming most packets will not be logged). If the verification fails, the NF then appends 1 and performs a second MAC verification. The success of the second verification means the NF should log the packet, while failure indicates a malicious/mangled packet. We formally prove this approach is secure in §D.

Theorem 3 (Secret Logging Security) *Consider the game described in §D.5 with adversary $\mathcal{A}$. When the MAC algorithm is GMAC, the adversary's advantage is negligible.*

6.2 Prototype Details

We implement an end-to-end prototype using Safebricks [61]. Safebricks is an NFV framework that builds on top of DPDK [5] and runs NFs in Intel Enclaves [25]. While the idea of AuditBox could be applied to other NFV frameworks [48, 58, 59, 79], we choose Safebricks mainly to leverage its I/O optimization to avoid expensive enclave transitions. We modified approximately 4k lines of Rust to implement our protocols, and added 2.5k lines of C and 100 lines of x86 assembly to implement and test the updatable GMAC implementation. Here, we focus on the implementation of the verification shim and the AuditTrailer, which are the key enablers to avoid any NF changes.

Verification Shim: As shown in Figure 9, AuditBox inserts a verification shim that sits between the enclave I/O interface and the NF in each enclave. The shim implements our custom verification protocol using two modules: one verifies the incoming packets by checking the AuditTrailer, and the other updates the AuditTrailer on outgoing packets. Both modules use our *updatable* GMAC algorithm, and both modules have access to the logging function to save logs for offline auditing.

Packet Trailer: Typically, to carry a wrapper or metadata header through unmodified NFs, one incurs two overheads. First, one must *strip* the header from the packet and copy the packet (starting from the IP or Ethernet header) to the first byte of the packet buffer; the packet can then enter the NF as if it had come in off the wire. Second, one must *restore* the header to the packet; when packets have been modified by the NF this step may require complex algorithms to infer the correct mapping from original input packet to final output packet [32].

By using a trailer, AuditBox sidesteps these challenges in many cases. Before passing the pointer to the packet buffer into the NF, the shim adjusts the packet length to the end of the encapsulated packet, leaving the trailer at the bottom of the buffer and invisible to the NF code. Even if the packet has been modified or shortened, when the packet egresses the NF, the shim can simply restore the trailer sitting at the bottom of the allocated memory. When NFs extend the length of the packet, this overwrites the trailer and so the trailer must be restored similarly to the header operations above, however, we find for most NFs leaving the trailer at the base of the buffer is an effective way to improve performance (§7.3).

7 AuditBox Evaluation

AuditBox aims to enable real-time auditing for NFV deployments with low overhead. In this section, we evaluate its overhead using a testbed and traces and show that:

- AuditBox correctly detects a broad class of practical policy violations (§7.1).
- AuditBox enables auditing for unmodified NFs while adding less overhead than existing verification protocols (§7.2). We discuss our optimizations in §7.3.

Setup: Our testbed has four severs: three SGX servers (4-core 3.80 GHz Intel Xeon E3-1270 v6 CPUs, 64 GB RAM, Intel XL710 40Gb NICs) run AuditBox, and one server (dual-socketed Intel Xeon E5-2680 v2 GHz Xeon CPUs, with

	Blue Team Policies	Attacks	AuditBox		OPT
			Packet	Flow	
1	Mutable packets (Fig. 2): Load balancer modifies packets	-	no	no	yes
2	Mutable packets (Fig. 2): Load balancer modifies packets	modify	yes	yes	yes
3	Dynamic paths (Fig. 3): Light IPS reroutes packets	-	no	no	yes
4	Dynamic paths (Fig. 3): Light IPS reroutes packets	reroute	yes	yes	yes
5	Stateful NFs (Fig. 4): NAT tracks flow states	reorder	-	yes	no
6	Stateful NFs (Fig. 4): NAT tracks flow states	drop	-	yes	no

Table 1: Example scenarios that use AuditBox and OPT [46] to verify whether policy violations happen; "yes"/"no" indicate whether the system reports a violation. Shaded cells are correct auditing results.

10 cores, 128 GB RAM, Intel XL710 40Gb NICs) is used as a traffic generator. Each server runs Ubuntu 18.04 with Linux kernel 4.4.186. We use Moongen (DPDK-based) [30] to generate synthetic test traffic, as well as replay empirical traces [10] of varying packet sizes. We enable jumbo frames to allow the trailer to be added to large packets (which makes them larger than the default 1500 byte MTU). For each experiment, we report the median value of 20 tests, error bars represent one standard deviation (which in some cases are too small to see).

Sample NFs: AuditBox supports all existing NFs in Safebricks without any NF changes. To evaluate the performance of AuditBox, we choose three sample NFs with varying complexity: (1) *NAT* rewrites IP and TCP headers, representing NFs that modify packets (Figure 2); (2) *Stateful Firewall* which tracks connection states (Figure 4), configured with a campus ruleset (643 rules); and (3) *DPI*, which represents the most computationally expensive NF in Safebricks, configured with the Snort Community ruleset [14].

7.1 Functionality Evaluation

To validate the end-to-end effectiveness of AuditBox, we run different red-blue team exercises. In each scenario, the blue team (operator) chooses a service chain policy and the protocol (AuditBox packet, AuditBox flow, or OPT). The red team (attacker) randomly picks an attack vector. Then, we emulate this scenario with generated traffic. Finally, we reveal the ground truth and the audit trails to check if the protocol helped the blue team correctly identify/diagnose the attack.

We run these scenarios in a combination of a real testbed and a custom simulator. For the testbed we use one server to generate traffic, and three to run NFs. We introduce attacks (*e.g.,* modifying packets) using the I/O thread (Figure 9), and apply them when packets arrive at the NF. We built a custom simulator, where all NFs are connected via "one big switch" and this switch executes one or more adversarial actions (*e.g.,* rerouting) when forwarding packets between NFs.

Table 1 shows a subset of the scenarios, including the motivating examples from §2. In all scenarios, with AuditBox, the blue team successfully detects the policy violations and has correct auditing trails for doing so. OPT [46] detects some

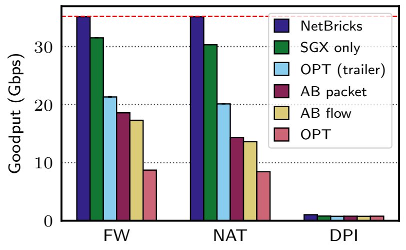


Figure 10: Cost of auditing for a single NF using empirical packet traces (AB represents AuditBox).

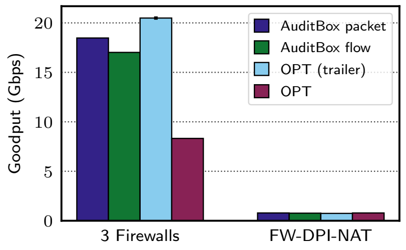


Figure 11: Cost of auditing for different chains of NFs using empirical packet traces.

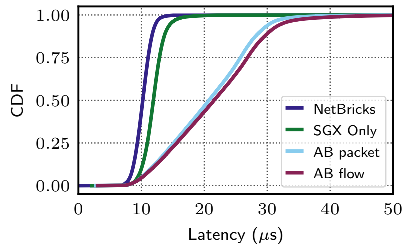


Figure 12: RTT at 80% load using empirical packet traces.

scenarios (scenarios 2 and 4), but has both false positives (1 and 3) and false negatives (5 and 6). Note that this is not unique to OPT; other VRPs [55] have the same issues as well.

7.2 Performance Evaluation

We first measure the performance of AuditBox for a single NF and a chain of NFs using empirical traffic [10]. For all tests, we run the NF using one core, and we report the goodput by excluding extra bytes used for verification. In these experiments, we compare against three alternative baselines, none of which are an apples-to-apples comparison with AuditBox (in that each was designed for a different purpose) but nonetheless we believe the comparison helps to put our results in context.

The first baseline is 'NetBricks' [59], which is a Rust-based NFV framework and does not use SGX. The second baseline is 'SGX-only', which runs NFs in enclaves but does not use any special protocol for verified routing. Our third and final baseline is OPT, which, as we have discussed (§2.4) is most closely related to AuditBox's use case but nonetheless cannot provide correct routing in the presence of dynamic routing and packet modifications.

Single NF: Figure 10 shows the goodput for running a single NF. The red dotted line shows the maximum rate of our traffic generator. NetBricks is able to process packets at the packet generator rate for both firewall and NAT. Compared to running NFs outside the enclave, the SGX baseline ('SGX only') incurs up to 15% overhead. Relative to running NFs in the enclave alone, AuditBox incurs 19% overhead for the firewall, 38% for the NAT, and 3% overhead for the DPI. The flow-level incurs slightly more overhead than the packet-level as it involves flow-table updates to track flow states.

AuditBox achieves up to two times higher goodput than the strawman OPT due to our reduced packet overhead (24B for AuditBox-pkt, 32B for AuditBox-flow vs. 84B for OPT) and our use of a trailer, instead of a header. We hypothesize that OPT's high overhead stems from its need to strip and restore large headers at each hop. To test this hypothesis, we implement OPT using our trailer optimization ('OPT-trailer'). The optimized OPT achieves a higher throughput than AuditBox, which is expected as it requires fewer MAC computations.

NF Chains: We also ran experiments to evaluate the performance of AuditBox under different NF chains across multiple nodes, similar to prior work [61]. As shown in Figure 11, for a chain of 3 firewalls AuditBox has 67% better goodput than OPT. Unlike AuditBox, which has a constant packet-size overhead regardless of chain length, OPT's header grows with chain length (116B for 3 NFs and 148B for 5 NFs), which contributes to the drop in goodput. Our optimized version, 'OPT-trailer,' gets better goodput after eliminating the header stripping and restoring overhead. We also evaluated a chain with a Firewall, followed by a DPI and a NAT. For this chain, both AuditBox and OPT achieve similar performance since the entire chain is bottlenecked by the heavy DPI.

Cost Analysis: It is worth putting the performance numbers in context to see the effective "cost" of auditability. Consider an organization deploying specialized hardware appliances for regulatory compliance. Today, this costs roughly \$600-\$3500 per-Gbps for firewalls and \$6000-\$12000 per-Gbps for IPS (*e.g.,* Cisco FirePOWER 8140 [4], Juniper Networks SRX345 [9]). If this organization shifts to NFV on commodity servers because of the auditability offered by AuditBox, we estimate the cost will be 12X-60X lower for the firewall, and 1.9X-9X lower for the IPS.While we acknowledge that any such cost analysis is fraught with uncertainties (e.g., cost at scale, reliability, service contracts, power), this rough estimate suggests that RegTech customers can still achieve financial gains through NFV.

Latency Overhead: Figure 12 shows the latency overhead of AuditBox using empirical packet traces. For each test, we measure the RTT at 80% of the maximum throughput of the system under test as a metric for latency. Compared to the SGX baseline, AuditBox-pkt adds around 18μs for 99^{th} percentile latency, and AuditBox-flow adds another 6μs.

Sweep packet size and NF type: Figure 16 (Appendix E) shows the overhead of AuditBox for varying packet sizes and NF types. Across all NFs, the overhead of AuditBox decreases as packet sizes increase. Since NAT modifies the packet, we do not use our updatable GMAC, making the overhead of AuditBox more noticeable.

7.3 Impact of Our Optimizations

Header vs Trailer: Figure 13 shows the benefit of implementing AuditTrailer as a trailer instead of a header. Unlike our trailer which can be made invisible inside the NF (shown in Figure 9), the shim needs to strip off the added header

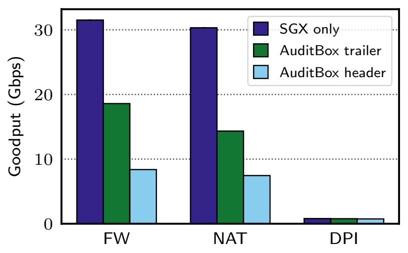


Figure 13: Effect of using trailer vs. header to carry verification data.

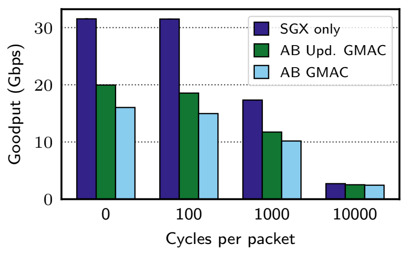


Figure 14: Effect of using updatable GMAC as NF complexity scales.

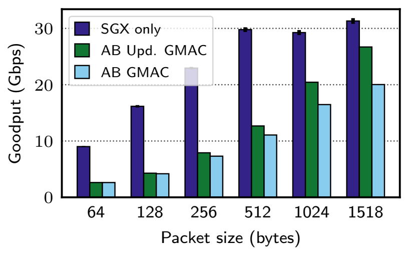


Figure 15: Effect of using updatable GMAC for varying packet sizes.

before delivering the packet to the NF to avoid modifying the NF. After the packet is processed, the shim needs to prepend the header before sending it to the next NF. These header manipulations slow down packet processing and verification, resulting in decreases of goodput by up 2×.

Regular GMAC vs Updatable GMAC: In Figures 14 and 15, we compare the performance of AuditBox when using our implementation of the updatable GMAC with the regular GMAC that is used by popular cryptographic libraries (*e.g.,* OpenSSL [12], NSS [11]). When varying the NF complexity (cycles/pkt), using updatable GMAC improves goodput by up to 23%. When varying the packet sizes, our updatable GMAC outperforms the regular GMAC by up to 25% for large packets. These improvements can largely be attributed to reusing the first MAC to compute the second MAC, which avoids the recomputing overhead for non-modified data blocks.

Dedicated log bit vs Virtual log bit: We compared our virtual logging bit with an approach with a real encrypted bit. On average, using a virtual log bit improves goodput by 4-5%.

8 Related Work

We have already discussed existing VRPs (*e.g.,* [46, 55]), service chaining policies and mechanisms (*e.g.,* [32, 58]), and how auditing is different from network verification (*e.g.,* [44, 45, 50, 60]). Here, we focus on other classes of related work.

Traditional NFV Frameworks: Prior NFV [6] frameworks focus on management [35, 42, 58, 65, 67], performance [28, 43, 51, 79], and programability [23, 48, 59]. We argue auditability should be added as a first-order feature of the NFV framework, which would relieve enterprises' concerns about deploying the NFV for security-critical services or outsourcing them to a third-party provider [33, 68].

Securing NFs: Prior work has proposed the use of SGX to protect an NF's source code [61, 72], an NF's state [69], traffic metadata [29], and to support end-to-end encryption [39, 56]. While these enhance security for NFs, they only focus on individual NFs on a single server, and they do not provide mechanisms to audit the entire service chain.

NF Verification: NF verification [27, 75, 76, 78] guarantees that a certain NF implementation is correct (memory-safe, crash-free, *etc.*). AuditBox assumes that vendors have 'certified' NF implementations as secure, and we expect this class of work would strengthen trust in such vendor certifications.

Verifiable fault localization and measurements: In addition to VRPs, auditing is also related to prior work on secure fault localization (*e.g.,* [81, 82]), robust sampling algorithms (*e.g.,* [57]), verifiable performance measurements (*e.g.,* [16]), and secure network provenance (SNP) (*e.g.,* [37, 84]). Some of our building blocks share conceptual similarity with these efforts. However, they focus on different goals. For example, SNP leverages tamper-evident logging [38] to identify misbehavior offline, unlike the runtime guarantees we provide.

9 Conclusions

In this paper, we have presented AuditBox, a framework that brings NFV to security-critical environments that require auditing. By leveraging enclaves to run NFs and continuously verifying the traffic between NFs, AuditBox provides a strong run-time guarantee that the NFV system remains in compliance with policy goals. AuditBox also supports traditional offline auditing by generating audit trails for manual inspection.

We see AuditBox as a first step in a conversation with regulators. Would our audit trails be more trustworthy if they included additional data? Should we combine our SGX protection with formal verification of the NF code [27]? In practice, should AuditBox be combined with formal verification of operator policies [60]? We expect, and hope, to see considerable evolution in supporting the RegTech space beyond AuditBox's current capabilities.

In the long run, adoption only comes when human auditors *feel comfortable* trusting the guarantees provided by any particular system. Our hope is that by not only replicating the capabilities that auditors have today, but also strengthening SFCs with runtime correctness guarantees, AuditBox will merit the trust of auditors and hence hasten the adoption of NFV in security-sensitive networks.

Acknowledgements: We thank our shepherd Alex Snoeren and the anonymous reviewers for their insightful comments. We also thank Rishabh Poddar, Limin Jia and Sze Yiu Chaus. This work was supported in part by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, ERDF Project AIDA (POCI-01-0247-FEDER-045907), in part by the NSF/VMware Partnership on Software Defined Infrastructure as a Foundation for Clean-Slate Computing Security (SDI-CSCS) program under Award No. CNS-1700521, and by NSF awards 1440056 and 1440065.

References

[1] ARM TrustZone. https://developer.arm.com/technologies/trustzone.

[2] AT&T Domain 2.0 Vision White Paper. https://www.att.com/Common/about_us/pdf/AT&T%20Domai%202.0%20Vision%20White%20Paper.pdf.

[3] Blue Planet NFV Service Orchestration. https://www.blueplanet.com/products/nfv-orchestration.html.

[4] Cisco-FirePOWER-8140. https://www.cdw.com/product/Cisco-FirePOWER-8140-security-appliance.

[5] Data Plane Development Kit (DPDK). http://www.dpdk.org/.

[6] European Telecommunications Standards Institute. NFV whitepaper. https://portal.etsi.org/NFV/NFV_White_Paper.pdf.

[7] Family Educational Rights and Privacy Act (FERPA). https://www2.ed.gov/policy/gen/guid/fpco/ferpa/index.html.

[8] HIPAA Security Rule. https://www.hhs.gov/hipaa/for-professionals/security/laws-regulations/index.html.

[9] Juniper Networks SRX345 Services Gateway . https://www.cdw.com/product/juniper-networks-srx345\-services-gateway-security-appliance/4739102?pfm=srh.

[10] Malware Capture Facility Project. https://www.stratosphereips.org/datasets-normal.

[11] Network Security Services. https://hg.mozilla.org/projects/nss.

[12] OpenSSL-Cryptography and SSL/TLS Toolkit. https://www.openssl.org/.

[13] Payment Card Industry Security Standards Council. https://www.pcisecuritystandards.org/.

[14] Snort Community Rulesets. https://www.snort.org/downloads.

[15] ISO/IEC 27001 Information Security Management. https://www.iso.org/isoiec-27001-information-security.html, 2013.

[16] Katerina J. Argyraki, Petros Maniatis, and Ankit Singla. Verifiable network-performance measurements. In *Proceedings of the 2010 ACM Conference on Emerging Networking Experiments and Technology, CoNEXT 2010, Philadelphia, PA, USA, November 30 - December 03, 2010*, page 1. ACM, 2010.

[17] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 689–703, Savannah, GA, November 2016. USENIX Association.

[18] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014.

[19] M. Bellare and P. Rogaway. Entity authentication and key distribution. In *Advances in Cryptology*. CRYPTO, 1993.

[20] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 1996.

[21] Dan Boneh and Victor Shoup. *A Graduate Course in Applied Cryptography*. Jan 2020.

[22] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks are Practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, 2017.

[23] Anat Bremler-Barr, Yotam Harchol, and David Hay. OpenBox: A software-defined framework for developing, deploying, and managing network functions. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, 2016.

[24] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. SGXSpectre attacks: Leaking enclave secrets via speculative execution. *arXiv preprint arXiv:1802.09085*, 2018.

[25] Victor Costan and Srinivas Devadas. Intel SGX Explained. Cryptology ePrint Archive, Report 2016/086 (2016). https://datatracker.ietf.org/doc/draft-brockners-proof-of-transit/.

[26] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium (USENIX Security 16)*, 2016.

[27] Mihai Dobrescu and Katerina Argyraki. Software dataplane verification. *Communications of the ACM*, 2015.

[28] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. Routebricks: Exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, 2009.

[29] Huayi Duan, Cong Wang, Xingliang Yuan, Yajin Zhou, Qian Wang, and Kui Ren. Lightbox: Full-stack protected stateful middlebox at lightning speed. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, 2019.

[30] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. Moongen: A scriptable high-speed packet generator. In *Proceedings of the 2015 ACM Internet Measurement Conference, IMC 2015, Tokyo, Japan, October 28-30, 2015*, pages 275–287. ACM, 2015.

[31] Seyed K. Fayaz, Tianlong Yu, Yoshiaki Tobioka, Sagar Chaki, and Vyas Sekar. Buzz: Testing context-dependent policies in stateful networks. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, 2016.

[32] Seyed Kaveh Fayazbakhsh, Luis Chiang, Vyas Sekar, Minlan Yu, and Jeffrey C Mogul. Enforcing Network-Wide Policies in the Presence of Dynamic Middlebox Actions using FlowTags. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014.

[33] Seyed Kaveh Fayazbakhsh, Michael K Reiter, and Vyas Sekar. Verifiable network function outsourcing: requirements, challenges, and roadmap. In *Proceedings of the 2013 workshop on Hot topics in middleboxes and network function virtualization*, pages 25–30, 2013.

[34] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, October 2017.

[35] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. Opennf: enabling innovation in network function control. In *ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014*, 2014.

[36] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security*, pages 1–6, 2017.

[37] Andreas Haeberlen, Ioannis Avramopoulos, Jennifer Rexford, and Peter Druschel. NetReview: Detecting when interdomain routing goes wrong. In *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI'09)*, Apr 2009.

[38] Andreas Haeberlen, Petr Kuznetsov, and Peter Druschel. PeerReview: Practical accountability for distributed systems. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*, Oct 2007.

[39] Juhyeng Han, Seongmin Kim, Jaehyeong Ha, and Dongsu Han. SGX-Box: Enabling Visibility on Encrypted Traffic using a Secure Middlebox Module. In *Proceedings of the First Asia-Pacific Workshop on Networking - APNet'17*, 2017.

[40] Gael Hofemeier and Robert Chesebrough. Introduction to Intel AES-NI and Intel Secure Key instructions. `https://software.intel.com/en-us/articles/introduction-to-intel\-aes-ni-and-intel-secure-key-instructions`, 2014.

[41] Joint Task Force Transformation Initiative. Security and privacy controls for federal information systems and organizations. Technical Report Special Publication 800-53 (Revision 4), NIST, April 2013.

[42] Murad Kablan, Blake Caldwell, Richard Han, Hani Jamjoom, and Eric Keller. Stateless network functions. In *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization, HotMiddlebox@SIGCOMM 2015, London, United Kingdom, August 21, 2015*, 2015.

[43] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. Metron: NFV service chains at the true speed of the underlying hardware. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018.

[44] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real time network policy checking using header space analysis. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013.

[45] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P Brighten Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013.

[46] Tiffany Hyun-Jin Kim, Cristina Basescu, Limin Jia, Soo Bum Lee, Yih-Chun Hu, and Adrian Perrig. Lightweight source authentication and path validation. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, 2014.

[47] Markus Legner, Tobias Klenze, Marc Wyss, Christoph Sprenger, and Adrian Perrig. Epic: Every packet is checked in the data plane of a path-aware internet. In *29th USENIX Security Symposium*, USENIX Security '20, 2020.

[48] Guyue Liu, Yuxin Ren, Mykola Yurchenko, K. K. Ramakrishnan, and Timothy Wood. Microboxes: High performance nfv with customizable, asynchronous tcp stacks and dynamic subscriptions. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, 2018.

[49] Xin Liu, Ang Li, Xiaowei Yang, and David Wetherall. Passport: Secure and adoptable source authentication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, 2008.

[50] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with anteater. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, 2011.

[51] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, et al. ClickOS and the art of network function virtualization. In *USENIX NSDI*, 2014.

[52] D. McGrew. Efficient authentication of large , dynamic data sets using Galois/counter mode (GCM). In *In Security in Storage Workshop*. IEEE, 2005.

[53] D. McGrew and J. Viega. The Galois/counter mode of operation (GCM). In *Submission to NIST Modes of Operation Process*, 2004.

[54] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How SGX amplifies the power of cache attacks. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 69–90. Springer, 2017.

[55] Jad Naous, Michael Walfish, Antonio Nicolosi, David Mazières, Michael Miller, and Arun Seehra. Verifying and enforcing network paths with icing. In *Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies*, CoNEXT '11, 2011.

[56] David Naylor, Richard Li, Christos Gkantsidis, Thomas Karagiannis, and Peter Steenkiste. And then there were more: Secure communication for more than two parties. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '17, 2017.

[57] Pavlos Nikolopoulos, Christos Pappas, Katerina J. Argyraki, and Adrian Perrig. Retroactive packet sampling for traffic receipts. *POMACS*, 3(1):19:1–19:39, 2019.

[58] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: A framework for NFV applications. In *SOSP '15*. ACM, 2015.

[59] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. NetBricks: Taking the V out of NFV. In *12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '16, pages 203–216, Savannah, GA, November 2016. USENIX Association.

[60] Aurojit Panda, Ori Lahav, Katerina Argyraki, Mooly Sagiv, and Scott Shenker. Verifying Reachability in Networks with Mutable Datapaths. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017.

[61] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. SafeBricks: Shielding Network Functions in the Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018.

[62] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cedric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph Wintersteiger, and Santiago Zanella-Beguelin. EverCrypt: A fast, verified, cross-platform cryptographic provider. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, May 2020.

[63] Zafar Ayyub Qazi, Rui Miao, Cheng-Chun Tu, Vyas Sekar, Luis Chiang, and Minlan Yu. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, 2013.

[64] P. Quinn, U. Elzur, and C. Pignataro. Network Service Header (NSH). RFC 8300, January 2018.

[65] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. Split/merge: System support for elastic execution in virtual middleboxes. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, 2013.

[66] Grand View Research. RegTech Market Size Worth $55.28 Billion by 2025. `https://www.bloomberg.com/press-releases/2019-08-14/regtech-market-size-worth-55-28-billion-by-2025\-cagr-52-8-grand-view-research-inc`, Aug 2019.

[67] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K. Reiter, and Guangyu Shi. Design and implementation of a consolidated middlebox architecture. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, 2012.

[68] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In *Proceedings of the 2012 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, SIGCOMM '12, pages 13–24, New York, NY, USA, 2012. ACM.

[69] Ming-Wei Shih, Mohan Kumar, Taesoo Kim, and Ada Gavrilovska. S-NFV: Securing nfv states by using SGX. In *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, SDN-NFV Security, 2016.

[70] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *24th Annual Network and Distributed System Security Symposium, NDSS*, 2017.

[71] V. Shoup. Sequences of games: a tool for taming complexity in security proofs. In *IACR Cryptology*. ePrint Archive, 2004.

[72] Bohdan Trach, Alfred Krohmer, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. ShieldBox: Secure Middleboxes using Shielded Execution. In *Proceedings of the Symposium on SDN Research - SOSR '18*, 2018.

[73] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 991–1008, 2018.

[74] Yifei Yuan, Soo-Jin Moon, Sahil Uppal, Limin Jia, and Vyas Sekar. NetSMC: A custom symbolic model checker for stateful network verification. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation*, NSDI'20, 2020.

[75] Arseniy Zaostrovnykh, Solal Pirelli, Rishabh Iyer, Matteo Rizzo, Luis Pedrosa, Katerina Argyraki, and George Candea. Verifying Software Network Functions with No Verification Expertise. In *Proceedings of the 27th Symposium on Operating Systems Principles - SOSP '19)*, 2019.

[76] Arseniy Zaostrovnykh, Solal Pirelli, Luis Pedrosa, Katerina Argyraki, and George Candea. A Formally Verified NAT. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication - SIGCOMM '17*, 2017.

[77] Fuyuan Zhang, Limin Jia, Cristina Basescu, Tiffany Hyun-Jin Kim, Yih-Chun Hu, and Adrian Perrig. Mechanized Network Origin and Path Authenticity Proofs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security - CCS '14*, 2014.

[78] Kaiyuan Zhang, Danyang Zhuo, Aditya Akella, Arvind Krishnamurthy, and Xi Wang. Automated verification of customizable middlebox properties with gravel. In *Proceedings of the 17th USENIX Conference on Networked Systems Design and Implementation*, NSDI '20, 2020.

[79] Wei Zhang, Guyue Liu, Wenhui Zhang, Neel Shah, Phillip Lopreiato, Gregoire Todeschi, K.K. Ramakrishnan, and Timothy Wood. OpenNetVM: A platform for high performance network service chains. In *HotMiddlebox*. ACM, 2016.

[80] Xin Zhang, Abhishek Jain, and Adrian Perrig. Packet-dropping adversary identification for data plane security. In *Proceedings of the 2008 ACM CoNEXT Conference on - CONEXT '08*, 2008.

[81] Xin Zhang, Chang Lan, and Adrian Perrig. Secure and Scalable Fault Localization under Dynamic Traffic Patterns. In *2012 IEEE Symposium on Security and Privacy*. IEEE, May 2012.

[82] Xin Zhang, Zongwei Zhou, Geoff Hasker, Adrian Perrig, and Virgil Gligor. Network fault localization with small TCB. In *2011 19th IEEE International Conference on Network Protocols*. IEEE, 2011.

[83] Xin Zhang, Zongwei Zhou, Hsu-Chun Hsiao, Tiffany Hyun-Jin Kim, Adrian Perrig, and Patrick Tague. ShortMAC: Efficient Data-Plane Fault Localization. In *19th Annual Network and Distributed System Security Symposium (NDSS)'12*, 2012.

[84] Wenchao Zhou, Qiong Fei, Arjun Narayan, Andreas Haeberlen, Boon Thau Loo, and Micah Sherr. Secure network provenance. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles - SOSP '11*, 2011.

A Modeling Duplicate Packets

The formalization of Property 4 assumes that the same packet is never sent twice. This seems to be a reasonable assumption. As we are forwarding under L2, an NF_A and an NF_B will always have different Ethernet headers even if the IP, TCP/UDP, and Payload are identical. And it seems reasonable to assume that an NF_A will never transmit the same packet twice either – even a re-transmitted TCP packet will come with a different IPID value.

Nonetheless, we could remove these requirements and allow NFs to transmit the same packet repeatedly by defining correctness as follows:

Let $\text{SEND-TO}_{F,P}(E) \to \{0,1\}$ be defined:

```
function $\text{SEND-TO}_{\text{NF}_i,p_i}(e)$
    if $e.\text{op} = \text{SEND} \wedge e.\text{pkt} = p_i \wedge \text{policy}(p_i, e.\text{NF}) = \text{NF}_i$
then
        return 1
    else
        return 0
```

Let $\text{RECV-PKT}_{F,P}(E) \to \{0,1\}$ be defined:

```
function $\text{RECV-PKT}_{\text{NF}_i,p_i}(e)$
    if $e.\text{op} = \text{RECV} \wedge e.\text{pkt} = p_i \wedge e.\text{NF} = \text{NF}_i$ then
        return 1
    else
        return 0
```

To allow identical packets, we could then say the system was correct under packet correctness iff:

$$\forall e \in E \text{ s.t. } e.\text{op} = \text{RECV}:$$
$$\sum_{i=1}^{e.\text{t}} \text{SEND-TO}_{e.\text{NF},e.\text{pkt}}(E[i]) = \sum_{i=1}^{e.\text{t}} \text{RECV-PKT}_{e.\text{NF},e.\text{pkt}}(E[i])$$

B Additional Definitions

B.1 Modeling Packet Exit

We define GW_{out} for when a packet exits the cluster as follows:

Algorithm 3 Model of Packets Exiting the Cluster

```
1: function $GW_{\text{OUT}}(\text{input})$
2:     input $\leftarrow f_{in}(input)$
3:     if input $\neq \perp$ then
4:         $E$.append(input, $\perp$, $\text{GW}_{\text{out}}$, $E$.length + 1)
5:     return input
```

B.2 Audit Trail Definition

While our routing protocol provides hop-by-hop guarantees, auditors are familiar with end to end 'what you see is what you get' *evidence* that packets are indeed following the correct route. To provide auditors the confidence of proven correctness with empirical evidence, AuditBox provides empirical evidence in the form of audit trails. Following the model of our event log E, one can take a packet in any of its states (prior to entry, between two NFs, post exit) and compute the forms the packet took on and all NFs it traversed across its entire traversal of the system. While 5.4 describes how this works in practice, we describe audit trails in the context of our model with Algorithm 4.

Algorithm 4 Audit Trail Definition

```
function CAUSED-BY(event)
    if event.pkt$_{in}$ = $\perp$ then
        return [(event.NF, event.pkt$_{out}$)]
    trail ← (event.pkt$_{in}$, event.NF, event.pkt$_{out}$)
    prev ← get-event$_E$(event.pkt$_{in}$, pkt$_{out}$)
    return caused-by(prev) + trail
function LEADS-TO(event)
    if event.pkt$_{out}$ = $\perp$ then
        return [(event.pkt$_{in}$, event.NF)]
    prev ← get-event$_E$(event.pkt$_{out}$, pkt$_{in}$)
    prev_trail ← (prev.pkt$_{in}$, prev.NF, prev.pkt$_{out}$)
    return prev_trail + leads-to(prev)
function AUDIT-TRAIL(event)
    return CAUSED-BY(event) + LEADS-TO(event)
```

Note that the definition assumes that all transmitted packets are unique.

C Pseudocode

In the algorithms below, we expand the notion of a packet to also include the AuditBox trailer.

In the flow-verification algorithm (Algorithm 6), we assume each NF, including the gateway, maintains a flow-counter table FC which maps a flow ID and destination NF to a counter value:

$$ctr \leftarrow FC[flowID, NF]$$

Performing a lookup with a new $flowID, NF$ pair implicitly initializes the counter to zero. Comments highlight differences relative to the packet-verification algorithm (Algorithm 5).

Algorithm 5 Packet Verification Protocol

```
Input: The shared symmetric key $K_\sigma$ for the current epoch $\sigma$.
 1: ▷ Generate an AuditTrailer for each packet at the gateway GW$_{in}$
 2: function GENERATE(pkt)
 3:     $out.pkt = pkt$
 4:     $out.pktID = genPktID(pkt)$
 5:     $out.srcNF = \mathrm{GW}_{in}$
 6:     $out.dstNF = Policy(pkt, \mathrm{GW}_{in})$
 7:     $out.tag = \mathrm{MAC}_{K_\sigma}(out.pkt|out.pktID|out.srcNF|out.dstNF)$
 8:     return $out$
 9:
10: ▷ Process a packet $in$ at NF$_i$
11: function PROCESS$_i$(in)
12:     ok ← $in.dstNF = \mathrm{NF}_i$   $\wedge$
13:             Verify$_{K_\sigma}(in.pkt|in.pktID|in.srcNF|in.dstNF, in.tag)$
14:     if ok then:
15:         $out.pkt = f_i(in.pkt)$
16:         $out.pktID = in.pktID$
17:         $out.srcNF = \mathrm{NF}_i$
18:         $out.dstNF = Policy(out.pkt, \mathrm{NF}_i)$
19:         $out.tag = \mathrm{MAC}_{K_\sigma}(out.pkt|out.pktID|out.srcNF|out.dstNF)$
20:     else
21:         $out \leftarrow \perp$       ▷ Drop packet and raise alert
    return $out$
```

Algorithm 6 Flow Verification Protocol

```
Input: The shared symmetric key $K_\sigma$ for the current epoch $\sigma$.
 1: ▷ Generate a flow AuditTrailer for each packet at the gateway GW$_{in}$
 2: function GENERATE(pkt)
 3:     ▷ Same as Packet Verification
 4:     $out.pkt = pkt$
 5:     $out.pktID = genPktID(pkt)$
 6:     $out.srcNF = \mathrm{GW}_{in}$
 7:     $out.dstNF = Policy(pkt, \mathrm{GW}_{in})$
 8:     ▷ New for Flow Verification
 9:     $out.flowID = computeFlowID(pkt)$
10:     $out.seqNum = FC[out.flowID, out.dstNF]{+}{+}$
11:     $out.tag = \mathrm{MAC}_{K_\sigma}(out.pkt|out.pktID|out.srcNF|out.dstNF$
            $|out.flowID|out.seqNum)$
12:     return $out$
13:
14: ▷ Process a packet $in$ at NF$_i$
15: function PROCESS$_i$(in)
16:     ok ← $in.dstNF = \mathrm{NF}_i$
17:         $\wedge$  Verify$_{K_\sigma}(in.pkt|in.pktID|in.srcNF|in.dstNF|in.flowID$
                $|in.seqNum, in.tag)$
18:         $\wedge$  $in.seqNum = FC[in.flowID, in.srcNF]$    ▷ New
19:     if ok then:
20:         $FC[in.flowID, in.srcNF]{+}{+}$    ▷ New
21:         $out.pkt = f_i(in.pkt)$
22:         $out.pktID = in.pktID$
23:         $out.srcNF = \mathrm{NF}_i$
24:         $out.dstNF = Policy(out.pkt, \mathrm{NF}_i)$
25:         ▷ New for Flow Verification
26:         $out.flowID = in.flowID$
27:         $out.seqNum = FC[out.flowID, out.dstNF]{+}{+}$
28:         $out.tag = \mathrm{MAC}_{K_\sigma}(out.pkt|out.pktID|out.srcNF|out.dstNF$
                $|out.flowID|out.seqNum)$
29:     else
30:         $out \leftarrow \perp$       ▷ Drop packet and raise alert
    return $out$
```

D Security Proofs

D.1 Cryptographic Assumptions

We introduce the standard notation we use and the standard cryptographic assumptions we make.

We write $x|y$ for the uniquely delimited (either via tags or fixed widths) concatenation of x and y. Hence, $x_0|y_0 == x_1|y_1$ implies $x_0 == x_1$ and $y_0 == y_1$.

Our scheme relies on a Message Authentication Code (MAC) scheme, which consists of three algorithms. A symmetric key K is generated by the KeyGen() algorithm. We write $\tau \leftarrow \text{MAC}_K(m)$ to indicate using key K to compute a MAC *tag* τ on message m, and $\text{Verify}_K(m,\tau)$ for the algorithm that uses key K to check the validity of tag τ for message m.

We assume that the MAC scheme is existentially unforgeable under chosen-message attacks (EUF-CMA) [21]. Intuitively, the definition says that an adversary who can request validly computed tags for n adaptively chosen messages cannot produce a new pair $(m,\tau) \notin \{(m_1,\tau_1),\ldots,(m_n,\tau_n)\}$ which passes $\text{Verify}_K(m,\tau)$. Standard algorithms, such as HMAC [20] and GMAC [53], are EUF-CMA secure.

D.2 Security Definition

To formalize AuditBox's security, we take the standard approach of defining our desired security property via a cryptographic *game* involving a challenger $\mathcal{C}$, and a probabilistic, polynomial-time adversary $\mathcal{A}$, which intuitively corresponds to the untrusted network, Φ. The game can be instantiated with an *audit protocol* that supplies NF functions f_i.

The challenger begins the game by creating an empty event log E and calling KeyGen() to produce key K. The adversary is then allowed to run and can call the following oracles which represent the various NFs in the system.

1. $p_{out} \leftarrow \text{Oracle-GW}_{IN}(p_{in})$ allows the adversary to introduce a new packet p_{in} to the gateway and obtain the packet p_{out} produced by the gateway.
2. $p_{out} \leftarrow \text{Oracle-NF}_i(p_{in})$ invokes NF_i on the adversarially supplied input packet p_{in} and gives the adversary the resulting packet p_{out}.

The challenger instantiates these oracles using the models described in §4.1. Specifically, $\text{Oracle-GW}_{IN}(p_{in})$ runs Algorithm 2 using the protocol-supplied function f_{in}, and $\text{Oracle-NF}_i(p_{in})$ runs Algorithm 1 using the protocol-supplied function f_i. Note that both oracles append to the event log E

When the adversary terminates, the game ends and outputs E.

D.3 Security Proof of Packet Correctness

Theorem 1 (Packet Correctness) *Consider the game described above with adversary $\mathcal{A}$ and instantiated with the AuditBox packet correctness protocol (§5.2). Specifically, looking at Algorithm 5, we instantiate f_{in} with the function* GENERATE *and f_i with* PROCESS$_i$*. The probability that the game outputs an event log E that violates Property 1 is negligible.*

Proof of Theorem 1: We prove security by reducing the security of our protocol to the EUF-CMA security of our MAC algorithm though a series of cryptographic games [19, 71]. The initial game matches the game defined in §D.2, and each subsequent game idealizes a portion of the protocol. At each step, we calculate the adversary's success in distinguishing between the two games.

Game 0 is defined as in §D.2 with an adversary $\mathcal{A}$ which queries its oracles a total of α times.

Game 1 is the same as above, except that the NF oracle, when given an input packet p_{in}, computes

$$m \leftarrow p_{in}.pkt|p_{in}.pktID|p_{in}.srcNF|p_{in}.dstNF$$

and immediately rejects the packet if m was not previously passed as an argument to $\text{MAC}_K(\cdot)$ (*i.e.,* $\mathcal{C}$ keeps a list $\mathcal{L}$ of all values passed to $\text{MAC}_K(\cdot)$, and upon receiving a packet p_{in} on which it would normally call $\text{Verify}_K(m,\tau)$, it looks up m in $\mathcal{L}$ and accepts/rejects based on the lookup, without looking at τ).

The adversary can distinguish Game 1 from Game 0 only if it can forge a valid tag τ for m. This happens with probability at most $\texttt{EUF-CMA}(\alpha)$, the probability of breaking our unforgability assumption given at most α chosen-message tags.

From Game 1, we show that Property 1 perfectly holds, which means that the probability that an adversary can break Property 1 is at most $\texttt{EUF-CMA}(\alpha)$, which is, by definition, negligible when we employ a secure MAC scheme.

In Game 1, consider any $e_b \in E$ such that $e_b.pkt_{in} \neq \perp$. By the construction of the *NF* model (Algorithm 1) an event e_b is only added to E after the NF runs f_i, which we instantiated with PROCESS$_i$. PROCESS$_i$ ensures $e_b.pkt_{in}.dstNF == e_b.NF$ (Line 12 of Algorithm 5). In Game 1, instead of running the MAC's verification algorithm on Line 13, it checks that the computed m is on the list $\mathcal{L}$ of previously MAC'ed messages. For this check to succeed, there must have been a logically earlier MAC call, which must have occurred during a previous invocation of Oracle-GW_{IN} or Oracle-NF_i, which each compute a tag on their outbound packet. For the verification lookup in $\mathcal{L}$ to succeed, that MAC call must have computed an m' for its output packet, where $m' == m$. This earlier oracle invocation would have produced event $e_a = (p, e_b.pkt_{in}, e_b.pkt_{in}.srcNF, i)$ for some other input packet p and index i, with $i < e_b.t$, since this was an earlier invocation, and the log E was necessarily shorter. Hence, the equality of m and m' implies that we have $e_a.pkt_{out} = e_b.pkt_{in}$, and $policy(e_a.pkt_{out}, e_a.NF) = e_b.NF$ (note Lines 6 and 18 of Algorithm 5), satisfying Property 1. ■

D.4 Security Proof of Flow Correctness

Theorem 2 (Flow Correctness) *Consider the game described in §D.2 with adversary $\mathcal{A}$ and instantiated with the AuditBox flow-correctness protocol (§5.3). Specifically, looking at Algorithm 6, we instantiate f_{in} with the function* GENERATE *and f_i with* PROCESS$_i$*. The probability that the game outputs an event log E that violates Properties 1-4 is negligible.*

Proof of Theorem 2:

We prove Theorem 2 by considering each property in turn.

Proof of Property 1 We begin by observing that compared with the packet-verification protocol, the flow-verification protocol

1. Includes in its AuditTrailer a superset of the packet-verification fields.
2. Computes the values the packet-verification fields in an identical manner.
3. Computes MACs over a superset of the packet-verification fields. algor
4. Performs a superset of the validation checks (compare Lines 16-18 of Algorithm 6 with Lines 12-13 of Algorithm 5).

Hence, we can apply an identical set of arguments as we did in our proof of Theorem 1 to show that Property 1 still holds when we employ a secure MAC. In the discussion, Property 1 allows us to assume that all packets in E originated from an NF, and hence we no longer need worry about adversarially mangled or injected packets.

Proof of Property 2 To show that Property 2 (*i.e.,* no packet injection or modification) holds, choose an arbitrary $\text{NF}_a, \text{NF}_b \in F$, and $e_{a1}, e_{a2}, e_{b2} \in E$ such that Lines 5-9 of Property 2 hold. We will show that there must exist $e_{b1} \in E$ such that

$$e_{b1}.\text{t} < e_{b2}.\text{t} \wedge e_{b1}.\text{NF} = \text{NF}_b \wedge e_{b1}.\text{pkt}_{\text{in}} = e_{a1}.\text{pkt}_{\text{out}}$$

When $e_{a1}.\text{pkt}_{\text{out}}$ was produced by $\textsc{Process}_a$, it was assigned a flow sequence number $FC_a[e_{a1}.\text{pkt}_{\text{out}}.\mathit{flowID},\ e_{a1}.\text{pkt}_{\text{out}}.\mathit{srcNF}]$ (Line 27 of Algorithm 6), and NF_a immediately increments the counter.

When we subsequently call $\textsc{Process}_a$ to produce $e_{a2}.\text{pkt}_{\text{out}}$ (we know this is a subsequent invocation of $\textsc{Process}_a$ because $e_{a1}.\text{t} < e_{a2}.\text{t}$), we can show that it will read the same counter from FC_a, which by observation always increases monotonically. Hence, it must be that $e_{a2}.\text{pkt}_{\text{out}}.\mathit{seqNum} > e_{a1}.\text{pkt}_{\text{out}}.\mathit{seqNum}$.

We can show that $\textsc{Process}_a$ accesses the same counter by showing that $e_{a1}.\text{pkt}_{\text{out}}.\mathit{flowID} = e_{a2}.\text{pkt}_{\text{out}}.\mathit{flowID} \wedge e_{a1}.\text{pkt}_{\text{out}}.\mathit{srcNF} = e_{a2}.\text{pkt}_{\text{out}}.\mathit{srcNF}$. The first is straightforward, since $e_{a1}.\text{pkt}_{\text{out}} = e_{a2}.\text{pkt}_{\text{out}} \implies \mathit{flow}(e_{a1}) = \mathit{flow}(e_{a2})$, and the second follows from Line 7 of Property 2.

For $e_{a2}.\text{pkt}_{\text{out}}$ to have passed the "ok" check in $\textsc{Process}_b$, and hence to have generated e_{b2}, it must have passed the check on Line 18, which means that at the time, $e_{a2}.\text{pkt}_{\text{out}}.\mathit{seqNum} = FC_b[e_{a2}.\text{pkt}_{\text{out}}.\mathit{flowID},\ e_{a2}.\text{pkt}_{\text{out}}.\mathit{srcNF}]$. Now consider the set $E_{b1} \subseteq E$ of all e_{b1} such that $e_{b1}.\text{t} < e_{b2}.\text{t} \wedge e_{b1}.\text{NF} = \text{NF}_b \wedge e_{b1}.\text{pkt}_{\text{in}}.\mathit{srcNF} = NF_a \wedge \mathit{flow}(e_{b1}) = \mathit{flow}(e_{a1})$; *i.e.,* all previous events generated by NF_b that came from NF_a and are part of the same flow as e_{a1} (and hence e_{a2} and e_{b2}). The crucial observation is that each such e_{b1} increments $FC_b[e_{b1}.\text{pkt}_{\text{in}}.\mathit{flowID},\ e_{b1}.\text{pkt}_{\text{in}}.\mathit{srcNF}]$, and these are the *only* events that do so prior to $e_{b2}.\text{t}$. Hence, it must be the case that $|E_{b1}| = e_{b2}.\text{pkt}_{\text{in}}.\mathit{seq}$, and each $e_{b1} \in E_{b1}$ has a unique sequence number (as guaranteed by the monotonically increasing counter value). Since we know that $0 \leq e_{a1}.\text{pkt}_{\text{out}}.\mathit{seqNum} < e_{a2}.\text{pkt}_{\text{out}}.\mathit{seqNum} = e_{b2}.\text{pkt}_{\text{in}}.\mathit{seqNum}$, there must be an $e_{b1} \in E_{b1}$ with $e_{b1}.\text{pkt}_{\text{in}}.\mathit{seqNum} = e_{a1}.\text{pkt}_{\text{out}}.\mathit{seqNum}$. Furthermore, the monotonic counter on the sending side (NF_a) guarantees that NF_a must have only assigned the sequence number $e_{a1}.\text{pkt}_{\text{out}}.\mathit{seqNum}$ to a single packet, namely $e_{a1}.\text{pkt}_{\text{in}}$. Since we have proven Property 1 holds (*i.e.,* the attacker cannot inject or modify packets), if NF_b received a packet $e_{b1}.\text{pkt}_{\text{in}}$ with sequence number $e_{a1}.\text{pkt}_{\text{out}}.\mathit{seqNum}$ from NF_a, it must be the case that $e_{a1}.\text{pkt}_{\text{out}} = e_{b1}.\text{pkt}_{\text{in}}$. Hence, we have identified an e_{b1} such that $e_{b1}.\text{t} < e_{b2}.\text{t} \wedge e_{b1}.\text{NF} = \text{NF}_b \wedge e_{b1}.\text{pkt}_{\text{in}} = e_{a1}.\text{pkt}_{\text{out}}$, proving that Property 2 holds.

Proof of Property 3 To show that Property 3 (no packet reordering) holds, choose an arbitrary $\text{NF}_a, \text{NF}_b \in F$ and $e_{a1}, e_{b1}, e_{a2}, e_{b2} \in E$ such that Lines 14-17 of Property 3 hold. Suppose for the sake of contradiction that $e_{a1}.\text{t} < e_{a2}.\text{t}$ but $e_{b1}.\text{t} \geq e_{b2}.\text{t}$ (*i.e.,* NF_b received the packets in reverse order). Since each NF increments its flow counter after sending a packet (Lines 10 and 27 in Algorithm 6), and we know that $\text{flow}(e_{a1}) = \text{flow}(e_{a2})$, it must be the case that $e_{a1}.\text{pkt}_{\text{in}}.\mathit{seqNum} < e_{a2}.\text{pkt}_{\text{in}}.\mathit{seqNum}$. Since we supposed $e_{b1}.\text{t} \geq e_{b2}.\text{t}$ (and $e_{b1}.\text{t} \neq e_{b2}.\text{t}$ because each entry in E has a unique position) it must be the case that $\textsc{Process}_b$ was called on $e_{a2}.\text{pkt}_{\text{out}}$ before it was called on $e_{a1}.\text{pkt}_{\text{out}}$. For $e_{a2}.\text{pkt}_{\text{out}}$ to have passed the "ok" check in $\textsc{Process}_b$, it must have passed the check on Line 18, which means that at the time, $e_{a2}.\text{pkt}_{\text{out}}.\mathit{seqNum} = FC_b[e_{a2}.\text{pkt}_{\text{out}}.\mathit{flowID},\ e_{a2}.\text{pkt}_{\text{out}}.\mathit{srcNF}]$. The counter is then incremented on Line 20, and continues to increase monotonically on subsequent invocations. Hence, when $\textsc{Process}_b$ was later called on $e_{a1}.\text{pkt}_{\text{out}}$, to have passed the "ok" check, it must be the case that $e_{a1}.\text{pkt}_{\text{out}}.\mathit{seqNum} > e_{a2}.\text{pkt}_{\text{out}}.\mathit{seqNum}$. This contradicts our starting point that $e_{a1}.\text{pkt}_{\text{out}}.\mathit{seqNum} < e_{a2}.\text{pkt}_{\text{out}}.\mathit{seqNum}$. Hence we can conclude that if $e_{a1}.\text{t} < e_{a2}.\text{t}$ then $e_{b1}.\text{t} < e_{b2}.\text{t}$.

To prove the other direction of the implication, namely that if $e_{a1}.\text{t} \geq e_{a2}.\text{t}$ then $e_{b1}.\text{t} \geq e_{b2}.\text{t}$. we can apply the same argument as above, swapping $a2$ for $a1$ and $b2$ for $b1$.

Proof of Property 4 To show that Property 4 (no packet replay) holds, choose an arbitrary $e_a \in E$. Suppose for the sake of contradiction that $\exists e_b \in E$ such that $e_a.\text{pkt}_{\text{in}} = e_b.\text{pkt}_{\text{in}}$. Since each entry in E has a unique position, we know $e_a.\text{t} \neq e_b.\text{t}$, so without loss of generality, assume $e_a.\text{t} < e_b.\text{t}$. Since we know that $e_a.\text{pkt}_{\text{in}}.\mathit{dstNF} = e_b.\text{pkt}_{in}.\mathit{dstNF}$, both entries must have been produced by invocations of $\textsc{Process}_{e_a.\text{pkt}_{\text{in}}.\mathit{dstNF}}$, and it was invoked on $e_a.\text{pkt}_{\text{in}}$ before

e_b.pkt$_{\text{in}}$ (because e_a.t$<e_b$.t). Since PROCESS did not raise an alert on e_a.pkt$_{\text{in}}$, it must be the case that it passed the check on Line 18, which means that at the time, e_a.pkt$_{\text{in}}$.$seqNum=FC_b[e_a$.pkt$_{\text{in}}$.$flowID$, e_a.pkt$_{\text{in}}$.$srcNF]$. The counter is then incremented on Line 20, and continues to increase monotonically on subsequent invocations. Hence, when PROCESS was later called on $e_b.pkt$, to have passed the "ok" check, it must be the case that e_b.pkt$_{\text{in}}$.$seqNum > e_a$.pkt$_{\text{in}}$.$seqNum$. But we supposed that e_a.pkt$_{\text{in}}$ = e_b.pkt$_{\text{in}}$, which means e_b.pkt$_{\text{in}}$.$seqNum = e_a$.pkt$_{\text{in}}$.$seqNum$. This contradiction show that e_b.pkt$_{\text{in}}$.$seqNum \neq e_a$.pkt$_{\text{in}}$.$seqNum$. ■

D.5 Security of Secret Logging

As described in §6.1, AuditBox implements secret logging via a *virtual* bit that is appended to the real data carried in the AuditTrailer when computing a MAC. For this approach to effectively sample packets even in the presence of an adversary, it should be computationally difficult to distinguish packets with the virtual bit set to one (indicating a packet that should be logged) from those with the virtual bit set to zero.

Security Definition We capture this security notion with the following game involving a challenger $\mathcal{C}$, and a probabilistic, polynomial-time adversary $\mathcal{A}$. When the game begins, $\mathcal{C}$ chooses a random bit b and then runs KeyGen() to produce key K. The adversary is then allowed to run and given access to Oracle-MAC($\cdot$), which when given a message m, returns MAC$_K(m|b)$. $\mathcal{A}$ eventually terminates and outputs its guess b'. The game returns 1 if $b{=}{=}b'$ and 0 otherwise.

We define $\mathcal{A}$'s advantage after making α queries to its oracle as $2\left|P-\frac{1}{2}\right|$, where P is the probability that the game returns 1.

Security Proof The game above is not secure for arbitrary MACs. In other words, given a EUF-CMA secure MAC scheme $\mathcal{M}$, we can construct a new scheme $\mathcal{N}$ that is also EUF-CMA secure, but where an adversary can achieve advantage 1 in the game above. For example, we could define $\mathcal{N}$.MAC$_K(m) = \mathcal{M}$.MAC$_K(m)|m$. This is EUF-CMA secure (since the adversary still cannot produce a forgery against $\mathcal{M}$), but message secrecy is entirely broken.

Fortunately, we *can* prove security for specific MAC algorithms, including HMAC and, crucially for our implementation, GMAC. In particular, we leverage the fact that

GMAC is defined (simplifying slightly) as:

$$\text{GMAC}_K(IV,m) = \text{PRF}_K(IV) \oplus \text{GHASH}(m)$$

Theorem 3 (Secret Logging Security) *Consider the game described above with adversary $\mathcal{A}$. When the MAC algorithm is GMAC, the adversary's advantage is negligible.*

Proof of Theorem 3: We prove security via the following two games.
Game 0 is defined as described above with an adversary $\mathcal{A}$ which queries its oracle a total of α times.
Game 1 is the same as above, except that Oracle-MAC(m) returns

$$\text{GMAC}_K(IV,m) = R \oplus \text{GHASH}(m)$$

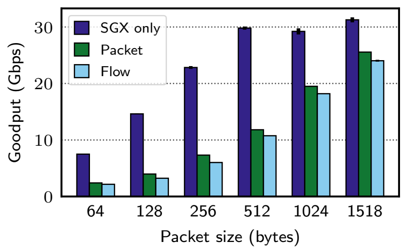
where R is a randomly sampled value.

The adversary can distinguish Game 1 from Game 0 only if it can distinguish the output of the PRF from the output of a truly randomly selected function. This happens with probability at most $\texttt{PRF}(\alpha)$, the probability of breaking the security of GMAC's PRF given at most α queries.

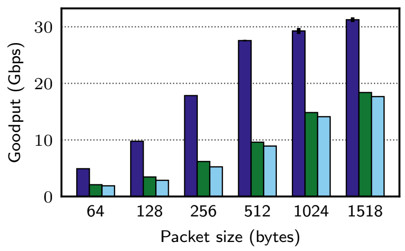
From Game 1, we show that the adversary has no information about the underlying message (and hence about the value of b), which means that the adversary's advantage in the security game is at most $\texttt{PRF}(\alpha)$, which is, by definition, negligible when we employ a secure PRF, which is also necessary for the standard EUF-CMA security of GMAC to hold.

In Game 1, the output of GHASH (which is the only information derived from m) is XOR'ed with a randomly chosen value of the same length. In other words, the output is the result of applying a one-time pad to GHASH(m), which is an informationally secure encryption scheme. Hence the adversary learns nothing about m from the output of its oracle. ■

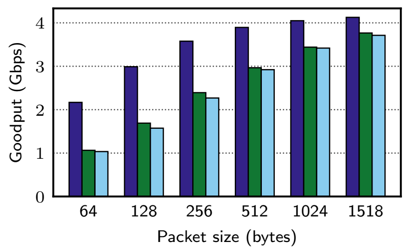
E Performance Sensitivity Analysis



(a) Firewall.



(b) NAT.



(c) DPI.

Figure 16: Sensitivity analysis across NFs for different packet sizes.

Contracting Wide-area Network Topologies to Solve Flow Problems Quickly

Firas Abuzaid[†*], *Srikanth Kandula*[†], *Behnaz Arzani*[†], *Ishai Menache*[†], *Matei Zaharia*[*], *Peter Bailis*[*]
Microsoft Research[†] *and Stanford University*[*]

Abstract– Many enterprises today manage traffic on their wide-area networks using software-defined traffic engineering schemes, which scale poorly with network size; the solver runtimes and number of forwarding entries needed at switches increase to untenable levels. We describe a novel method, which, instead of solving a multi-commodity flow problem on the network, solves (1) a simpler problem on a contraction of the network, and (2) a set of sub-problems in parallel on disjoint clusters within the network. Our results on the topology and demands from a large enterprise, as well as on publicly available topologies, show that, in the median case, our method nearly matches the solution quality of currently deployed solutions, but is 8× faster and requires 6× fewer FIB entries. We also show the value-add from using a faster solver to track changing demands and to react to faults.

1 Introduction

Wide-area networks (WANs), which connect locations across the globe with high-capacity optical fiber, are an expensive resource [7, 35, 36, 38]. Hence, enterprises seek to carefully manage the traffic on their WANs to offer low latency and jitter for customer-facing applications [28, 62, 69] and fast response times for bulk data transfers [46, 56].

The state-of-the-art approach used in several enterprises today [35, 36, 38] is to compute optimal routing schemes for the current demand by solving global multi-commodity flow problems [7,35,36,38]; the global flow problems are re-solved periodically, since demands may change or links may fail, and the computed routes are encoded into switch forwarding tables using software-defined networking techniques [7].

As network sizes grow, solving multi-commodity flow problems on the entire network becomes practically intractable. As noted in [36], the "algorithm run time increased super-linearly with the site count," which led to "extended periods of traffic blackholing during data plane failures, ultimately violating our availability targets," as well as "scaling pressure on limited space in switch forwarding tables." This problem is unlikely to go away: anecdotal reports indicate that WAN

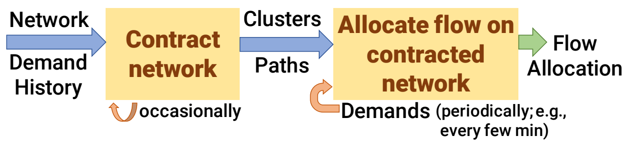

Network
Demand History
Contract network
occasionally
Clusters
Paths
Allocate flow on contracted network
Demands (periodically; e.g., every few min)
Flow Allocation

Figure 1: NCFlow's workflow.

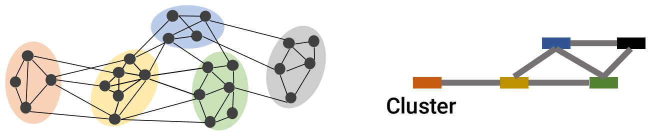


Figure 2: The original network on the left is divided into clusters, shown with different background colors. The contracted network is on the right.

footprints today are already over 10× larger than the few tens of sites that were considered in prior work [35, 36], since enterprises have built more sites to move closer to users.

In this paper, we seek to retain the benefits of global traffic management for large WAN networks without requiring excessively many forwarding entries at switches or prohibitively long solver runtimes. Also, by using a faster solver, WAN operators can reduce loss when faults occur and carry more traffic on the network by tracking demand changes.

Our solution is motivated by the observation that WAN topologies and demands are *concentrated*: the topology typically has well-connected portions separated by a few, lower-capacity edges, and more demand is between nearby datacenters. This is likely due to multiple operational considerations: (1) submarine cables have become shared choke points for connectivity between continents (see Figure 3), (2) the connectivity over land follows the road or rail networks along which fiber is typically laid out, and (3) enterprises build datacenters close to users, then steer traffic to nearby datacenters [12, 62, 69]. Therefore, more capacity and demand are available between *nearby* nodes; an analysis of data from a large enterprise WAN in §2 supports this observation.

We leverage this concentration of capacity and demand to decompose the global flow problem into several smaller problems, many of which can be solved in parallel. As shown

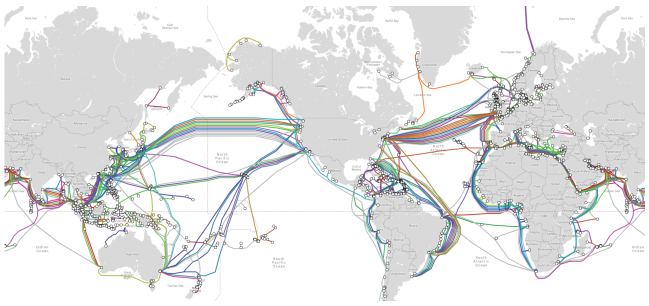


Figure 3: Submarine cables serve as choke points in WAN topologies; figure is excerpted from [63].

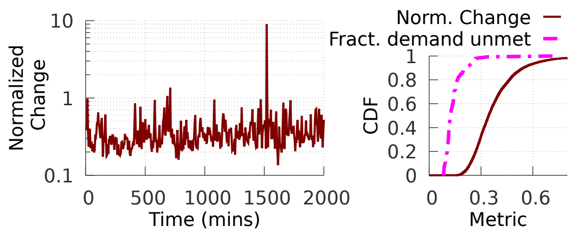


Figure 4: On the left, we plot the L2 norm of the change in the demands between successive 5-minute periods divided by the L2 norm of the traffic matrix at a time. On the right, we show the CDF of this change ratio. We also show a CDF of the fraction of demand that is unsatisfied if using the allocation computed for the previous period.

in Figure 2, we divide the network into multiple connected components, which we refer to as *clusters*. We then solve modified flow problems on each cluster, as well as on the *contracted network*, where nodes are clusters and edges connect clusters that have connected nodes. Prior work [4, 9, 15] notes that Google and other map providers use different contractions to compute shortest paths on road network graphs. Our goal is to closely match the multi-commodity max flow solution in quality (i.e., carry nearly as much total flow), while reducing the solver runtime and number of required forwarding entries. We discuss related work in §7; to our knowledge, we are the first to demonstrate a practical technique for multi-commodity flow problems on large WAN topologies.

Solving flow problems on the contracted network poses two key challenges:

1. How to partition the network into clusters? More clusters leads to greater parallelism, but maximizing the inter-cluster flow requires careful coordination between the sub-problems at multiple clusters.
2. How to design the sub-problems for each cluster to improve speed while reducing inconsistencies in allocation? The sub-problem for a cluster has fewer nodes and edges to consider, but it will not be be faster if it must consider all node pairs whose traffic can pass through the cluster.

Our solution NCFlow[1] achieves a high-quality flow allocation with a low runtime and space complexity by addressing each of these challenges in turn. First, we contract the network using well-studied algorithms such as modularity-based clustering [25] and spectral clustering [53], which are designed to identify the choke-point edges in a network. Second, we *bundle* demands whose sources and/or targets are in the same cluster, treating them as a single demand. In Figure 2 for example, the yellow cluster considers as one bundled demand all traffic from source nodes in the red cluster to target nodes in the green cluster. Doing so can lead to inconsistent flow allocations between clusters (which we explain in §3.1.1) and we devise careful heuristics to provably avoid them (§3.2). Finally, we reduce the forwarding entries needed at switches by reusing pathlets within clusters and traffic splitting rules across multiple demands (§3.5).

Figure 1 shows the workflow for NCFlow. First, we choose appropriate clusters and paths using an offline procedure over historical traffic—these choices are pushed into the switch forwarding entries. This step happens infrequently, such as when the topology and/or traffic changes substantially. Then, online (e.g., once every few minutes), NCFlow computes how best to route the traffic over the clusters and paths, similar to deployed solutions [35, 36, 38].

Overall, our key contributions are:

- We propose NCFlow, a decomposition of the multi-commodity max flow problem into an offline clustering step and an online, provably feasible, algorithm that solves a set of smaller sub-problems in parallel.
- We evaluate NCFlow using real traffic on a large enterprise WAN, as well as synthetic traffic on eleven topologies from the Internet Topology Zoo [6]. Our results show that, for multi-commodity max flow, NCFlow is within 2% of the total flow allocated by state-of-the-art path-based LP solvers [35, 36, 38] in 50% of cases; NCFlow is within 20% in 97% of cases. Furthermore, NCFlow is at least $8\times$ faster than path-based LP solvers in the median case; in 20% of cases, NCFlow is over $30\times$ faster. Lastly, NCFlow requires 2.7–16.7$\times$ fewer forwarding entries in the evaluated topologies. NCFlow also compares favorably to state-of-the-art approximation algorithms [27,41] and oblivious techniques [44,57].
- We show that, as a fast approximate solver, NCFlow can be used to react quickly to demand changes and link failures. Specifically, in comparison to TEAVAR [19], NCFlow carries more flow when no faults occur and suffers about the same amount of total loss during failures.

We have open-sourced an anonymous version of NCFlow [2], and are in the early stages of integrating NCFlow into production use at a large enterprise.

2 Background and Motivation

We analyze the changes in topology and traffic on a large enterprise WAN over a several-month period. As Figure 4

[1] short for **N**etwork **C**ontractions for **Flow** problems

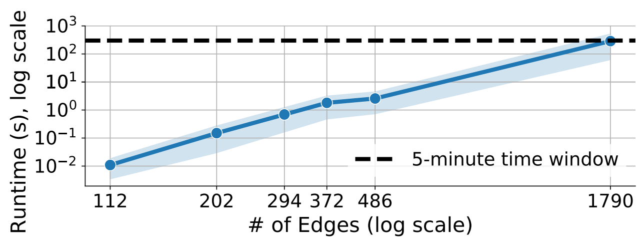


Figure 5: Runtimes of a state-of-the-art solver on topologies from Internet Topology Zoo [6]. Both axes are in log scale and the band represents standard deviation. In production WANs, new traffic demands arrive every few minutes [35, 38].

shows, the change in traffic demand from one 5-minute window to the next is substantial; the average change is 35%; in 20% of the cases, the traffic change is over 45%. The enterprise solves a global flow allocation problem every few minutes. The figure on the right shows the fraction of traffic that will remain unsatisfied if the flow allocation from the previous window were to be used instead of computing a new allocation. We see that the median loss is 13%; in 20% of the cases, over 20% of the demand remains unsatisfied. We verify that computing a new allocation will satisfy all of the demand; using the previous window's allocation causes loss because some datacenter pairs may receive more flow in the previous allocation than their current demand while other datacenter pairs go unsatisfied.

Given the above data, computing a new allocation in each time window is needed to carry more traffic on the WAN. However, solver runtime increases super-linearly with the size of the topology, as shown in Figure 5. For several public topologies and on a variety of traffic matrices, we benchmark the multi-commodity max-flow problem (specifically PF_4, as will be described in §5.1). The runtimes were measured on a server-grade machine using a production-grade optimization library [33]. As the figure shows, when the topology size exceeds a thousand edges, the time to compute a flow allocation can exceed the allotted time window.

A fast solver would not only ensure that new allocations complete in time—it could also enable more frequent allocations, e.g., every minute. Doing so would enable allocations to track changing demands at a finer granularity. Moreover, as we show in §5, a fast solver can help when reacting to link and switch failures.

Our observation that demand and capacity are concentrated among *nearby* nodes is grounded on the following measurements from a production WAN:

Demand properties:

- On average, 7% (or 16%) of the node pairs account for half (or 75%) of the total demand.
- When nodes are divided into a few tens of clusters, 47% of the total traffic stays within clusters. If the demands were distributed uniformly across node pairs, only 8% of the traffic would stay within clusters; thus the demand within clusters is about 6× larger than would be expected from a uniform distribution.

WAN topology properties:

- When nodes are divided into tens of clusters, 76% of all edges and 87% of total capacity is within clusters.
- The skew in capacity is small: the ratio between the largest edge capacity and the mean is 10.4.
- The skew in node degree is also small: the average node degree is 3.9, with $\sigma = 2.6$; the max is 16.
- Relative to the network size (hundreds of nodes), the average network diameter (=11) and the average shortest-path length (= 5.3) are very small.

Motivated by the above analyses, NCFlow seeks to be a fast solver for large WAN topologies by leveraging the concentration of traffic demands and capacity.

Background: Before we describe NCFlow's design, we give some background on multi-commodity flow problems. Given a set of nodes, capacitated edges, and demands between nodes, a flow allocation is feasible if it satisfies demand and capacity constraints. The goal of a multi-commodity flow problem is to find a feasible flow which optimizes a given objective; Table 1 lists some common flavors.

The fastest algorithms [27, 41] are approximate; i.e., given a parameter ε, they achieve at least $(1-\varepsilon)\times$ the optimal value. And, their runtime complexity is at least quadratic (Table 1).

Moreover, these solutions allow demands to travel on any edge, thus requiring millions of forwarding table entries at each switch for thousand-node topologies. Instead, production systems [35, 38] restrict flow to a small number of pre-configured paths per demand, which reduces the required forwarding table entries by 10–100×.

Using notation from Table 2, the feasible flow over a pre-configured set of paths can be defined as:

$$
\begin{aligned}
&\mathsf{FeasibleFlow}(\mathcal{V},\mathcal{E},\mathcal{D},\mathcal{P}) \triangleq \{f_k \mid \forall k \in \mathcal{D} \text{ and} && (1)\\
&\qquad f_k = \sum_{p \in \mathcal{P}_k} f_k^p, && \forall k \in \mathcal{D} \quad \text{(flow for demand } k\text{)}\\
&\qquad f_k \le d_k, && \forall k \in \mathcal{D} \quad \text{(flow below volume)}\\
&\qquad \sum_{\forall k, p \in \mathcal{P}_k, e \in p} f_k^p \le c_e, && \forall e \in \mathcal{E} \quad \text{(flow below capacity)}\\
&\qquad f_k^p \ge 0 && \forall p \in \mathcal{P}, k \in \mathcal{D} \quad \text{(non-negative flow)} \}
\end{aligned}
$$

Production systems use linear optimization-based solvers [35, 36, 38]. On WANs with thousands of nodes, the optimization problem could have millions of variables and equations just to verify that a flow allocation is feasible.

In this paper, we consider the problem of maximizing the total flow across all demands:

$$
\begin{aligned}
&\mathsf{MaxFlow}(\mathcal{V},\mathcal{E},\mathcal{D},\mathcal{P}) \triangleq \arg\max_{\mathbf{f}} \sum_{k \in \mathcal{D}} f_k && (2)\\
&\qquad \text{s.t.} \quad \mathbf{f} \in \mathsf{FeasibleFlow}(\mathcal{V},\mathcal{E},\mathcal{D},\mathcal{P})
\end{aligned}
$$

	Maximization term	**Additional Constraints**	**Used in**	**Known best complexity**
MaxFlow	$\sum_{k\in\mathcal{D}} f_k$	none	[35, 38]	$O(M^2\epsilon^{-2}\log^{O(1)} M)$ [27]
MaxFlow with Cost Budget	$\sum_{k\in\mathcal{D}} f_k$	$\sum_k \sum_{p\in\mathcal{P}_k} \sum_{e\in p} f_k^p \mathsf{Cost}_e \le \mathsf{Budget}$		$O(\epsilon^{-2} M \log M(M + N\log N)\log^{O(1)} M)$ [27]
Max Concurrent Flow	α	$d_k\alpha \le f_k, \forall k \in \mathcal{D}$	[19, 39, 40]	$O(\epsilon^{-2}(M^2 + KN)\log^{O(1)} M)$ [41]

Table 1: We illustrate a few different multi-commodity flow problems all of which find feasible flows but optimize for different objectives and can have additional constraints; see notation in Table 2. Equation 6 fleshes out the problem completely for the case of maximizing flow. More problems are discussed in [11].

Term	**Meaning**
$\mathcal{V}, \mathcal{E}, \mathcal{D}, \mathcal{P}$	Sets of nodes, edges, demands, and paths
N, M, K	The numbers of nodes, edges, and demands, i.e., $N = \lvert\mathcal{V}\rvert, M = \lvert\mathcal{E}\rvert, K = \lvert\mathcal{D}\rvert$
e, c_e, p	Edge e has capacity c_e; path p is a set of connected edges
(s_k, t_k, d_k)	Each demand k in $\mathcal{D}$ has source and target nodes ($s_k, t_k \in \mathcal{V}$) and a non-negative volume (d_k).
$\mathbf{f}, f_k^p$	Flow assignment vector for a set of demands and the flow for demand k on path p.

Table 2: Notation for framing multi-commodity flow problems.

$\mathcal{V}_{\text{agg}}$, $\mathcal{E}_{\text{agg}}$, $\mathcal{D}_{\text{agg}}$, $\mathcal{P}_{\text{agg}}$	Nodes, edges, demands, and paths in the aggregated graph
$\mathcal{V}_x, \mathcal{E}_x, \mathcal{D}_x, \mathcal{P}_x$	Subscript denotes entities in the restricted graph for cluster x
x, η	Each cluster x is a strongly connected set of nodes and η is the number of clusters
$k, K_{xy}, K_{sy}, K_{xt}$	An actual demand (k); the rest are bundled demands from one source (s) or all nodes in a cluster (x) to a target (t) or to all nodes in a cluster (y)

Table 3: Additional notation specific to NCFlow.

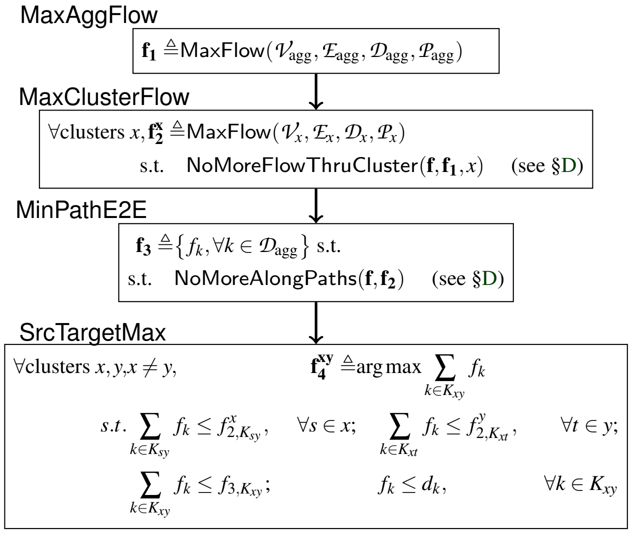

MaxAggFlow
$\mathbf{f_1} \triangleq \mathsf{MaxFlow}(\mathcal{V}_{\text{agg}}, \mathcal{E}_{\text{agg}}, \mathcal{D}_{\text{agg}}, \mathcal{P}_{\text{agg}})$
MaxClusterFlow
$\forall$clusters $x, \mathbf{f_2^x} \triangleq \mathsf{MaxFlow}(\mathcal{V}_x, \mathcal{E}_x, \mathcal{D}_x, \mathcal{P}_x)$
s.t. $\mathsf{NoMoreFlowThruCluster}(\mathbf{f}, \mathbf{f_1}, x)$ (see §D)
MinPathE2E
$\mathbf{f_3} \triangleq \{f_k, \forall k \in \mathcal{D}_{\text{agg}}\}$ s.t.
s.t. $\mathsf{NoMoreAlongPaths}(\mathbf{f}, \mathbf{f_2})$ (see §D)
SrcTargetMax
$\forall$clusters $x, y, x \neq y$, $\mathbf{f_4^{xy}} \triangleq \arg\max \sum_{k\in K_{xy}} f_k$
$s.t. \sum_{k\in K_{sy}} f_k \le f^x_{2,K_{sy}}, \quad \forall s \in x; \quad \sum_{k\in K_{xt}} f_k \le f^y_{2,K_{xt}}, \quad \forall t \in y;$
$\sum_{k\in K_{xy}} f_k \le f_{3,K_{xy}}; \quad f_k \le d_k, \quad \forall k \in K_{xy}$

Figure 6: The basic flow allocation algorithm used by NCFlow; notation used here is defined in Table 3.

SDN-based traffic engineering schemes [35, 38], in addition to repeatedly solving global optimizations, must maintain an up-to-date view of the topology, gather desired volumes for demands and update traffic splits at switches based on the result of the optimization. Our production experience is that most of these repetitive steps have a latency of a few RTTs (round trip times) and so solving the optimization dominates, especially on large topologies. Moreover, demands are limited to their allocated rates in software at the source servers and thus allocating less than the full desired rate need not result in packet loss [35]. Finally, applications that contribute a large fraction of the bytes moving between datacenters are elastic in short timescales; e.g., large dataset transfers for data analytics. That is, these apps seek a fast completion time but do not need a large rate in every optimization epoch. Some other applications have a decreasing marginal utility as their rate allocation increases such as video streams of varying quality [43]. Today's SDN-based TE solutions [35, 38] use multiple priority classes to maximize allocations for elastic traffic without affecting the latency-sensitive traffic.

3 NCFlow

In this section, we describe NCFlow. Our steps are as shown in Figure 1. Offline, based on historical demands, we divide the network into clusters and determine paths. Further details are in §3.4. Online, we allocate flow to the current demands by solving a carefully constructed set of simpler sub-problems, some of which can be solved independently and in parallel. We describe these sub-problems in §3.1. Although they can be solved quickly, disagreements between independent solutions can lead to infeasible allocations; we present a simple heuristic in §3.2 that provably leads to feasible flow allocations. In §3.3, we discuss extensions that increase the total flow allocated by NCFlow. We also show sufficient conditions under which NCFlow is optimal and matches the flow allocated by MaxFlow. Finally, in §3.5, we discuss how NCFlow uses fewer forwarding entries by reusing pathlets within clusters and splitting rules for different demands.

3.1 Basic Flow Allocation

We begin by describing a simple (but incomplete) version of NCFlow's flow allocation algorithm; the pseudocode is in Figure 6. We continue using Figure 2 as a running example. The basic algorithm proceeds in four steps.

In the first step, we allocate flow on the aggregated graph; as shown in MaxAggFlow in Figure 6. In the aggregated graph, an example of which is in Figure 2 (right), nodes are clusters and the edges are bundled edges from the original graph—the edge between the red and yellow clusters corresponds to the five edges between these clusters on the actual graph. Similarly, we bundle demands on the aggregated graph: the demand K_{xy} between the clusters x and y corresponds to all of the demands whose sources are in cluster x and targets are

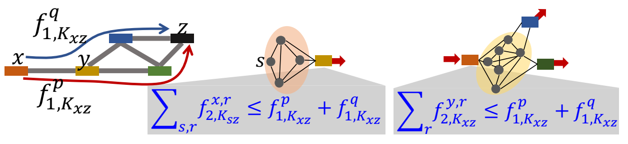

$f^q_{1,K_{xz}}$
z
x
y
$f^p_{1,K_{xz}}$
s
$\sum_{s,r} f^{x,r}_{2,K_{sz}} \le f^p_{1,K_{xz}} + f^q_{1,K_{xz}}$
$\sum_{r} f^{y,r}_{2,K_{xz}} \le f^p_{1,K_{xz}} + f^q_{1,K_{xz}}$

Figure 7: An example illustrating how the flow allocated in MaxAggFlow translates to constraints on the flow to be allocated in MaxClusterFlow.

in cluster y. The resulting flow allocation ($\mathbf{f_1}$) accounts for bottlenecks on the edges between clusters. However, this flow may not be feasible, since there may be bottlenecks *within* the clusters.

In the second step, we refine the allocation from step 1 to account for intra-cluster demands and constraints. Specifically, we allocate flow for the demands whose sources and targets are within the cluster. We also allocate no more flow than was allocated in $\mathbf{f_1}$ for the inter-cluster flows. MaxClusterFlow in Figure 6 shows code for this step. We note a few details:

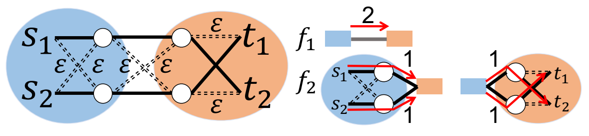
- We use virtual nodes to act as the sources and targets for the inter-cluster flows; the flow allocated in $\mathbf{f_1}$ determines which virtual node (i.e., which neighboring cluster) is the sender or the receiver for an inter-cluster demand.
- Figure 7 shows two examples on the right where the virtual nodes are drawn using squares.
- Figure 7 also shows the NoMoreFlowThruCluster constraints for demands from sources in the red cluster to targets in the black cluster (depicted as x and z respectively). On the aggregated graph, the flow for this demand takes the two paths shown. In the red cluster, as shown in the equation, the traffic from all sources (s), along multiple paths (r) to the virtual node, is restricted to be no more than what was allocated in $\mathbf{f_1}$.
- Figure 7 on the right also shows a more complex case that happens in the yellow cluster. Here, the traffic arrives at one virtual node but can leave to multiple virtual nodes. In MaxClusterFlow, we set up paths between all pairs of virtual nodes. As shown in the equation, the traffic leaving the red virtual node on paths (r) to either of the other virtual nodes must be no more than the total flow on paths p and q from $\mathbf{f_1}$.
- Observe that bundling demands ensures fewer variables and constraints for MaxClusterFlow. The demand from red to black clusters comprises twenty node pairs in the actual graph in Figure 2 (left); four sources in the red cluster and five targets in the black cluster. However, the MaxClusterFlow for the red cluster only has four bundled demands, from each source to the virtual node, and the yellow cluster has just one bundled demand from and to virtual nodes.

In the third step, we reconcile end-to-end; that is, we find the largest flow that can be carried along each path on the aggregate graph. As shown by MinPathE2E in Figure 6, for each bundle of demands and each path, we take the minimum flow allocated ($\mathbf{f_2^x}$) at each cluster on the path.

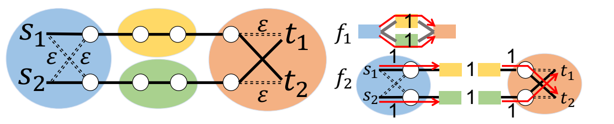
The flow allocation for the demands in a cluster x can be read directly from the $\mathbf{f_2^x}$ solution of MaxClusterFlow. For demands that span clusters, however, more work remains because the steps thus far do not directly compute their flow. In particular, $\mathbf{f_3}$ allocates flow for cluster bundles; such as say for all the demands whose sources are in cluster x and targets are in cluster y. The corresponding per-cluster flow allocations, $\mathbf{f_2^x}$ and $\mathbf{f_2^y}$, allocate flow from a source node and to a given target respectively. Thus, in the final step, SrcTargetMax, we assign the maximal flow to each inter-cluster demand that respects all previous allocations.

Problem	# of Nodes	# of Edges	# of Demands
MaxFlow	N	M	K
MaxAggFlow	η	$\le \min(M, \eta^2)$	$\le \min(K, \eta^2)$
MaxClusterFlow	$\sim \frac{N}{\eta} + \eta$	$\sim \frac{M}{\eta} + 2\eta$	$\sim \frac{K}{\eta^2} + 2\frac{N}{\eta} + \eta^2$

Table 4: Sizes of the problems in Figure 6 using notation from Tables 2 and 3. Just verifying that flow is feasible (i.e., FeasibleFlow in Eq. 1) uses $O(\#\text{ nodes} * \#\text{ edges})$ number of equations and variables. NCFlow has one instance of MaxAggFlow and executes the η instances of MaxClusterFlow in parallel. MinPathE2E and SrcTargetMax, are relatively insignificant.



(a) Disagreement arising from bundling edges: As shown on the right, the algorithm in Figure 6 will allocate 2 units of flow but only 5ε units can be carried.



(b) Disagreement arising from bundling demands: As shown on the right, the algorithm in Figure 6 will allocate 2 units of flow, but only 2ε units can be carried.

Figure 8: Illustrating how disagreements in flow allocation can occur in the basic flow allocation algorithm; see §3.1.1.

3.1.1 Properties of Basic Flow Allocation

Solver runtime: The numbers of equations and variables in the sub-problems are shown in Table 4. If the number of clusters η is 1, note that there is exactly one per-cluster problem, MaxClusterFlow, which matches the original problem from Eqn. 2. When using a few tens of clusters, we will show in §5 that all of the sub-problems are substantially smaller than the original problem (MaxFlow).

Feasibility: The flow allocated by Figure 6 satisfies demand and capacity constraints; we will prove this formally in §B.1. For demands whose source and target are in different clusters, however, disagreements may ensue since the different problem instances assign flow to different bundles of edges and demands. We illustrate two such examples in Figure 8; both have 1 unit of demand from s_1 to t_1 and from s_2 to t_2. The dashed edges have a capacity of $\varepsilon \ll 1$ and all of the other edges have a very large capacity.

- The example in Figure 8a illustrates an issue with bundling edges. The actual graph on the left can only

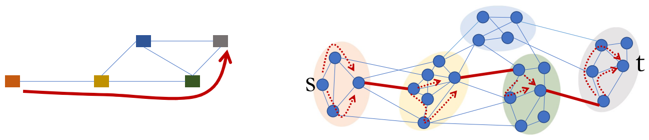


Figure 9: To guarantee feasibility, each cluster bundle is allocated flow on only one path on the aggregated graph (left) and on only one edge between each pair of clusters (right); the usable path and edges are shown in dark red. Note that multiple paths can still be used within clusters.

carry 5ε units of flow for each demand. However, as the figures on the right show, MaxAggFlow allocates two units of flow since the four edges between these two clusters can together carry all of the two units of demand. The MaxClusterFlow instances also allocate two units of flow as shown. The discrepancy arises because the problems in Figure 6 do not know that the *top* egress of the left cluster can take in all of the demand of s_1 but has only a low capacity to t_1.

- The example in Figure 8b illustrates an issue with bundling demands. Here too, observing the actual network on the left will show that 2ε units can be carried for each demand split evenly between the top and the bottom path. Again, as the figures on the right show, the basic flow allocation algorithm will conclude that both units of demand can be carried. Here, the discrepancy arises from the bundling of demands, the problems in Figure 6 cannot discern that the MaxClusterFlow instance of the left cluster sends the first demand to the brown cluster while the MaxClusterFlow of the right cluster wants to receive the second demand from the brown cluster.

3.2 A feasible heuristic

To avoid end-to-end disagreements, we make two simple changes to the basic flow allocation in §3.1.

First, when solving MaxAggFlow, only one path on the aggregated graph can be used for all of the demands between a given pair of clusters; we call such groups of demands to be cluster bundles. Next, between a pair of connected clusters, only one edge can carry the flow for a cluster bundle. Figure 9 shows in dark red an example path for a cluster bundle and the allowed edges between clusters; we also show the intra-cluster paths that can carry flow for this bundle.

There are multiple ways to avoid disagreements while keeping the problem sizes small via bundling. We discuss the above changes here because they are simple and sufficient. Specifically, we show that:

Theorem 1. *The algorithm in Figure 6, when constrained as discussed above, will always output a feasible flow.*

Proof. The proof is in §B.2. Intuitively, these changes suffice because the independent decisions made by different problems in Figure 6 cannot disagree; per cluster bundle, all problem instances allocate flow to the same edge and path. □

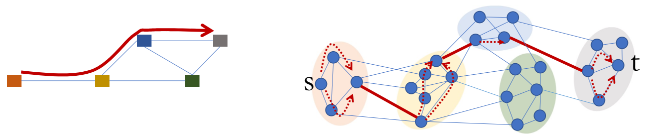


Figure 10: Contrasting with Figure 9, for the same cluster bundle, in a subsequent iteration, NCFlow allocates flow on a different path on the aggregate graph and on different inter-cluster edges. The chosen paths and edges are again shown in red.

3.3 Stepping towards optimality

The flow allocation algorithm described thus far is fast but not optimal; that is, it may allocate less total flow over all demands than the flow allocated by solving the larger global problem (MaxFlow from Eqn. 2). There are a few reasons why this happens. The MaxAggFlow in Figure 6 allocates flow on paths through clusters without knowing how much flow the clusters can carry. Switching the order, i.e., solving MaxClusterFlow before MaxAggFlow, could be worse because each cluster must allocate flow without knowing how much flow can be carried end-to-end. Furthermore, the heuristic in §3.2 constrains each cluster bundle to use only one edge between clusters and one path on the aggregated graph. We now discuss a few extensions to increase the flow allocation.

First, we re-solve the problems in Figure 6 multiple times. A simple way to do this would be to deduct the allocated flow and use the residual capacity on edges in the next iteration. Also, we pick different edges between clusters and/or different paths on the aggregated graph in different iterations (see Figure 10 for an example). The number of iterations is configurable; we continue as long as the total flow increases in each iteration by at least a pre-specified amount (say 5%). One could apply other policies such as a timeout. We show in §5 that a small number of iterations suffice for a sizable increase in the total flow. We will also show that later iterations finish faster than the first iteration perhaps because fewer demands remain to satisfy.

Next, we empirically observe that the choice of clusters and edges/paths to use in different iterations has an effect on flow allocation. For instance, the disagreements in Figure 8 *go away* by using a different choice of clusters—specifically, see Figure 31d and Figure 31e. We discuss how NCFlow precomputes cluster and edge/path choice in §3.4.

To sum up, we prove that flow allocation will be optimal when a few sufficient conditions hold:

Theorem 2. *The method in Figure 6 leads to the optimal flow allocation when any path can be used within each optimization and the number of clusters is* 1 *or equal to the number of nodes or all of the following conditions hold:*

- *the aggregated graph $\mathcal{G}_{agg}$ is a tree,*
- *only one edge connects any pair of clusters,*
- *all demands are satisfiable.*

Proof. By optimal, we mean that the total allocated flow must be as large as an instance of Equation 6 wherein any path can be used. The proof is in §B.3. Intuitively, when the number of clusters is 1 and any paths can be used, a single instance of MaxClusterFlow is identical to the optimal problem in Equation 6. Similarly, when the number of clusters equals the number of nodes, MaxAggFlow is identical to the optimal problem. Furthermore, the conditions listed lead to optimality because the optimal flow allocation can be transformed into an allocation that can be outputted by Figure 6. □

Even though the listed conditions appear restrictive, note that the topology within clusters can be arbitrary. We will show in §5 that NCFlow offers nearly optimal flow allocations even when the above conditions do not hold.

3.4 Choosing clusters and paths

The choice of clusters and paths affects both the solution quality and runtime of NCFlow. We cast cluster choice as a graph partitioning problem [5, 21, 65] with these objectives:

- **Concentrated with a low cut:** NCFlow can output better flow allocations when much of the total demand and the total edge capacity is between nodes in the same cluster.
- **Balanced cut:** Intuitively, NCFlow will have a smaller runtime when the complexity of MaxAggFlow balances with that of MaxClusterFlow. Recall from Table 4 that the former depends on the number of clusters whereas the latter depends on the size of the largest cluster.

We empirically observe, based on experiments with many WANs and different types of demands, that:

- On a graph with N nodes, about $\sqrt{N}$ clusters, irrespective of the clustering technique, leads to the best result, i.e., smallest runtime and fewest forwarding entries while allocating nearly the largest amount of flow possible; see Figure 13.
- When choosing the same number of clusters, one of the three considered clustering techniques (described below) generally performs better than the others but not in all cases; see Figure 21.

Thus, the *optimal* clustering choice for a WAN is unclear; it is possible that hand-tuning or using a learning technique may lead to better-performing clusters. Nevertheless, any of the three simple clustering schemes discussed below already suffice for NCFlow to improve substantially over baselines.

We consider the following clustering choices because they are simple and fast; unless otherwise noted, results in this paper use FMPartitioning.

- FMPartitioning [18, 25] divides nodes into clusters so as to maximize a "modularity" score which prefers more edges to lie within than between clusters. In NCFlow, we apply modularity-based clustering with edge weights set to their capacity.
- Spectral clustering [53] computes eigenvectors of the weighted adjacency matrix and chooses a desired number of the top eigenvectors as *cluster heads*; each node is assigned to the cluster of their closest eigenvector (e.g., using k-means).
- Leader Election picks a desired number of nodes at random as leaders and assigns each other node to the closest leader; wherein, distance is measured as the path length using invcap edge weights.

Some other clustering techniques [5, 42, 65] can balance cluster sizes or trade-off between concentration and balance but are more complex computationally; it is possible that using such schemes can further improve NCFlow.

Path choice in NCFlow: On the aggregated graph and on each cluster graph, we pre-compute offline a small number of paths between every pair of nodes. We consider the following different path choices and pick paths that lead to the largest flow allocation on historical demands:

- k-shortest paths [70] with edge weight of 1 or $\frac{1}{c_e}$ where c_e is the capacity of edge e and $k = 4, 8$ or 16.
- As above, but with the additional requirement that the paths for a node pair are edge-disjoint [52].

NCFlow also pre-computes offline (1) a pseudo-random choice of which edges to use between a pair of connected clusters in each iteration and (2) which path on the aggregated graph to use for each cluster bundled demand in each iteration.

3.5 Setting up switch forwarding entries

NCFlow uses many fewer switch forwarding entries than prior works due to the following reasons.

First, the paths along which NCFlow allocates flow can be thought of as a sequence of pathlets [32, 47, 68] in each cluster connected by crossing edges between clusters. Figures 9 and 10 illustrate such paths on the right. This observation is crucial because a pathlet can be reused by multiple demands. For example, in Figure 9, the flow from any source in the red cluster to any target in the grey cluster would use the same pathlets shown in the yellow, green, and blue clusters. Prior work [35, 36], on the other hand, establishes paths for each demand. Using pathlets has two advantages. The number of pathlets used by NCFlow is about η times less than the number of paths used by prior works[2]. Furthermore, a typical pathlet has fewer hops than a typical end-to-end path. Thus, NCFlow uses many fewer rules to encode paths in switches.

[2]More precisely, the number reduces from $PN(N-1)$ to $\sum_x P(N_x)(N_x-1)$ where P is the number of paths per node pair, the N nodes are divided into η clusters, and cluster x has N_x nodes. If clusters are evenly sized, $N_x = N/\eta$, and the ratio of these terms is $\sim \eta$.

Next, whenever NCFlow allocates flow at the granularity of cluster bundles, all of the demands in a bundle take the same paths and are split in the same way across paths. Hence, NCFlow uses one traffic splitting rule for all demands in such bundles. For instance, the demands from source s in the red cluster in Figure 9 to any target in the grey cluster are split with the same ratio across the same pathlets in all clusters (except the grey cluster where they take different pathlets to reach their different targets). Thus, with NCFlow, the number of splitting rules at a source decreases by a factor of $\sqrt{N}/2$[3].

The paths and splitting rules to push into switch forwarding tables are determined by the offline component of NCFlow and only change occasionally. After each allocation, only the splitting ratios change. More details on the data-plane of NCFlow such as how to compute the total flow that can be sent by each demand and the splitting ratios as well as how to move packets from one pathlet to the next are in Appendix C. In §5, we measure the numbers of rules used by NCFlow.

4 Implementing NCFlow

Our current prototype of NCFlow is about 5K lines of Python code, which invokes Gurobi [33] v8.1.1 to solve all of the optimization problems. For clustering WAN topologies, we adapt [26] to find clusters that maximize modularity; we also use our own implementation of NJW spectral clustering [53]. We use a grid search over the number of clusters (η) and the above clustering techniques to identify the best performing choice for each topology on a set of historical traffic matrices. To compare with state-of-the-art techniques, we customize the public implementations of SMORE [44, 45] and TEAVAR [19]. We have also implemented Fleischer's algorithm [27]; our implementation is about 10× faster than public implementations [8, 37] since we carefully optimize a key bottleneck in Fleischer's algorithm. All of these code artefacts are available on GitHub [2].

5 Evaluation

We evaluate NCFlow on several WAN topologies, traffic matrices, and failure scenarios to answer the following questions:

- Compared to state-of-the-art LP solvers and approximate combinatorial algorithms, does NCFlow offer a good trade-off between runtime and total flow allocation? Is it substantially faster, with only a small decrease in total flow?
- For real-world TE scenarios, in which flow solvers must adapt to changing demands and faults, how much benefit does NCFlow offer relative to the state-of-art?

[3]A source uses $N-1$ splitting rules in prior works but with NCFlow only requires $N_x+\eta-2$ rules when the source's cluster has N_x nodes; if clusters are evenly sized and $\eta \sim \sqrt{N}$, the ratio of these terms is $\sqrt{N}/2$.

Topology	# Nodes	# Edges	# Clusters
PrivateLarge	∼ 1000s	∼ 1000s	31
Kdl	754	1790	81
PrivateSmall	∼ 100s	∼ 1000s	42
Cogentco	197	486	42
UsCarrier	158	378	36
Colt	153	354	36
GtsCe	149	386	36
TataNld	145	372	36
DialtelecomCz	138	302	33
Ion	125	292	33
Deltacom	113	322	30
Interoute	110	294	20
Uninett2010	74	202	24

Table 5: Some of the WAN topologies used in our evaluation; see §5.1.

- How do our various design choices in NCFlow impact its performance?

5.1 Methodology

Here, we describe our methodology—the topologies, traffic, baselines, and metrics used in our evaluation.

Topologies: We use two real topologies from a large enterprise—PrivateSmall is a production internet-facing WAN with hundreds of sites, and PrivateLarge is a larger WAN that contains many more sites. We also use several topologies from the Internet Topology Zoo [6] and reuse topologies used by prior works [19, 38]. Table 5 shows details for some of the used topologies; note that the topologies shown are 10× to 100× larger than those considered by prior work [19, 35, 38, 44, 49].

Traffic Matrices (TMs): We benchmark NCFlow on traffic traces from PrivateSmall, which contain the total traffic between node pairs at 5-minute intervals. We also generate the following kinds of synthetic traffic matrices for all topologies:

- **Poisson**(λ, δ) models demands with varying concentration; the demand between nodes s and t is a Poisson random variable with mean $\lambda\delta^{\mathrm{d}_{st}}$, where d_{st} is the hop length of the shortest path between s and t and $\delta \in [0,1)$ is a *decay factor*. We choose δ close to 0 or to 1 to model strongly and weakly concentrated demands, respectively.
- **Gravity**(v) [14, 60]: The total traffic leaving a node is proportional to the total capacity on the node's outgoing links (parameterized by v); this traffic is divided among other nodes proportional to the total capacity on their incoming links.
- **Uniform**$([0,a))$: The traffic between any pair of nodes is chosen uniformly at random, between 0 and a.
- **Bimodal**$([0,a),[b,c),p)$ [14]: A p fraction of the node pairs, chosen uniformly at random, receive demands from **Uniform**$([b,c))$ while the rest receive demands from **Uniform**$([0,a))$. We use $p = 0.2$.

For each above model, we select parameters such that fully satisfying the traffic matrix leads to a maximum link utilization of about 10% in each topology. Then, we scale all entries in the TM by a constant $\alpha \in \{1,2,4,8,16,32,64,128\}$. Doing so creates demands that range from easily satisfiable to only partially satisfiable; with $\alpha = 128$, the satisfiable portion of the demand varies between 25-70%. We generate five samples for each traffic model and scale factor for each topology.

Baselines: We compare NCFlow with these techniques:

Path Formulation (PF_4) solves the multi-commodity max-flow problem shown in Equation 2 using k-shortest paths between node pairs where $k = 4$. Results for other path choices are in §G.4.

PF *Warm Start* (PF_{4w}) matches PF_4 except that it allows the LP solver to "warm start"; that is, over a sequence of traffic matrices, the flow allocated to the previous TM is used as a starting point to compute allocation for the next TM. When traffic changes are small, warm start leads to faster solutions.

Approximate Combinatorial Algorithms: Fleischer's algorithm [27] is the best-known approximation for MaxFlow. We use two variants: Fleischer-Path where flow is restricted to a path set and Fleischer-Edge without any path restrictions. We show results here for an approximation guarantee of 0.5; that is, the techniques must achieve at least half of the optimal flow allocation. Results for other approximation guarantee values are in [10].

SMORE [44] allocates flow dynamically on paths that are pre-computed using Räcke's Randomized Routing Trees (RRTs). We use the code from [45] to compute paths. Since the LP in [45] requires demands to be fully satisfiable, we implement a variant, SMORE*, that maximizes the total flow on the computed paths, regardless of demand satisfiability.

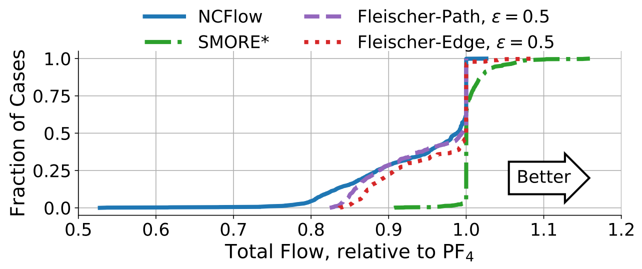
TEAVAR [3, 19] models link failure probabilities and computes flow allocations given an availability target. We implement a variant, TEAVAR*, that maximizes the total flow[4]; further details are in Appendix F.

Clusters, Paths, and # of Iterations: Table 5 shows the number of clusters used by NCFlow per topology. Here, we report results on edge-disjoint paths, chosen using inverse capacity as the edge length; results for other path choices are qualitatively similar (see §G.4). All schemes that use paths (i.e., PF_4, Fleischer-Path, TEAVAR*, and NCFlow) use the same method to compute paths. For each iteration up to $I = 6$, we also pre-compute offline the path to use on the aggregated graph, and the edge to use between connected clusters for each cluster bundle.

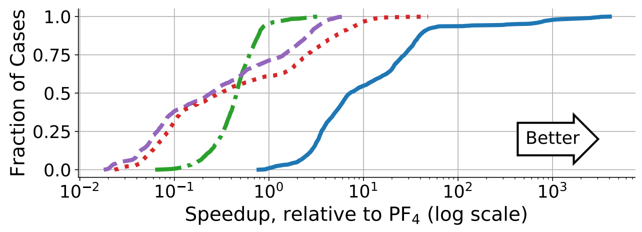
Metrics: We compare the schemes on the following metrics:

- **Relative total flow** is the total flow achieved by a scheme relative to PF_4.
- **Speedup ratio** is the runtime of each scheme relative to PF_4. For LP-based methods, we report the Gurobi solver runtimes, since models can be constructed once offline in practice. For combinatorial methods, we report algorithm execution time. All runtimes are measured on an Intel Xeon 2.3GHz CPU (E52673v4) with 16 cores and 112 GB of RAM.
- **FIB Entries:** We measure the number of switch forwarding entries used.

[4]TEAVAR [3, 19] maximizes the *concurrent* flow; see Table 1



(a) CDF of total flow relative to PF_4



(b) CDF of speedup relative to PF_4

Figure 11: CDFs comparing NCFlow with state-of-the-art methods. With only a modest decrease in total flow, NCFlow offers a substantial runtime speedup.

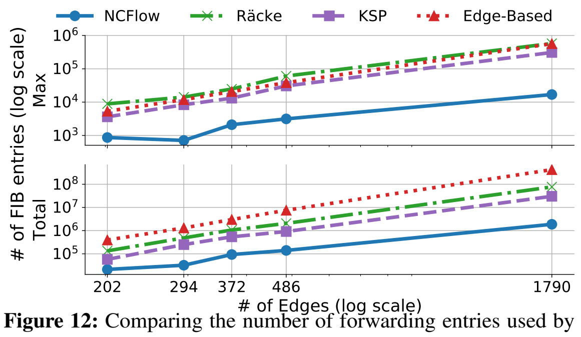


Figure 12: Comparing the number of forwarding entries used by various methods for the experiments from Figure 11.

5.2 Comparing NCFlow to the State of the Art

Figures 11a and 11b show cumulative density functions (CDFs) of the relative total flow and speedup ratio for NCFlow and several baselines. These results consist of 2,600 traffic matrices and 13 topologies. If a scheme matches the baseline PF_4, its CDF will be a pulse at $x = 1$ in both figures; the fraction of cases to the left (or right) of $x = 1$ indicate how often a scheme is worse (or better) than PF_4. Note that the x-axis for the speedup ratio is in log scale.

We see that SMORE*, shown using green dashed lines in the figures, modestly improves the flow allocation (in 25% of the cases) while almost always taking longer to run than PF_4.

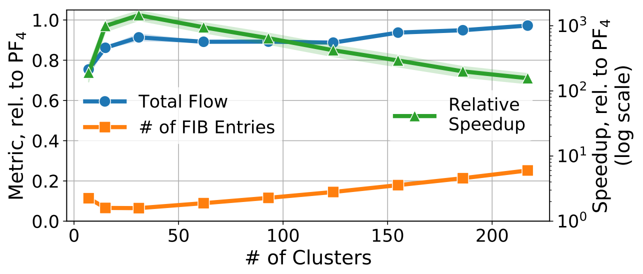


Figure 13: NCFlow's performance when using different numbers of clusters on PrivateLarge. The speedup ratio is plotted on the right y-axis in log scale; the other metrics use the left y-axis.

Both effects are because SMORE* allocates flow on Räcke's RRTs instead of k-shortest paths.

The edge and path variants of Fleischer's, shown using purple and red lines in the figures, perform similarly; since they are approximate algorithms, they allocate less flow than PF_4 in roughly 50% of cases, but are also faster than PF_4 in slightly less than 50% of cases. We conclude that these approximate algorithms are not practically better than PF_4.

In contrast, NCFlow, shown with dark blue lines in the figures, almost always allocates at least 80% of PF_4's total flow, while achieving large speedups. In the median case, NCFlow achieves 98% of the flow and is over 8× faster. These improvements accrue from NCFlow solving smaller optimization problems than PF_4.

Figures 18 and 19 tease apart the above results by load, traffic type and topology. Figures 23–27 show results for alternate path choices. Taken together, these results indicate that NCFlow's improvements hold across a variety of scenarios.

For the same experiments considered above, Figure 12 shows the number of switch forwarding entries used in different topologies. (A full set of results is in Table 6.) The bottom plot is the total number of forwarding entries across all switches, while the top shows the maximum for any switch. Note that both the x and y axes are in log scale. NCFlow consistently uses fewer forwarding entries; using NCFlow offers a greater amount of relative savings than switching from all edges to just a handful of paths per demand. The savings from NCFlow also increase with topology size. The reason, as noted in §3.5, is that NCFlow reuses pathlets and traffic splitting rules for many different demands.

5.3 Effect of Design Choices

Figure 13 shows how NCFlow's performance varies with the numbers of clusters used on PrivateLarge. While NCFlow allocates roughly the same amount of total flow, using about 30 clusters improves runtime and reduces forwarding entries. Figure 21 compares NCFlow's performance when using different clustering techniques; more details are in §G.2.

Recall from §3.3 that NCFlow uses multiple iterations of Figure 6. In the above experiments, the first iteration alone accounts for 75% of the runtime and for roughly 90% of the

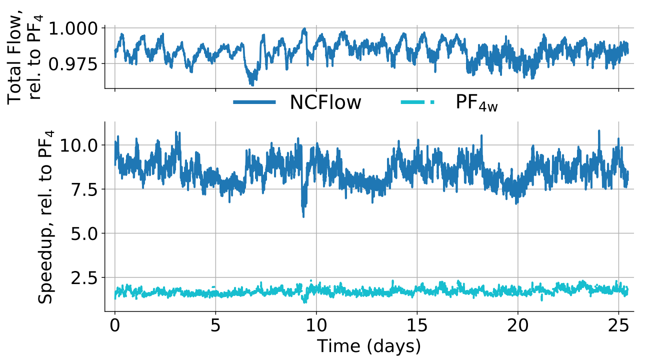


Figure 14: Allocated flow and speedup relative to PF_4 on a sequence of production TMs from PrivateSmall. In half of the cases, NCFlow allocates at least 98.5% of the flow and is at least 8.5× faster.

flow that is allocated by NCFlow. Later iterations are faster perhaps because they have less traffic to consider.

Breaking down the runtime by the steps in Figure 6, we see cases where MaxClusterFlow accounts for over 70% of NCFlow's runtime perhaps because the largest cluster contains a large fraction of the nodes. Better cluster choice or recursively dividing the largest clusters can further lower runtime.

5.4 NCFlow on Real-World Traffic

Here, we experiment with a sequence of traffic traces collected on the PrivateSmall WAN. Figure 14 plots the moving average (over 5 windows) of the total flow and speedup relative to PF_4 for two schemes—NCFlow in blue and PF_{4w} in light blue. The figure shows that PF_{4w}'s warm start yields a median speedup of 1.66×. NCFlow achieves a consistently higher speedup (8.5× in the median case), and the flow allocation is nearly optimal: the median total relative flow is 98.5%, and NCFlow always allocates more than 93%.

5.5 Tracking Changing Demands

Here, we evaluate the impact of a technique's runtime on its ability to stay on track with changing demands. Specifically, on the PrivateLarge topology, we use a time-series of traffic matrices, wherein a new TM arrives every five minutes and the change from one TM to the next is consistent with the findings in Figure 4 (more details are in Figure 20). At each time-step, all techniques have the opportunity to compute a new allocation for the current TM or to continue computing the allocation for an earlier TM if they have not yet finished; in the latter case, their most recently computed allocation will be used for the current TM. For example, a technique that requires five minutes to compute a new allocation will be always *one window behind*, i.e., each TM will receive the allocation that was computed for the previous TM.

Figure 15 shows the fraction of demand that is satisfied by three different schemes; we also show the value for an instantaneous scheme which is not penalized for its runtime.

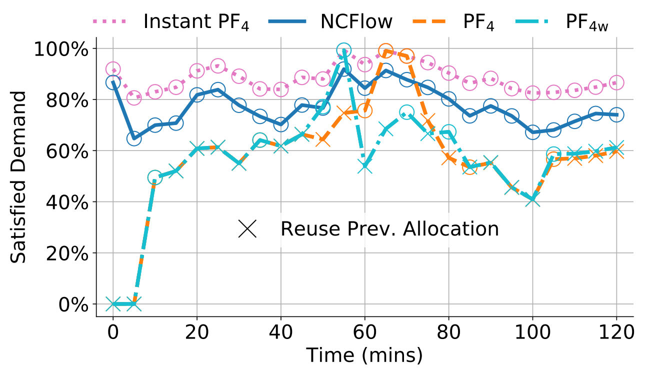


Figure 15: When demands change, how solver runtimes affect flow allocation on PrivateLarge: Due to the slow runtime, PF$_4$ and PF$_{4w}$ carry only 62% of the traffic that can be satisfied by Instant PF$_4$, a (hypothetical) scheme which has zero runtime. NCFlow carries 87% of the traffic since its faster runtime compensates for its sub-optimality.

PF$_4$'s average runtime here is over 15 minutes; hence, as the orange dashed line shows, PF$_4$ is able to compute a new allocation only for every third or fourth TM. This leads to substantial demand being unsatisfied: for node pairs whose current demand is larger than before, PF$_4$ will not allocate enough flow. On the other hand, node pairs whose current demand is less than their earlier demand will be unable to fully use PF$_4$'s allocation. As the figure shows, PF$_4$ only satisfies 53% of the changing demand on average, whereas Instant PF$_4$ satisfies 87% of the demand.

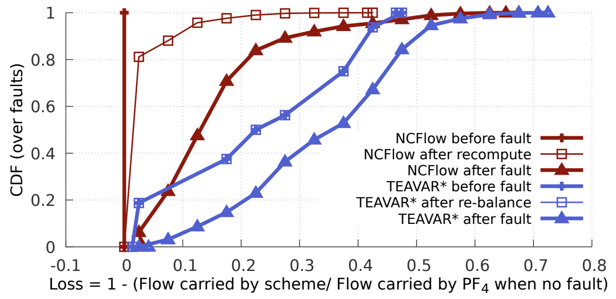
PF$_{4w}$ (the dash-dot light blue line), where the solver warm starts using the previous allocation, is modestly faster than PF$_4$ on average. As the figure shows, the average demand satisfied by PF$_{4w}$ is only slightly larger than PF$_4$ (about 54%).

In contrast, NCFlow (the solid dark blue line) finishes well within five minutes which allows allocations to change along with the changing demands. We find that on average NCFlow satisfies 75% of the demands; its smaller runtime more than makes up for sub-optimality, allowing NCFlow to carry more flow than PF$_4$ when demands change.

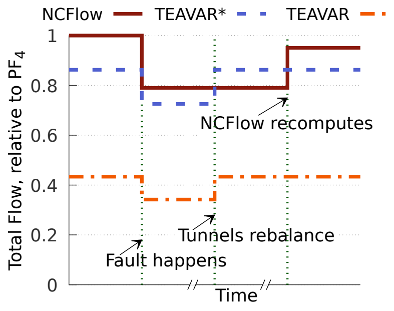
5.6 Handling Failures with NCFlow

Here, we evaluate the effect of link failures. As we note in §F, TEAVAR* did not finish within several days on any of the topologies listed in Table 5 because when all possible 2-link failure scenarios are considered, the number of equations and variables in the optimization problem increase from $O(N^2)$ for MaxFlow to $O(M^2N^2)$ for TEAVAR [19], where N and M are the numbers of nodes and edges, respectively. Hence, we report results on the 12-node, 38-edge WAN topology from B4 [38]. We generate synthetic traffic matrices as noted in §5.1. Using link failure probabilities from TEAVAR [3], we generate several hundred failure scenarios and, for each TM, we measure the flow carried by NCFlow and TEAVAR* before the fault, immediately after the fault, and after recovery.

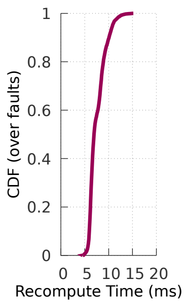
A key difference in fault recovery between NCFlow and TEAVAR* is that TEAVAR* requires sources to rebalance the traffic splits when a failure happens; doing so takes about one RTT on the WAN. Given a parameter β, TEAVAR* guarantees that there will be no flow loss after the tunnels re-balance with a probability of $1-\beta$. See §F for more details. We use $\beta = 0.99$, as recommended in [19]. NCFlow, on the other hand, recomputes flow allocations taking into account the links that have failed; doing so takes one execution of NCFlow and some RTTs to change the traffic splits at switches; more details are in §E. Figure 16c shows that the recomputation time is well within one RTT on the WAN.



(a) CDFs of the flow *loss* before faults, immediately after faults and after recovery (B4 topology, many traffic matrices and faults; see §5.6).



(b) Timelapse of when a fault occurs (B4 topology, Uniform traffic matrix, $\beta = 0.99$)



(c) NCFlow's time to recompute after fault.

Figure 16: Comparing failure response of NCFlow with prior work.

Figure 16b shows a timelapse of the flow carried on the network before the fault, immediately after the fault, and after recovery. As the figure shows, TEAVAR* can have a smaller loss and for a shorter duration; i.e., until sources rebalance traffic while NCFlow can carry more flow before fault and after recovery; moreover, the fast solver time can reduce the duration of loss.

Figure 16a shows CDFs over many faults and traffic matrices for NCFlow and TEAVAR*. We record the flow loss at three stages: before the fault, immediately after the fault, and after recovery. As the figure shows, NCFlow's ability to carry more flow before the fault and after recovery more than compensates for the slightly larger loss it may accrue in between.

6 Discussion

Extending beyond MaxFlow: FeasibleFlow is a common constraint for many objectives beyond MaxFlow (see Table 1). Since the algorithm in §3.1 and the heuristic in §3.2 guarantee feasibility, NCFlow can apply to objectives beyond MaxFlow;

however, we believe that more work is needed to improve the solution quality for different objectives.

Optimality guarantee: In §I, we show that constraining by clusters and paths, as done by NCFlow, does not necessarily reduce the flow allocation; that is, nearly the maximum amount of flow can be carried while respecting clustering and path constraints. This is promising because a better heuristic (than Figure 6) may allocate more flow without losing the benefits of solving smaller per-cluster problems. Furthermore, although NCFlow achieves sizable speedups by using simple clustering methods, the optimal cluster choice is uncertain; we show examples in §H to illustrate the challenges.

Recursive (or multiple levels of) clusters: For large topologies or when the largest cluster has a disproportionate number of nodes, we can further divide a cluster into sub-clusters. Doing so is an extension of the algorithm in Figure 6 where, in the iterative step, the MaxClusterFlow problem at a cluster is replaced with a new instance of all of the steps in Figure 6 along with the additional constraints that arise from the current level (e.g., NoMoreFlowThruCluster constraints). We leave further details to future work.

7 Related Work

NCFlow builds upon a few themes in prior work. We discuss and evaluate against some prior works already. To recap: (1) Some large enterprises use path-based global optimization problems similar to MaxFlow to manage traffic on their WANs [35, 36, 38]. We saw in §5 that doing so does not scale to the WAN topologies of today or the future, which consist of thousands of sites; (2) We saw that approximate algorithms for multi-commodity max flow, such as [27], require a large number of switch forwarding entries since they can send flow along any edge. Also, NCFlow allocates more flow and is faster compared to path-based versions of these algorithms. (3) Probabilistic fault protection schemes such as TEAVAR [19] take infeasibly long to run on large topologies when considering multiple link failures; they also allocate less flow to reserve capacity to deal with possible failures. Other oblivious techniques [13,14,19,44,49,66] have a similar tradeoff. Quickly recomputing using NCFlow trades off slightly more loss after a fault to carry much more traffic before the fault and after recomputation; hence, we believe that NCFlow is better suited to enterprise WANs, which target very high link utilization and have traffic that is elastic to short-term loss (e.g., scavenger-class traffic, such as replicating large datasets [35, 38, 49]). Here, we discuss other related work.

TE on WANs: Typically, a WAN node is not a single switch, but rather a group of switches connected in a specific way such as a full mesh. Similarly, a WAN edge is a systematic collection of links between many switches. [36] discusses how to hide the intra-node connectivity from the global TE solution. NCFlow complements this technique; it can use a similar intra-node scheme and can support WANs that are 10× larger than were considered in [36]. The specific contraction used by NCFlow—node clusters with large capacity and/or demand between themselves—also differs from the contractions used in route planning [4, 9, 15]. Some BGP-based TE schemes [24, 62, 69], which address how best to move traffic between different (BGP) domains, are also complementary to NCFlow which considers the WAN of a single enterprise (domain). Other TE schemes use different protocols, such as OSPF, or work over longer timescales (e.g., hours to days) [29, 39, 46, 51].

Multi-Commodity Flow Solutions: Both the edge- and path-based LP formulations are well-studied [16, 67]. Some works consider the case of a single commodity, i.e., one source and target, and do not directly extend to the case of multiple commodities [34,48,55]. The best-known approximate algorithms for multi-commodity flow problems incrementally allocate flow on the shortest path and increase the length of all edges on that path [17, 27, 30, 41]. For the problem sizes considered here, LP solvers such as Gurobi are faster in practice, perhaps because they take larger steps towards the optimal allocation. A few works customize LP solvers to improve performance on flow problems [23, 50]. NCFlow is agnostic to the solver used and can use any solver for the sub-problems in Figure 6.

Decompositions: Using standard decomposition techniques for large optimization problems, such as Dantzig-Wolfe and Benders [16,20], for multi-commodity flow problems has lead to inconclusive results [31,54]; i.e., not consistently faster than MaxFlow. NCFlow can be thought of as a problem-specific decomposition that leverages the observation that both capacity and demands are concentrated in today's WANs.

8 Conclusion

We present a fast and practical solution for allocating flow on large WANs. We leverage the concentrated nature of demands and topologies to divide nodes into clusters and solve sub-problems per cluster and on the aggregated graph. Our heuristics guarantee feasibility and empirically achieve close-to-optimal flow allocations. By reusing pathlets and splitting rules across demands, we require fewer forwarding entries in switches. Empirically, on topologies that are over 10× larger than were considered in prior work and many traffic matrices, our solution NCFlow is 8.2× faster than the state of the art, while allocating 98.8% of the total flow and using 6× fewer forwarding entries in the median case. We demonstrate that NCFlow offers sizable benefits when tracking changing demands and reacting to failures. As enterprise WANs continue to grow, we believe techniques such as NCFlow can enable improved traffic orchestration and higher link utilization.

Acknowledgements: We thank Himanshu Raj, Umesh Krishnaswamy, Dejan Kostic, Jakub Tarnawski and the NSDI reviewers for comments which improved this paper.

References

[1] Capacity planning for the Google backbone network. https://bit.ly/2lViJ4t.

[2] Code for NCFlow and Baselines. https://github.com/netcontract/ncflow.

[3] Code for TEAVAR. https://github.com/manyaghobadi/teavar.

[4] Contraction Hierarchies Path Finding Algorithm. https://bit.ly/3eaiqtg.

[5] GAP: Generalizable Approximate Graph Partitioning Framework. https://arxiv.org/pdf/1903.00614.pdf.

[6] Internet Topology Zoo. http://www.topology-zoo.org/.

[7] Market Trends: SD-WAN and NFV for Enterprise Network Services. https://gtnr.it/3c8hNyA.

[8] Cristinel Ababei. Code for Karakostas. https://bit.ly/2woSloP.

[9] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. A Hub-Based Labeling Algorithm for Shortest Paths in Road Networks. In *Experimental Algorithms*, 2011.

[10] Firas Abuzaid, Srikanth Kandula, Behnaz Arzani, Ishai Menache, Matei Zaharia, and Peter Bailis. Solving Flow Problems Quickly by Contracting Wide-area Network Topologies: Extended Version. https://bit.ly/35oyQdU.

[11] Ravindra Ahuja, Thomas Magnanti, and James Orlin. *Network Flows. Theory, Algorithms, and Applications*. Prentice Hall.

[12] Muthukaruppan Annamalai et al. Sharding the Shards: Managing Datastore Locality at Scale with Akkio. In *OSDI*, 2018.

[13] D. Applegate, L. Breslau, and E. Cohen. Coping with Network Failures: Routing Strategies for Optimal Demand Oblivious Restoration. In *SIGMETRICS*, 2004.

[14] David Applegate and Edith Cohen. Making Intra-Domain Routing Robust to Changing and Uncertain Traffic Demands. In *SIGCOMM*, 2003.

[15] Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route Planning in Transportation Networks. *CoRR*, 2015.

[16] Dimitris Bertsimas and John N Tsitsiklis. *Introduction to linear optimization*, volume 6. Athena Scientific Belmont, MA, 1997.

[17] Daniel Bienstock. *Potential function methods for approximately solving linear programming problems: theory and practice*, volume 53. Springer Science & Business Media, 2002.

[18] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks, 2008.

[19] Jeremy Bogle et al. TEAVAR: striking the right utilization-availability balance in WAN traffic engineering. In *SIGCOMM*, 2019.

[20] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[21] P. Brucker. On the complexity of clustering problems. In *Optimizing and Operations Research*, Berlin, West Germany, 1977. Springer-Verlag.

[22] Yiyang Chang, Sanjay Rao, and Mohit Tawarmalani. Robust validation of network designs under uncertain demands and failures. In *NSDI*, 2017.

[23] P. Chardaire and A. Lisser. Simplex and Interior Point Specialized Algorithms for Solving Nonoriented Multicommodity Flow Problems. *Operations Research*, 2002.

[24] David Chou, Tianyin Xu, Kaushik Veeraraghavan, Andrew Newell, Sonia Margulis, Lin Xiao, Pol Mauri Ruiz, Justin Meza, Kiryong Ha, Shruti Padmanabha, et al. Taiji: Managing Global User Traffic for Large-Scale Internet Services at the Edge. In *OSDI*, 2019.

[25] A. Clauset, M.E.J. Newman, and C. Moore. Finding community structure in very large networks. *Phys. Rev.*, 2004.

[26] Aaron Clauset. Fast Modularity Community Structure Inference Algorithm. https://bit.ly/3aAVGQH.

[27] Lisa K. Fleischer. Approximating Fractional Multicommodity Flow Independent of the Number of Commodities. *SIAM J. Discret. Math.*, 2000.

[28] Ken Florance. How Netflix Works With ISPs Around the Globe to Deliver a Great Viewing Experience. https://bit.ly/2RYYrEM, 2016.

[29] B. Fortz and Mikkel Thorup. Internet Traffic Engineering by Optimizing OSPF Weights in a Changing World. In *INFOCOM*, 2000.

[30] Naveen Garg and Jochen Könemann. Faster and Simpler Algorithms for Multicommodity Flow and Fractional Packing Problems. *SIAM J. Comput.*, 2007.

[31] A. M. Geoffrion and G. W. Graves. Multicommodity Distribution System Design by Benders Decomposition. *Management Science*, 1974.

[32] P. Brighten Godfrey, Igor Ganichev, Scott Shenker, and Ion Stoica. Pathlet routing. In *SIGCOMM*, 2009.

[33] Zonghao Gu, Edward Rothberg, and Robert Bixby. Gurobi optimizer reference manual, version 5.0. *Gurobi Optimization Inc., Houston, USA*, 2012.

[34] Jeff Hartline and Alexa Sharp. Hierarchical Flow. Technical Report 2004-09-29, Cornell University, 2004.

[35] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven WAN. In *SIGCOMM*, 2013.

[36] Chi-Yao Hong, Subhasree Mandal, Mohammad Al-Fares, Min Zhu, Richard Alimi, Chandan Bhagat, Sourabh Jain, Jay Kaimal, Shiyu Liang, Kirill Mendelev, et al. B4 and after: managing hierarchy, partitioning, and asymmetry for availability and scale in google's software-defined WAN. In *SIGCOMM*, 2018.

[37] Yuanfang Hu, Yi Zhu, Hongyu Chen, Ronald L. Graham, and Chung-Kuan Cheng. Communication latency aware low power NoC synthesis. In *DAC*, 2006.

[38] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, and Min Zhu. B4: Experience with a globally-deployed software defined WAN. In *SIGCOMM*, 2013.

[39] Virajith Jalaparti, Ivan Bliznets, Srikanth Kandula, Brendan Lucier, and Ishai Menache. Dynamic pricing and traffic engineering for timely inter-datacenter transfers. In *SIGCOMM*, 2016.

[40] S. Kandula, I. Menache, R. Schwartz, and S. R. Babbula. Calendaring for Wide Area Networks. In *SIGCOMM*, 2014.

[41] George Karakostas. Faster Approximation Schemes for Fractional Multicommodity Flow Problems. *ACM Trans. Algorithms*, 2008.

[42] Robert Krauthgamer, Joseph (Seffi) Naor, and Roy Schwartz. Partitioning Graphs into Balanced Components. In *SODA*, 2009.

[43] Alok Kumar et al. Bwe: Flexible, hierarchical bandwidth allocation for wan distributed computing. In *SIGCOMM*, 2015.

[44] Praveen Kumar et al. Semi-Oblivious Traffic Engineering: The Road Not Taken. In *NSDI*, 2018.

[45] Praveen Kumar, Chris Yu, Yang Yuan, Nate Foster, Robert Kleinberg, and Robert Soulé. YATES: Rapid Prototyping for Traffic Engineering Systems. In *SOSR*, 2018.

[46] Nikolaos Laoutaris, Michael Sirivianos, Xiaoyuan Yang, and Pablo Rodriguez. Inter-datacenter Bulk Transfers with NetStitcher. In *SIGCOMM*, 2011.

[47] David Lebrun, Mathieu Jadin, François Clad, Clarence Filsfils, and Olivier Bonaventure. Software Resolved Networks: Rethinking Enterprise Networks with IPv6 Segment Routing. In *SOSR*, 2018.

[48] Chansook Lim, S. Bohacek, Joao Hespanha, and Katia Obraczka. Hierarchical Max-Flow Routing. In *Globecom*, 2005.

[49] Hongqiang Harry Liu, Srikanth Kandula, Ratul Mahajan, Ming Zhang, and David Gelernter. Traffic engineering with forward fault correction. In *SIGCOMM*, 2014.

[50] Richard McBride. Progress Made in Solving the Multicommodity Flow Problem. *SIAM Journal on Optimization*, 1998.

[51] Srinivas Narayana, Joe Jiang, Jennifer Rexford, and Mung Chiang. Distributed Wide-Area Traffic Management for Cloud Services. In *SIGMETRICS*, 2012.

[52] NetworkX. Edge Disjoint Paths. `https://bit.ly/37VJ71k`.

[53] Andrew Y Ng, Michael I Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. In *NIPS*, 2002.

[54] Murat Oguz, Tolga Bektas, and Julia A. Bennell. Multicommodity flows and Benders decomposition for restricted continuous location problems. *European Journal of Operational Research*, 2017.

[55] James Orlin. A polynomial time primal network simplex algorithm for minimum cost flows. *Math. Programming*, 1997.

[56] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Victor Bahl, and Ion Stoica. Low Latency Geo-distributed Data Analytics. In *SIGCOMM*, 2015.

[57] H Racke. Optimal Hierarchical Decompositions for Congestion Minimization in Networks. In *STOC*, 2008.

[58] R Tyrrell Rockafellar and Stanislav Uryasev. Conditional Value-at-Risk for General Loss Distributions. *Journal of banking & finance*, 26(7):1443–1471, 2002.

[59] E. Rosen, A. Viswanathan, and R. Callon. Multi-Protocol Label Switching Architecture. RFC 3031.

[60] Matthew Roughan, Albert Greenberg, Charles Kalmanek, Michael Rumsewicz, Jennifer Yates, and Yin Zhang. Experience in Measuring Backbone Traffic Variability: Models, Metrics, Measurements and Meaning. In *IMW*, 2002.

[61] S. Kandula and D. Katabi and S. Sinha and A. Berger. Dynamic Load Balancing Without Packet Reordering. In *CCR*, 2006.

[62] Brandon Schlinker et al. Engineering Egress with Edge Fabric: Steering Oceans of Content to the World. In *SIGCOMM*, 2017.

[63] Submarine Cable Map. http://www.submarinecablemap.com.

[64] Asaf Valadarsky, Michael Schapira, Dafna Shahaf, and Aviv Tamar. Learning to Route. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets, 2017.

[65] Santosh Vempala, Ravi Kannan, and Adrian Vetta. On Clusterings Good, Bad and Spectral. In *FOCS*, 2000.

[66] Hao Wang, Haiyong Xie, Lili Qiu, Yang Richard Yang, Yin Zhang, and Albert Greenberg. COPE: Traffic Engineering in Dynamic Networks. In *SIGCOMM*, 2006.

[67] I-Lin Wang. Multicommodity Network Flows: A Survey, Part I: Applications and Formulations. *Internal Journal of Operations Research*, 2018.

[68] Mathieu Xhonneux, Fabien Duchêne, and Olivier Bonaventure. Leveraging eBPF for programmable network functions with IPv6 Segment Routing. In *CoNext*, 2018.

[69] Kok-Kiong Yap et al. Taking the edge off with espresso: Scale, reliability and programmability for global internet peering. In *SIGCOMM*, 2017.

[70] Jin Yen and NetworkX. K-Shortest Paths. https://bit.ly/2OpNGJn.

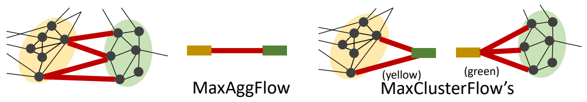


Figure 17: Considering the crossing edges between the yellow and green clusters from Figure 2; MaxAggFlow has a single bundle; the yellow and green instances of MaxClusterFlow have one bundle for each incident node in their cluster.

A More Discussion

NCFlow is **agnostic to the underlying solver** used for the problems in Figure 6 and can benefit from future improvements to LP solvers and approximate methods [27, 30, 41].

Further use cases: Beyond serving as a drop-in replacement for today's production WAN traffic controllers, NCFlow can be used whenever fast and close-to-optimal solutions are desirable such as: when allocating flow for future timesteps [39, 40] or to compare topology changes [1, 22] or to accelerate the training of ML-based routing systems [64].

B Properties of NCFlow's flow allocation algorithm

B.1 Proof that the algorithm in §3.1 meets demand and capacity constraints

Satisfying demand constraints: Demands whose source and target are in the same cluster are considered by only one instance of MaxClusterFlow; hence, they do not receive more flow than their demands. Specifically, MaxClusterFlow in Figure 6 invokes MaxFlow which in turn imposes the demand constraints listed in FeasibleFlow; Equation 1.

Demands whose source and target are in different clusters receive no more flow than their demand due to SrcTargetMax; observe in Figure 6 that one of the four constraints in SrcTargetMax explicitly controls the flow for such demands.

Satisfying edge capacity constraints: We say an edge is local to a cluster if both its incident nodes are within the same cluster. Flow is assigned to a local edge only by the MaxClusterFlow instance of the cluster that contains that edge. Since MaxClusterFlow ultimately invokes FeasibleFlow; by Equation 1 a local edge is allocated no more than its capacity.

Edges that are not local receive flow allocation in MaxAggFlow where, as noted in §3.1, all of the edges that lie between a pair of clusters are treated as a single edge whose capacity equals the sum of the capacity of the underlying edges. Thus, the flow assigned to a bundle of edges by MaxAggFlow is no more than the total capacity of the edges in the bundle. Subsequently, MaxClusterFlow instances behave similarly; that is, the flow allocated for a bundle of edges is no more than the capacity of that bundle. For example, Figure 17 shows the four edges between the yellow and green clusters in Figure 2 as well as the bundles considered by MaxAggFlow (in the middle) and the two instances of MaxClusterFlow corresponding to the yellow and green clusters on the right. The later steps in Figure 6 do not increase flow and so we conclude that capacity constraints are satisfiable for all non-local edges.

B.2 Proof that the heuristic in §3.2 leads to feasible flow allocations

Here, we prove Theorem 1. First, note that the heuristic in §3.2 which only restricts the edges between clusters and paths on the aggregate graph that can be used by some demands does not affect the proof in §B.1; that is, edges still receive flow less than their capacity and demand constraints hold.

We now prove that the heuristic will satisfy flow conservation; that is, at any node in the network, for any demand which neither originates nor ends at this node, the net flow is zero, i.e., incoming flow to the node equals the flow leaving that node.

It is easy to see that flow conservation holds for demands whose source and target are in the same cluster even without the heuristic in §3.2 because: (1) Only the instance of MaxClusterFlow for that cluster assigns flow to such a demand. (2) Since MaxClusterFlow invokes FeasibleFlow in Equation 1, the flow is allocated along paths which start and end at the source and target of that demand respectively. (3) Thus, every node that is neither the source or target will have incoming flow equal to the outgoing flow.

We now consider the remaining demands, that is, whose source and target are in different clusters.

It is easy to see that for such demands, flow conservation holds at all nodes that do not have edges to or from other clusters by logic that is similar to the above. The MaxClusterFlow instance of the cluster containing such a node would allocate flow to some bundle of demands on paths in this cluster that neither start nor end at such a node.

The only case left is nodes which have edges to and from other clusters. Suppose by contradiction that some demand k violates flow conservation at such a node u. The heuristic in §3.2 allocates flow for demand k along only one path in the aggregated graph and on only one edge between connected clusters. If the cluster containing u is not on the chosen path or none of the chosen edges are incident on u, then the net flow allocated for k over all edges incident on u will be zero. Let e be that one chosen crossing edge incident on u which can receive non-zero flow for demand k. Observe that all of the other demands whose source and target are in the same clusters as k would also be allocated flow on the same path and edges as k. Thus, all the flow allocated for these demands entering or leaving node u as the case may be would be on edge e. Two instances of MaxClusterFlow, one corresponding to the cluster that contains u and another corresponding to the other side of edge e, will assign possibly different flow values for this bundle of demands on edge e. To conclude our proof, note that MinPathE2E takes the minimum flow assigned along all such crossing edges e on the chosen path through the aggregated graph and that SrcTargetMax further breaks open the bundle to assign feasible flow for each actual demand contained in the bundle.

If more than one crossing edge or more than one path on the aggregate graph are used for a demand, it is easy to see how the above proof will break. The two instances of MaxClusterFlow that correspond to the clusters on either side of a crossing edge will be forced by MinPathE2E to only agree on the total volume for the cluster bundle of demands for all edges between the pair of clusters; that is, these instances may allocate different flow on different edges or allocate different flow to individual demands in the bundle. Figure 8 shows simple examples of such disagreement.

B.3 Proof of optimality for algorithm in §3.1 given some sufficient conditions

Here, we prove Theorem 2. We already discussed in §3.3 the case where the number of clusters, η, is 1 or N, the number of nodes in the graph. To prove optimality for the other sufficient conditions, we posit a helper theorem.

Theorem 3. *Given a set of paths $\mathcal{P}$ that can be used by flows, there exists a clustering of nodes into clusters such that any flow allocated on a set of paths $\mathcal{P}$ can also be allocated by the method in Figure 6 over those clusters.*

Proof. The claim is trivially true by using N clusters, where each node is in a cluster by itself. We show that it is possible to use fewer clusters next. Let $\mathcal{S}$ be a set of nodes such that every path in $\mathcal{P}$ contains at most one contiguous sequence of the nodes in $\mathcal{S}$. For example, the set $\{u, v\}$ satisfies this property if every path in $\mathcal{P}$ has neither u nor v, just u but not v (no repetitions allowed), just v but not u, $u \to v$ (no repetitions of u or v anywhere else in the path) or $v \to u$. Coalescing each such set $\mathcal{S}$ into a cluster would allow the method in Figure 6 to allocate the same flow as MaxFlow using the paths in $\mathcal{P}$. □

If G_{agg} is a tree and there is at most one edge between any pair of clusters, any set of paths $\mathcal{P}$ on the actual graph would consist of contiguous segments that are contained within each cluster. Thus, per the above theorem, any flow allocated by MaxEdgeFlow (Equation 6) can also be allocated by the method in Figure 6. The only difference then between the global optimization and the method in Figure 6 is that whereas the former is a single optimization call, the latter is a sequence of optimizations. Since demands are satisfiable, however, all of the steps in Figure 6 will allocate the entirety of demand and hence will allocate the maximum amount of flow.

Note, in particular, that for the sufficient conditions listed in Theorem 2 a single iteration of the steps in Figure 6 suffice.

In §H, we show some counter-examples where NCFlow can lead to sub-optimal allocations when any of these sufficient conditions do not hold.

C Data-plane details for NCFlow

C.1 Actions at the NCFlow controller, after each allocation

The SDN controller for NCFlow computes total flow per demand and some splitting ratios after each allocation.

Total Flow: The flow assigned to a demand whose source and target are in different clusters is read off SrcTargetMax, i.e., $f_{4,k}$. For intra-cluster demands, their flow is read off MaxClusterFlow, i.e., $f^x_{2,k}$ at the cluster x that contains the source and target of demand k. These flow values are summed up over all the iterations used by NCFlow.

Splitting ratios at sources: At source s of cluster x, we have two cases depending on whether the target of the demand is within the cluster x or in some other cluster y.

For the former case, let $\mathcal{P}_{st}$ be the path set to target t for demand k; the splitting ratio for each path p in the set is $f^{x,p}_{2,k}$ summed up over all iterations, divided by the total flow assigned to demand k above. Here, $f^{x,p}_{2,k}$ is the flow assigned to demand k on path p by the MaxClusterFlow instance for cluster x.

For the latter case, let z_i be the next cluster on the one path that can receive flow in iteration i for all traffic going to targets in cluster y. The splitting ratio for path p in the path set $\bigcup_i \mathcal{P}_{sz_i}$ is the value of $\sum_{r\in K_{sy}} f^{x,p}_{2,r}$ summed up over all iterations where K_{sy} is the set of all demands from source s to targets in cluster y divided by the total value for all such paths.

Uniquely, note that each source s has a splitting ratio per target t within the same cluster or per target cluster y.

We call a subset of nodes as ingresses if they have at least one edge to a node in another cluster that is chosen by the offline component of NCFlow in §3.4 as a crossing edge

Splitting ratios at ingresses are computed in a similar way to the splitting ratios at sources. At each ingress node w of cluster y for traffic from cluster x, there are two cases depending on whether the target is some node t in the same cluster as the ingress (y) or in some other cluster z.

For the former case, in iteration i, the splitting ratio for path p in the set $\mathcal{P}_{wt}$ is the value of $\sum_{r\in K_{xt}} f^{y,p}_{2,r}$ in iteration i divided by the total over all such paths. As above, K_{xt} is the set of demands from sources in cluster x to target t.

For the latter case, in iteration i, let z_i be the next cluster on the path to targets in z; the splitting ratio for path p in the set $\mathcal{P}_{wz_i}$ is the value of $\sum_{r\in K_{xz}} f^{y,p}_{2,r}$ divided by the total value over all such paths. As above, K_{xz} is the set of all demands from sources in cluster x to targets in cluster z.

Note that an ingress node w has splitting ratios only for demands whose chosen path at an iteration contains w's cluster (y) and whose chosen edge enters y at w.

C.2 Details on switch forwarding entries

Pathlets: NCFlow sets up label-switched paths (LSPs) between each pair of nodes in each cluster. Which paths to setup is pre-determined by the offline component in §3.4.

Splitting rules: A source s in cluster x has a splitting rule for each other node in the same cluster and for each other cluster. The splitting ratios are as computed in §C.1.

In each iteration, at each cluster, at most one ingress node is active per pair of other clusters. This is because the bundle of demands for a given pair of clusters has at most one crossing edge entering a cluster.

The active ingress node at a cluster x for the bundle of demands from cluster y to cluster z has one splitting rule when $z \neq x$ and one splitting rule per target in cluster x when $z = x$.

Packet content: The LSP (which pathlet to use) is encoded in the L2 header [59]. Additionally, NCFlow has the following tuple in each packet: (x, y, i, e) where x and y are the source and target cluster ids, i is the iteration number of the flow allocation that the packets have been assigned to and e is the edge to leave the current cluster on. The bits needed are $2\ln\eta + \ln I + \ln \text{node degree}$.[5] We note that 16 bits of header space suffice for all the WAN topologies and experiments considered in this paper; that is $\eta \leq 64$ clusters, $I \leq 8$ iterations and up to 2 edges to nodes in other clusters being used per egress node by NCFlow.

Data path actions:

- At source s in cluster x:
  - The host or middleware adds the cluster-ids x and y into the packet.
  - Source switch uses the appropriate splitting rule to pick a (p, i, e) tuple; the values e and i are placed in the packet and the L2 header gets the identifier for path p. To avoid reordering packets in the same TCP flow, traffic can be split using flow hashes or flowlets [61].
- Each cluster egress removes e from the packet header and forwards packets to the next-hop of the edge e.
- Each cluster ingress uses the appropriate splitting rule to pick a (p, e) tuple; the value e is put into the packet header and p determines the identifier in the L2 header.

D Definitions of NoMoreFlow

In the flow vector computed by MaxClusterFlow at a cluster x, $\mathbf{f}^{\mathbf{x}}_{\mathbf{2}}$, we use the subscript k to denote a bundle that may include (1) transit demands through cluster x (i.e., from all sources in some other cluster w to targets in some other cluster z), (2) leaving demands (i.e., from a source in cluster x to

[5]The edge id must suffice to distinguish at an egress node between the edges to a particular next cluster; so node degree is an overestimate.

all targets in some other cluster z) or (3) entering demands (i.e., to a target in cluster x from all sources in some other cluster z). Furthermore, we use the subscript y_{out} to denote the flow allocated for the bundle k on paths to the virtual node that corresponds to the cluster y. Thus, $f^x_{2,k,y_{\text{out}}}$ is the flow allocated at cluster x for all demands in the per-cluster bundle k on paths to the virtual node corresponding to a neighboring cluster y.

With this background, Equation 3 ensures that the flows allocated in MinPathE2E for an inter-cluster bundle K in $\mathcal{D}_{\text{agg}}$ on all paths in $\mathcal{P}_{\text{agg}}$ that contain a cluster edge (x,y) is no more than the flow that is allocated at either cluster x or cluster y for their respective per-cluster bundles that are contained in K to and from each other respectively.

$$\boxed{\begin{aligned}&\mathsf{NoMoreAlongPaths}(\mathbf{f},\mathbf{f_2}) \triangleq \forall K \in \mathcal{D}_{\text{agg}},\ \forall x,y \in \mathcal{V}_{\text{agg}}, x \neq y,\\ &\sum_{p\in\mathcal{P}_{\text{agg}},(x,y)\in p} f^p_K \leq \min\left(\sum_{k\in K} f^x_{2,k,y_{\text{out}}}, \sum_{k'\in K} f^y_{2,k',x_{\text{in}}}\right) \qquad (3)\end{aligned}}$$

Equation 4 is logically similar to Equation 3 except that the constraints are specific to a cluster x and the constants and variables have been flipped; that is, here, the flows on the paths in the aggregate graph are given ($f^p_{1,K}$) and the flow on paths within the cluster are to be computed by MaxClusterFlow. In particular, note that $\sum_{p'\in\mathcal{P}_{x,*y_{\text{out}}}} f^{p'}_k \triangleq f^x_{2,k,y_{\text{out}}}$; that is, the flow assigned in MaxClusterFlow of cluster x on all paths leading to the virtual node corresponding to a neighbor cluster y is precisely the value on the right that is used above in Equation 3.

$$\boxed{\begin{aligned}&\mathsf{NoMoreFlowThruCluster}(\mathbf{f},\mathbf{f_1},x) \triangleq \forall K \in \mathcal{D}_{\text{agg}},\ \forall y \in \mathcal{V}_{\text{agg}} : y \neq x,\\ &\sum_{p\in\mathcal{P}_{\text{agg}}:(y,x)\in p} f^p_{1,K} \geq \sum_{k\in K,\ p'\in\mathcal{P}_{x,y_{\text{in}}*}} f^{p'}_k, \text{ and}\\ &\sum_{p\in\mathcal{P}_{\text{agg}}:(x,y)\in p} f^p_{1,K} \geq \sum_{k\in K,\ p'\in\mathcal{P}_{x,*y_{\text{out}}}} f^{p'}_k \qquad (4)\end{aligned}}$$

E Fault Model

When failures happen, prior works [19, 49] assume that the sources of the label switched paths (LSPs) will proportionally shift traffic. That is, a source that splits traffic in the ratio of $(0.3, 0.5, 0.2)$ between three paths will change to a splitting ratio of $(0.6, 0, 0.4)$ when the middle LSP fails. Doing so can cause congestion on either of the remaining LSPs.

The key idea in prior works [19, 49] is to proactively allocate flow such that the maximal load on any link remains under capacity—FFC [49] protects against up to k simultaneous link failures, whereas TEAVAR [19] ensures that the flow at risk is below a given fraction (e.g., 99.9% of flow can be carried by the network on average over all possible failure scenarios).

The cost of such congestion protection is two-fold: (1) proactive schemes substantially increase the solution runtime, and (2) they under-allocate flow, since capacity must be set aside to help with possible failures. Instead, NCFlow uses a *reactive* strategy, and recomputes a new flow allocation after the fault occurs. This enables NCFlow to carry more flow before the fault, and potentially carry more flow after recovery. Furthermore, since NCFlow uses fewer FIB entries for the same number of paths, it is naturally easier to spread flow onto more paths with NCFlow. Thus, the key trade-off is slightly longer and more lossy episodes immediately after a fault when using NCFlow versus longer solver runtimes and flow under-allocation with proactive schemes [19, 49].

F Benchmarking TEAVAR and TEAVAR*

F.1 Formulation for TEAVAR*

Here, we discuss our adaptation of TEAVAR to maximize total multi-commodity flow. The TEAVAR [19] paper considers a different objective – maximizing the *concurrent* multi-commodity flow (see Table 2). When all demands are satisfiable, both objectives allocate the same flow; however, when not enough capacity is available to meet the desired failure assurance, maximizing total flow leads to a strictly larger allocation. We describe TEAVAR* from first principles here.

In addition to the inputs of MaxFlow (see Equation 2), TEAVAR* has the following inputs:

- A value $\beta \in [0,1]$; larger values of β correspond to greater fault assurance.
- A set of fault scenarios, $\mathcal{S}$; each scenario i has a probability of occurrence β_i and a set of failed edges $\mathcal{E}_i$.

In a fault scenario i, the edges in $\mathcal{E}_i$ will fail and so the flow allocated to paths that contain any edge in $\mathcal{E}_i$ will be *lost*. The number of possible fault scenarios is exponential in the number of edges in the network. Thus, to keep the optimization tractable, we consider only a subset of scenarios.

Let $\mathcal{L}(i)$ denote the total flow lost in fault scenario i. Per Proposition 8 in [58], minimizing the potential function, $\alpha + \frac{1}{1-\beta}\mathbb{E}[\mathcal{L}_i - \alpha]^+$, would minimize the conditional value at risk. Here, the expectation is over all possible fault scenarios. Since we only consider a subset of fault scenarios to keep optimization tractable, we minimize: $\alpha + \frac{1}{1-\beta}\left(\sum_{i\in\mathcal{S}} \beta_i[\mathcal{L}_i - \alpha]^+ + (1 - \sum_{i\in\mathcal{S}} \beta_i)(1-\alpha)\right)$. The last term accounts for the unconsidered scenarios for which we must assume the worst possible loss. Note that we can simplify this expression by dropping the constant $\frac{1-\sum_{i\in\mathcal{S}}\beta_i}{1-\beta}$.

$$\begin{aligned}
&\textsf{TEAVAR}^*(\mathcal{V},\mathcal{E},\mathcal{D},\mathcal{P},\beta,\mathcal{S}) && && (5)\\
&\triangleq \arg\min_{\mathbf{f}} \left(\alpha + \tfrac{1}{1-\beta}\left(\textstyle\sum_{i\in\mathcal{S}} \beta_i \textsf{Excess}_i - (1-\textstyle\sum_{i\in\mathcal{S}}\beta_i)\alpha\right)\right)\\
&\text{s.t. } \mathbf{f}\in \quad \textsf{FeasibleFlow}(\mathcal{V},\mathcal{E},\mathcal{D},\mathcal{P}), && && \text{(Eqn. 1)}\\
&\quad \mathcal{L}_{i,k}\ge 0, && \forall i,k && \text{(loss is non-negative)}\\
&\quad \mathcal{L}_{i,k}\ge d_k - \sum_{p\in\mathcal{P}_k} f_k^p \textsf{Active}_{p,i}, && \forall i,k && \text{(loss)}\\
&\quad \alpha \ge 0 && && \text{(loss } \textit{cutoff}\text{)}\\
&\quad \textsf{Excess}_i \ge 0 && \forall i && \text{(excess loss in scenario } i\text{)}\\
&\quad \textsf{Excess}_i \ge \sum_{k\in\mathcal{D}} \mathcal{L}_{i,k} - \alpha, && \forall i && \text{(excess loss)}
\end{aligned}$$

The formulation for TEAVAR* is in Equation 5. Recall that f_k^p is the flow assigned to demand k on path p. Active$_{p,i}$ is an indicator denoting whether path p is active in fault scenario i. Thus, the allocation for demand k in scenario i will be $\sum_{p\in\mathcal{P}_k} f_k^p \textsf{Active}_{p,i}$. When the allocation is below the required volume d_k, the demand will suffer loss; we use $\mathcal{L}_{i,k}$ to denote the flow loss for demand k in scenario i.

The flow allocation resulting from the above formulation cannot be promised to the demands; in particular, more flow will be assigned on some paths to account for possible failures on other paths. After solving the above LP, we compute the flow allocation for a demand k as follows: (1) sort the per-scenario losses $\mathcal{L}_{i,k}$ in ascending order; (2) starting at index 0, add up the probability of each scenario until the running sum is at least β—let i_β be the unique crossing index; (3) Set demand k's flow to be $d_k - \mathcal{L}_{i_\beta,k}$, the demand minus the loss at the crossing index.

Choosing the fault scenarios to use in TEAVAR*:

- Intuitively, achieving a greater amount of fault assurance requires considering more fault scenarios. Specifically, if the total probability of considered scenarios is below β, the above LP as well as the LP used by TEAVAR become unbounded. To see why, the coefficient of α in Eqn. 5 is $\frac{(\sum_{i\in\mathcal{S}}\beta_i)-\beta}{1-\beta}$. If the probability of considered scenarios is less than β, this coefficient becomes negative, and the objective value reaches $-\infty$ by setting α to ∞.
- Intuitively, if the total probability of considered scenarios is just larger than β, the flow allocated to demands is very small. To see why, the smaller the value of $\sum_{i\in\mathcal{S}}\beta_i - \beta$, the smaller the positive coefficient of α in the objective of Eqn 5. Thus, the solution of Eqn 5 will have a large value of α and a very small amount of allocatable flow.
- In light of these two points, in our experiments, we choose all scenarios that are individually more likely to occur than a cutoff ρ and multiplicatively reduce ρ until the total probability of considered scenarios exceeds $1-\frac{1-\beta}{2}$.

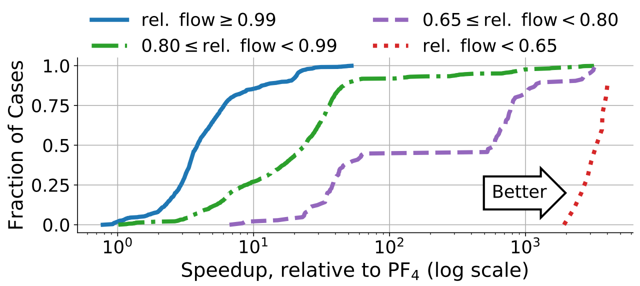


Figure 18: Breaking down the NCFlow results from Figure 11b into four separate CDFs based on relative total flow.

F.2 Comments on benchmarking TEAVAR

Observe that the number of scenarios affects the complexity of the TEAVAR* optimization; specifically, the number of equations and variables increases by $|\mathcal{S}| * |\mathcal{P}|$. The path set is at least as large as the node pairs, i.e., $|\mathcal{P}| > N^2$ where N is the number of nodes. The appropriate choice of fault scenarios to consider, as discussed above, depends on the size of the topology, the failure probability of edges, and the required assurance level β. Suppose one considers all 2-edge failure scenarios; then $|\mathcal{S}| \sim M^2$ where M is the number of edges. Hence, the increase in equations and variables exceeds N^2M^2. Note that MaxFlow is substantially simpler, having at most $O(N^2)$ variables and constraints (Equation 1).

On the topologies listed in Table 5, our implementation of TEAVAR* never ran to completion even after several days. We ran with $\beta = 0.99$ and link failure probability set to 0.004; both of these are the default values used in [3]. The reason is that the optimization problem becomes intractably large. TEAVAR behaves similarly [19]. We conclude that probabilistic fault protection using this methodology is infeasible on large topologies and for non-trivial fault assurance levels such as when considering multiple link failures.

We also note that we are unable to simultaneously achieve the solution quality and the runtimes that are reported in TEAVAR [19] using their code [3]. Specifically, achieving the assurance levels reported in their experiments requires many scenarios to be considered. The runtimes reported in [19] appear to have been measured when considering only single link failures.

G Additional Experiments

G.1 Breakdown of NCFlow's Performance

To further understand the performance of NCFlow, Figure 18 breaks down the results in Figure 11 into four ranges based on total relative flow. We plot CDFs of the speedup ratio per range. The solid blue and green dashed line, which correspond to relative flow above 0.99 and in $[0.8, 0.99)$ respectively, account for 49% and 46% of all experiments. The figure shows that NCFlow achieves sizable speedups while allocating large

Topology	Edge-Based	Räcke	KSP	NCFlow
Total # FIB Entries				
PrivateLarge	945,038,502	52,515,090	22,483,244	**1,694,027**
Kdl	427,524,786	76,794,001	30,199,751	**1,876,289**
PrivateSmall	7,684,182	1,232,866	625,282	**139,346**
Cogentco	7,567,952	2,054,323	915,207	**139,862**
UsCarrier	3,894,542	1,520,821	510,894	**82,301**
Colt	3,534,912	1,048,779	346,905	**67,307**
GtsCe	3,263,696	1,077,350	535,135	**101,368**
TataNld	3,006,720	1,062,629	540,088	**93,179**
DialtelecomCz	2,590,122	1,427,780	529,663	**83,128**
Ion	1,922,000	886,414	418,362	**71,614**
Deltacom	1,417,472	459,159	246,811	**53,948**
Interoute	1,306,910	483,960	249,979	**32,193**
Uninett2010	394,346	133,742	57,428	**21,185**
Maximum # FIB Entries				
PrivateLarge	962,361	828,397	313,850	**18,124**
Kdl	567,009	576,274	309,575	**16,926**
PrivateSmall	38,809	49,663	21,796	**3,639**
Cogentco	38,416	60,676	30,601	**3,144**
UsCarrier	24,649	41,897	17,822	**2,234**
Colt	23,104	47,077	17,344	**3,572**
GtsCe	21,904	36,070	15,477	**2,748**
TataNld	20,736	24,776	13,179	**2,104**
DialtelecomCz	18,769	34,014	11,084	**1,393**
Ion	15,376	25,261	12,954	**1,387**
Deltacom	12,544	25,135	13,029	**1,737**
Interoute	11,881	14,182	8,346	**710**
Uninett2010	5,329	8,891	3,626	**868**

Table 6: Number of FIB entries for NCFlow vs. edge-based formulations (e.g., Fleischer-Edge), path-based formulations using Räcke Randomized Routing Trees (SMORE*), and path-based formulations using *k*-shortest paths (PF_4, Fleischer-Path, TEAVAR*) on every topology.

amounts of flow.

Figure 19 further breaks down the aggregate results from Figure 11 across various aspects of interest. In the two left-most columns, we break down the results by different settings of α, which illustrates how NCFlow performs on both under-subscribed ($\alpha = \{1,8\}$) and over-subscribed ($\alpha = \{32,64,128\}$) traffic matrices. In the former case, NCFlow is typically able to fully satisfy the TM's requested demand, thereby matching the total flow allocated by the other methods. At the same time, NCFlow is strictly faster on all TMs, except for those belonging to smaller topologies (e.g., Uninett2010), which we discuss later on. As α increases, so, too, does NCFlow's runtime advantage; however, this does come at the cost of the total flow allocated. For example, when $\alpha = 32$, we see many instances where NCFlow is $> 100\times$ faster than PF_4, but allocates 75% of PF_4's total flow in the worst case. This effect becomes more evident for the largest settings of α: here, the speedups are $> 1000\times$, but more flow is sacrificed for some TMs. This behavior occurs perhaps because, as the traffic volume increases and the topology becomes more congested, paths that are not allowed by NCFlow's scheme become more critical for maximizing the total flow.

In the middle two columns, we break down the results by traffic model. NCFlow tends to perform best when demands are highly concentrated within clusters. In the bottom middle plot (Poisson, $\delta \to 0$), we see that NCFlow allocates $> 90\%$ of PF_4's total flow for almost every TM, while still achieving speedups $> 100\times$. Recall that as $\delta \to 0$ in the Poisson traffic model, the traffic volume *between* clusters decreases, thus generating concentrated demands. In contrast, when $\delta \to 1$, demands are less concentrated, which leads to worse performance for NCFlow in terms of total flow, but not in terms of runtime.

Finally, in the two right-most columns, we break down the results by topology size. On Uninett2010, the smallest topology in our evaluation set, NCFlow's trade-off between total flow and runtime is not much better than the other baselines, particularly Fleischer-Edge.

As the topology size increases, NCFlow's advantage becomes more apparent. On Colt, NCFlow offers faster runtimes and sacrifices little flow, no more than 10% less than PF_4. On PrivateSmall and Kdl, NCFlow's speedup increases even more: $> 100\times$ faster than PF_4 on the majority of cases on Kdl. But flow is sacrificed, particularly for large values of α. However, NCFlow's trade-off is still favorable compared to other methods: for Kdl, we see multiple instances where NCFlow achieves $1{,}000\times$ speedups at only a 20% reduction in flow. For PrivateLarge, we see both the biggest speedups and the smallest fraction of total flow relative to PF_4. As previously discussed, the outlier coincides with a highly oversubscribed TM ($\alpha = 128$). When we move to other regimes on PrivateLarge, NCFlow's performance improves: on 31 of the 400 TMs with $\alpha \in \{32,64\}$, NCFlow is $> 1{,}000\times$ faster than PF_4 while achieving $> 80\%$ of PF_4's total flow.

In summary, we can see in this panel of CDF plots where NCFlow's strengths lie: on (1) large topologies, and (2) TMs with moderate demand volumes that are highly concentrated within the topology.

G.2 Alternate clustering methods

For each topology, we evaluate the three different clustering techniques mentioned in §3.4; on each topology we ask each technique to compute the number of clusters listed in Table 5. Figure 21 shows CDFs of the ratio of total flow and latency speed-up of a clustering technique relative to that achieved by using FMPartitioning; thus values to the left of $x = 1$ indicate worse performance compared to FMPartitioning while those on the right indicate better performance. The figure shows that clusters discovered by FM partitioning almost always let NCFlow carry more flow (red lines); using either spectral clustering or leader election leads to a noticeably smaller allocation in about 20% and 40% of the cases. The figure shows a less clear-cut separation on latency speed-up; clusters discovered by leader election offer more speedup in over 30% of the experiments. Overall, we see that FMPartitioning performs better on average but not in all cases.

G.3 Effect on path latency

Figure 22 shows a CDF of the *normalized path latency* for de-

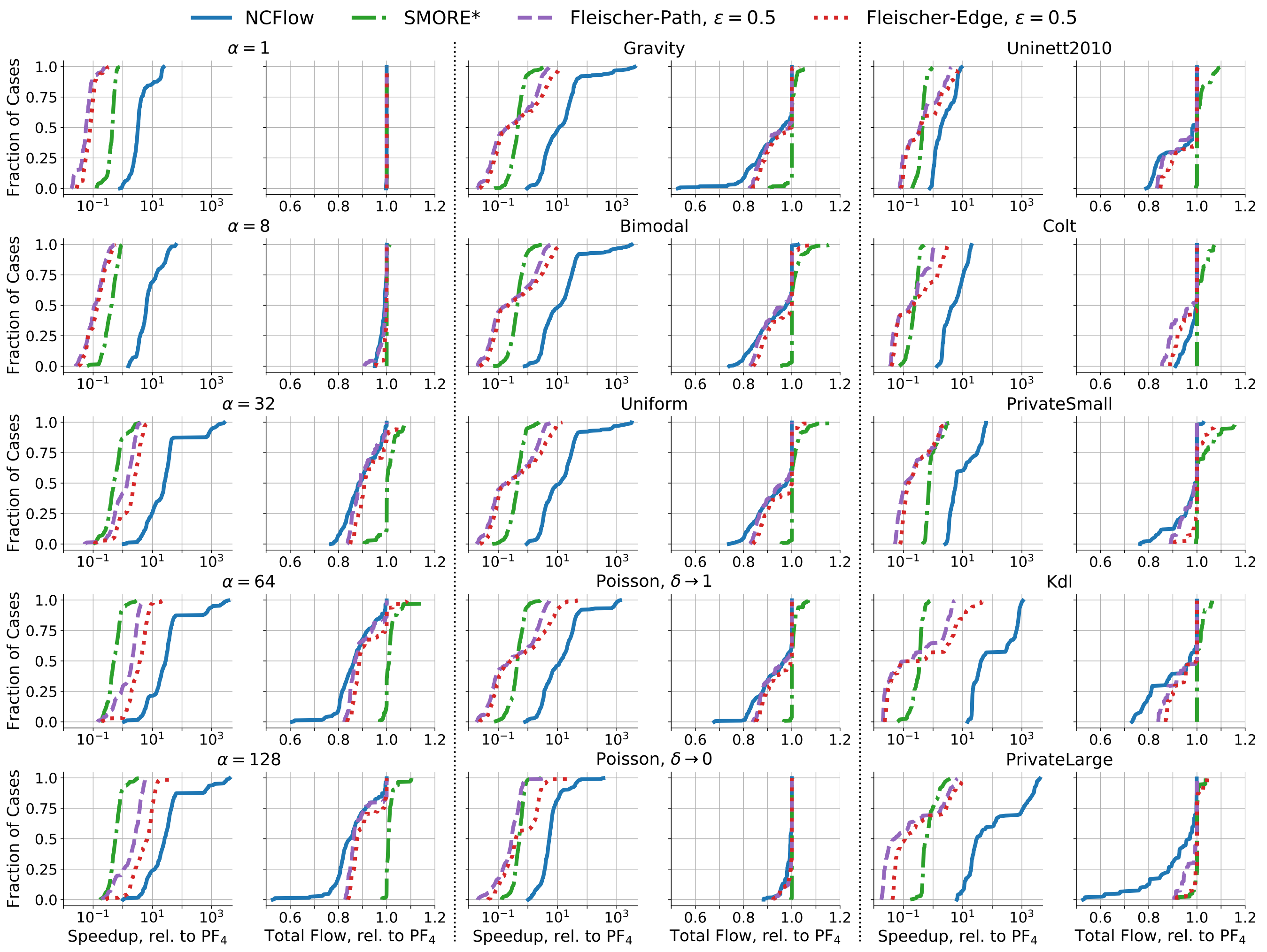


Figure 19: A breakdown of the experimental results from Figure 11 along various dimensions of interest: scale factor, traffic model, and topology size. NCFlow excels on large topologies with TMs that have highly concentrated demands.

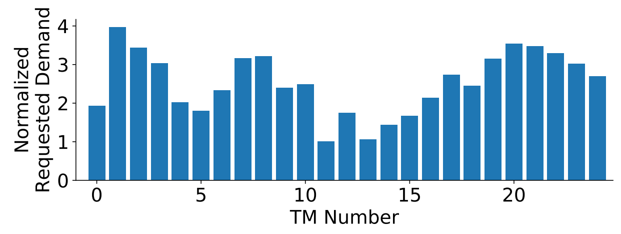


Figure 20: Traffic demand for each traffic matrix used in the demand tracking experiment (see Figure 15) on PrivateLarge. The exact values are not shown for confidentiality reasons.

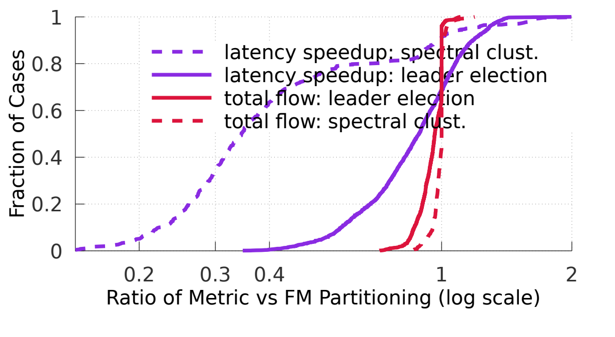


Figure 21: Comparing the total flow allocated and the speedup in computing allocations when clusters are chosen using the three techniques mentioned in §3.4–FM partitioning, spectral clustering and leader election. The default technique used in our evaluation, FM partitioning, generally performs better but not in all cases.

mands[6] under different flow allocations. The figure on the top shows CDFs of the actual normalized path latency. Observe that these distributions are nearly identical. The figure on the bottom shows a CDF of the ratio of normalized latency; we see that roughly 70% of the demands are carried by NCFlow on paths that are at most as long as the paths used by PF_4 (i.e., to the left of x=1). Most of the cases where NCFlow uses

[6]The latency of the paths along which each demand is routed weighted by the fraction of the demand routed along each path. That is, if a demand is divided equally between two paths, the normalized latency will be the average of the path latencies.

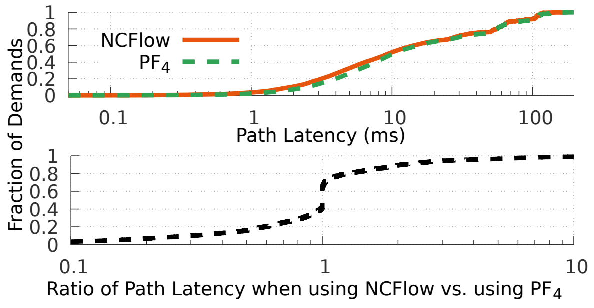
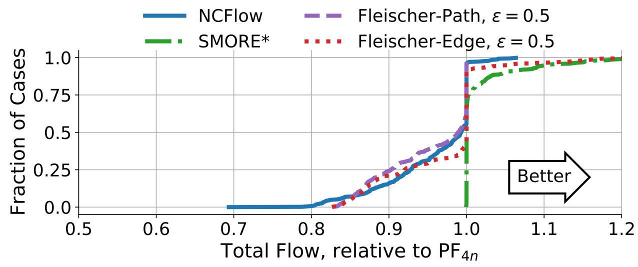
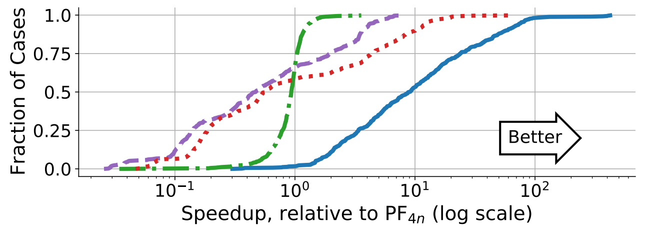


Figure 22: Effect of NCFlow on path latency



(a) CDF of total flow relative to PF$_{4n}$



(b) CDF of speedup relative to PF$_{4n}$

Figure 23: Similarly to Figure 11 all schemes use up to $k = 4$ shortest paths between each pair of nodes except that the paths are chosen *without* ensuring edge disjointness. The figure shows no qualitative difference relative to Figure 11.

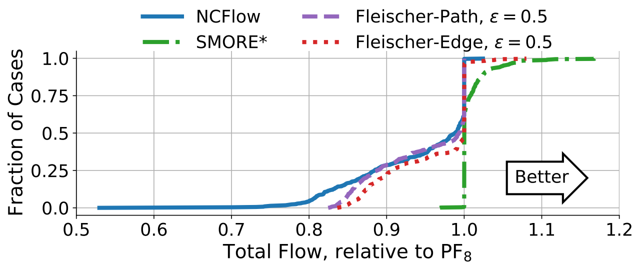
relatively longer paths are for demands that have very small latency paths as illustrated by the top figure.

Note that path latency can be further explicitly controlled in NCFlow by determining which paths can be used or by weighting the objective to prefer shorter paths in the various steps of Figure 6.

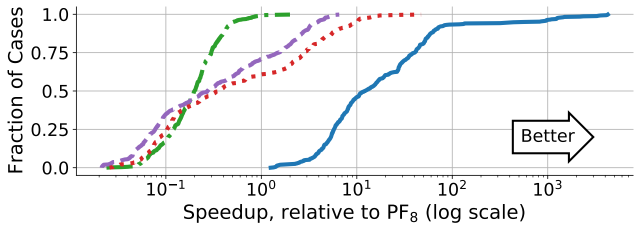
G.4 Alternate path choices

With Figure 23, Figure 24, Figure 25, Figure 26, Figure 27 we evaluate different numbers of paths between node pairs chosen with or without edge disjointness. PF$_k$ refers to path formulation with k shortest paths chosen using edge disjointness and PF$_{kn}$ indicates paths chosen without edge disjointness. Comparing these figures with Figure 11, we note that NCFlow's improvements over baselines hold across different path choices.

Note that Figure 26 and Figure 27 are missing some of the larger topologies listed in Table 5 for some of the baseline schemes because the baselines ran out of memory (we used a server with up to 3TB of memory) or raised some other exception.

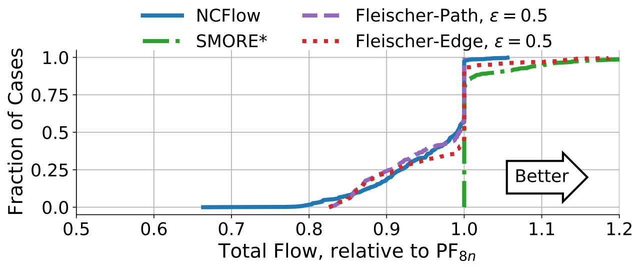


(a) CDF of total flow relative to PF$_8$

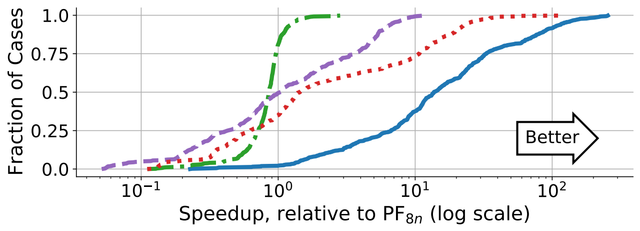


(b) CDF of speedup relative to PF$_8$

Figure 24: Similar to Figure 11 except all schemes use up to $k = 8$ shortest paths between each pair of nodes; paths chosen *with* edge disjointness. The figure shows no qualitative difference relative to Figure 11.



(a) CDF of total flow relative to PF$_{8n}$



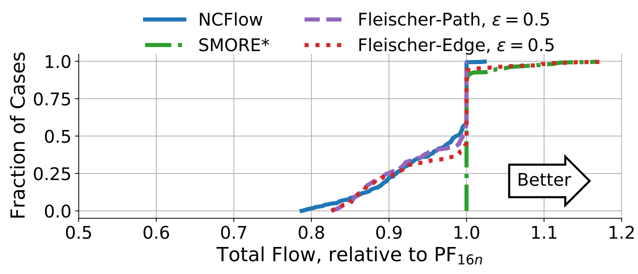
(b) CDF of speedup relative to PF$_{8n}$

Figure 25: Similar to Figure 11 except all schemes use up to $k = 8$ shortest paths between each pair of nodes; paths chosen *without* ensuring edge disjointness. The figure shows no qualitative difference relative to Figure 11.

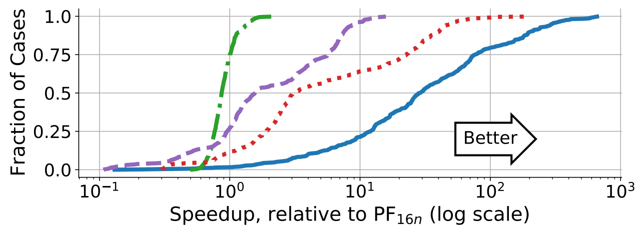
H Illustrative examples

Here, we show some illustrative examples where applying NCFlow using adversarially chosen clusters can lead to sub-optimal flow allocation.

Figure 28 shows a case wherein NCFlow is sub-optimal because the aggregate graph (wherein nodes are clusters) is

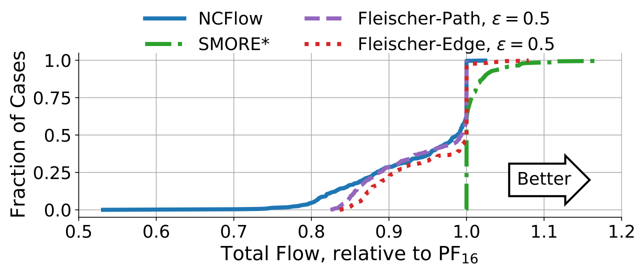


(a) CDF of total flow relative to PF_{16n}

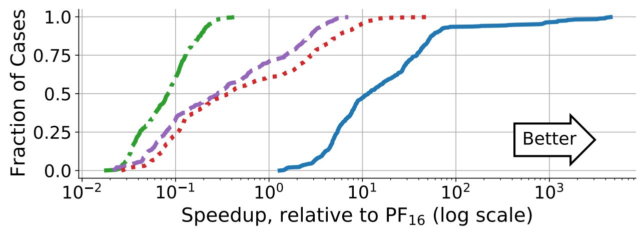


(b) CDF of speedup relative to PF_{16n}

Figure 26: Similar to Figure 11 except all schemes use up to $k = 16$ shortest paths between each pair of nodes; paths chosen *without* ensuring edge disjointness. The figure shows no qualitative difference relative to Figure 11.

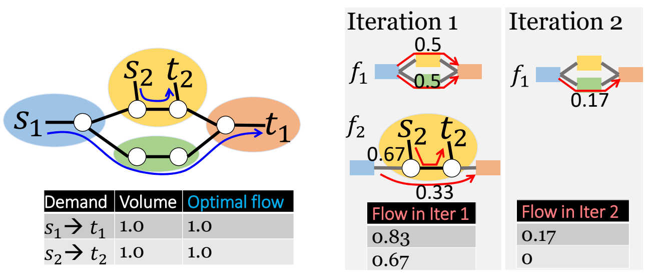


(a) CDF of total flow relative to PF_{16}



(b) CDF of speedup relative to PF_{16}

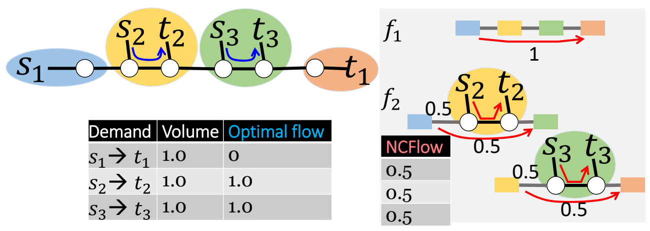
Figure 27: Similar to Figure 11 except all schemes use up to $k = 16$ shortest paths between each pair of nodes; paths chosen *with* ensuring edge disjointness. The figure shows no qualitative difference relative to Figure 11.



Demand	Volume	Optimal flow
$s_1 \rightarrow t_1$	1.0	1.0
$s_2 \rightarrow t_2$	1.0	1.0

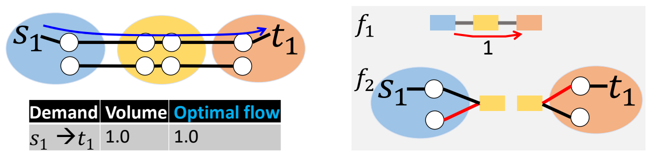
Flow in Iter 1	Flow in Iter 2
0.83	0.17
0.67	0

Figure 28: Sub-optimality of NCFlow when the aggregate graph is not a tree.



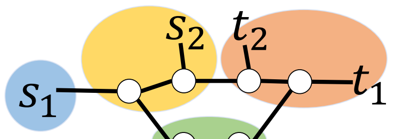
Demand	Volume	Optimal flow	NCFlow
$s_1 \rightarrow t_1$	1.0	0	0.5
$s_2 \rightarrow t_2$	1.0	1.0	0.5
$s_3 \rightarrow t_3$	1.0	1.0	0.5

Figure 29: Sub-optimality of NCFlow when demands cannot be fully satisfied.

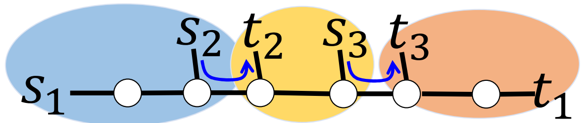


Demand	Volume	Optimal flow
$s_1 \rightarrow t_1$	1.0	1.0

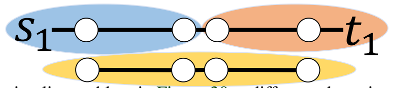
Figure 30: Sub-optimality of NCFlow when there are multiple-edges between pairs of clusters.



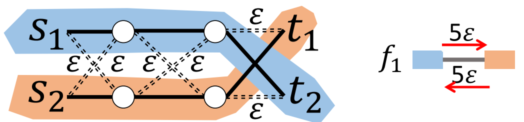
(a) For the suboptimality problem in Figure 28, a different clustering choice that leads to optimal flow allocation with NCFlow.



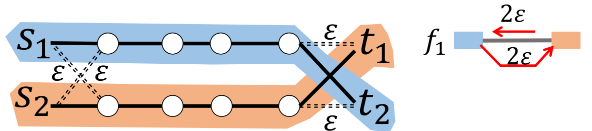
(b) For the suboptimality problem in Figure 29, a different clustering choice that leads to optimal flow allocation with NCFlow.



(c) For the suboptimality problem in Figure 30, a different clustering choice that leads to optimal flow allocation with NCFlow.



(d) For the disagreement problem in Figure 8a, a different clustering choice that does not lead to such a disagreement.



(e) For the disagreement problem in Figure 6, a different clustering choice that does not lead to such a disagreement.

Figure 31: Alternate clustering choices that fix suboptimality concerns and disagreements.

not a tree. The network topology and optimal allocations are shown in the graph on the left; assume each link has a unit capacity. With NCFlow, as shown in the figures on the right, MaxAggFlow can route the flow from s_1 to t_1 on either the top or the bottom path or divide between the two paths in some proportion; note that MaxAggFlow is not aware of demands that are local to a cluster (such as the flow from s_2 to t_2). Whenever MaxAggFlow assigns non-zero flow for the $s_1 \to t_1$ demand on the top path, NCFlow will be sub-optimal because then the other demand cannot be fully satisfied when MaxClusterFlow executes later on the yellow cluster. Any unsatisfied volume for $s_1 \to t_1$ can be routed on the bottom path in a later iteration but the flow for $s_2 \to t_2$ will not increase since the links that demand can use are fully utilized in the first iteration. The root of the problem here is that MaxAggFlow allocates traffic over multiple paths without being aware of the demands within clusters.

Figure 29 shows a case wherein NCFlow is sub-optimal when demands cannot be fully satisfied. As above, the topology and optimal allocations are shown on the left. Also, as above, the root of the issue here is that MaxAggFlow allocates the cross-cluster flow on the aggregate graph without being aware of the demands within clusters. As shown, subsequently, MaxClusterFlow will under-allocate flow for the local demands even though total flow would be larger if the local demands are fully satisfied.

Reordering the sub-problems, i.e., executing MaxClusterFlow before MaxAggFlow, may appear promising based on these examples but simple counter-examples exist even for such a reordered solution. The underlying cause of sub-optimality is not the order in which the global and local solutions are computed but rather that the optimal flow allocation requires *jointly solving* these problems.

Figure 30 shows a case wherein NCFlow can be sub-optimal when multiple edges connect clusters. As above, each unmarked link has unit capacity and the optimal allocations are shown in blue. Recall that NCFlow uses exactly one edge between each pair of clusters per iteration to avoid disagreements. There are two edges between each cluster but among the four possible crossing edge choices in an iteration, exactly one choice can carry non-trivial amount of flow (the top edge for each cluster pair). If that choice is somehow not picked, as shown marked in red on the right in Figure 30, NCFlow will not satisfy the demand. Simply increasing the number of iterations may not suffice either since the number of edge choices can be large, depending on the path lengths on the aggregate graph and on the number of edges between clusters.

As noted previously, the above examples are in part due to poor cluster choices; Figure 31 shows different cluster choices for these examples under which NCFlow will lead to optimal flow allocation.

I Optimality gap

$u_e, v_e \in \mathcal{V}$	Edge $e \in \mathcal{E}$ goes from node u_e to node v_e
$m_u, \forall u \in \mathcal{V}$	m_u denotes the cluster containing node u. Note that $m_u \in \mathcal{V}_{\text{agg}}$ (i.e., the cluster is a node on the aggregate graph) and $u \in \mathcal{V}_{m_u}$ (i.e., the node u belongs in the restricted graph for the m_u'th cluster)
$\forall k \in \mathcal{D}, m_{s_k} \neq m_{t_k}, x \in \mathcal{V}_{\text{agg}}$	
OutNodes(x,k)	The nodes in cluster x that can carry flow of demand k *out to* some other cluster, i.e., $\{u \mid m_u = x, \exists v \in \mathcal{V}, p \in \mathcal{P}_{\text{agg},K_{s_k t_k}}$ s. t. $m_v = y, (x,y) \in p, (u,v) \in \mathcal{E}\}$
InNodes(x,k)	The nodes in cluster x that can carry flow of demand k *into* cluster x, i.e., $\{u \mid m_u = x, \exists v \in \mathcal{V}, p \in \mathcal{P}_{\text{agg},K_{s_k t_k}}$ s. t. $m_v = y, (y,x) \in p, (v,u) \in \mathcal{E}\}$

Table 7: Additional notation for optimality gap; builds on top of notation from Table 2 and Table 3.

$$\begin{aligned}
&\text{MaxEdgeFlow}(\mathcal{V},\mathcal{E},\mathcal{D}) \triangleq \arg\max_{\mathbf{f}} \sum_{k\in\mathcal{D}} f_k \text{ s.t.} && && (6)\\
&\mathbf{f} = \{f_{ke} \mid \forall k \in \mathcal{D}, e \in \mathcal{E}\} && \text{and} &&\\
&f_{ke} \geq 0 && \forall e \in \mathcal{E}, k \in \mathcal{D} && \text{(non-negative flow)}\\
&f_k \leq d_k, && \forall k \in \mathcal{D} && \text{(below volume)}\\
&\sum_{\forall k,e} f_{ke} \leq c_e, && \forall e \in \mathcal{E} && \text{(below capacity)}\\
&\sum_{e,u_e=u} f_{ke} - \sum_{e,v_e=u} f_{ke} = \begin{cases} f_k & \text{if } u = s_k \\ -f_k & \text{if } u = t_k \\ 0 & \text{o/w.} \end{cases} && \forall k \in \mathcal{D}, u \in \mathcal{V} && \text{(flow cnsrvtn.)}
\end{aligned}$$

I.1 Optimal MaxEdgeFlow

The optimal flow allocation algorithm, in terms of carrying the maximum amount of flow possible on a network, is as shown in Equation 6. We will call this the EF, short for Max**E**dge**F**low. Some additional notation is in Table 7. Observe that, in this formulation, any demand can be allocated on any edge (the variable f_{ke}) as long as flow conservation holds (the longer equation at the bottom). As noted in §2, this *edge-form* of the problem carries the maximal amount of flow but has a high computation time and requires a large number of forwarding entries at switches (one rule per nodepair at each node).

I.2 Edge flow with cluster constraints

Relative to the optimal MaxEdgeFlow, we first ask how much flow will be lost by using clusters. To compute this value, we add to MaxEdgeFlow the constraint shown in Equation 7. Specifically, demands whose source and target are in the same cluster can only use edges within the cluster. However, as above, paths remain otherwise unconstrained.

$$f_{ke} = 0 \;\; \forall e, u_e \notin \mathcal{V}_x \text{ or } v_e \notin \mathcal{V}_x, \text{ if } m_{s_k} = m_{t_k} = x. \qquad (7)$$

We will call this optimization problem EF with cluster constraints.

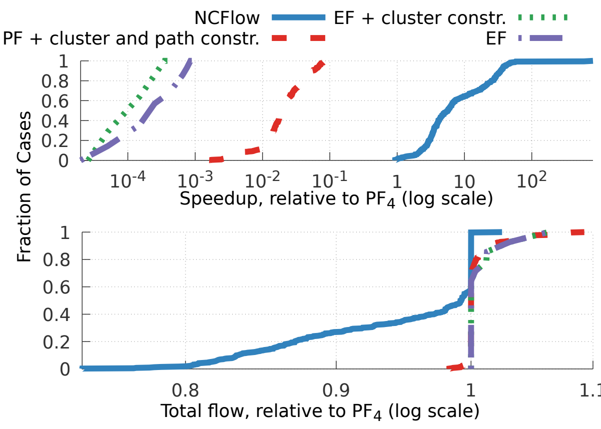


Figure 32: Comparing flow allocated by NCFlow with the best possible flows

I.3 Path form with cluster and path constraints

Next, we ask how much flow will be lost when using the clusters as well as the given set of paths within and between clusters? Computing this value is somewhat more complex because we have to stitch together the flow carried on paths within each cluster with the flow on the edges between clusters while also ensuring that flow follow the chosen paths on the aggregate graph (where clusters are nodes). For reference, we write this out in Equation 8.

In more detail, this optimization problem, as shown in Equation 8, has three classes of decision variables – f^p_K, f^p_k, f_{ke} – which respectively are the flow allocated to a bundled demand on a path on the aggregate graph, the flow allocated to a demand on a path within a cluster and the flow allocated to a demand on a crossing edge between clusters.

Equation 9 computes the net flow for each demand k which for the case of a demand whose source and target are in the same cluster is the sum of flow carried on all intra-cluster paths. For demands whose source and target are in different clusters, the net flow is the flow from the demand's source to all of the nodes in the source's cluster that connect with other clusters as well as the flow to the demand's target from all of the nodes in the target's cluster that connect with other clusters.

For flow conservation, consider Equation 11 which ensures that all of the flow leaving at a node u for a demand k on crossing edges to other clusters equals the flow that comes into the node u either from the source of the demand (if the source is within its cluster) or from all of the nodes in u's cluster that can receive flow for demand k from other clusters–$\mathsf{InNodes}(m_u, k)$. Equation 12 considers the converse case for demands that leave at a node. Finally, Equation 13 relates the total flow between a pair of clusters x, y on the crossing edges between these clusters with the flow along paths on the aggregate graph that contain the edge (x, y). We will call this optimization problem PF with cluster and path constraints.

Note that the above constraints naturally lead to a reduction in forwarding table size as discussed in §3.5. However, it is not clear how much less flow these constraints allow for relative to the optimal EF. Moreover, since this optimization has more variables (and constraints) than PF_4 (see Equation 2), it can take longer to compute and may not be practically useful. We use this optimization problem to discern how much flow is lost by the constraints used in NCFlow (restricting to clusters and paths) relative to the flow that is lost due to the heuristic allocation process described in §3.

I.4 Experimental results

Our results are in Figure 32; the baseline is PF_4 and the figures plot CDFs of total flow and latency speedup for many topologies and traffic demands. Note that using the edge formulation (purple dash-dots) often leads to substantially more flow being allocated compared to PF_4; however, as the figure on the top shows, edge formulation is a more complex problem that takes longer to run (over 1000× longer).

Adding the clustering constraint to edge formulation has an un-noticeable effect on the flow allocation (green dashes). Note that we use clusters computed using FMPartitioning for all topologies.

Constraining the path formulation using both the given clusters and the given paths (between clusters and within each cluster), as shown with the red dash line, allocates much more flow than PF_4 and not much less than is allocated in edge formulation. Thus, empirically, constraining flow allocation to traverse the chosen clusters and paths does not limit the flow that can be allocated. The figure also shows that computing the optimal flow given clusters and paths takes longer than PF_4 (roughly 10× – 100× longer). Thus, NCFlow offers a heuristic which finishes substantially faster than PF_4.

To sum up, our two main contributions are: (1) constraining flow allocations to use specific clusters and paths which reduces the number of forwarding table entries needed without affecting the flow that can be allocated and (2) a heuristic that computes flow allocations quickly given this constraint but can under-allocate flow. We believe that future work can improve the heuristic to reduce the flow loss further.

$$\text{MaxClusterPathFlow}(\mathcal{V},\mathcal{E},\mathcal{D},\mathcal{P}) \triangleq \arg\max_{\mathbf{f}} \sum_{k\in\mathcal{D}} f_k \qquad \text{s.t.} \tag{8}$$

$$\begin{aligned}
\mathbf{f} = \big\{ & f_K^p \mid \forall K \in \mathcal{D}_{\text{agg}}, p \in \mathcal{P}_{\text{agg}}, && \text{(flow on inter-cluster paths)}\\
& f_k^p \mid \forall k \in \mathcal{D}, p \in \mathcal{P}, && \text{(flow on intra-cluster paths)}\\
& f_{ke} \mid \forall k \in \mathcal{D}, e \in \mathcal{E}, m_{u_e} \neq m_{v_e} \text{(flow on edges between clusters)}\big\}
\end{aligned}$$

and

$$f_k = \begin{cases} \sum\limits_{p\in\mathcal{P}_{s_k,t_k}} f_k^p & \text{if } m_{s_k} = m_{t_k} \quad \text{(flow within a cluster)}\\ \sum\limits_{t\in\text{OutNodes}_{(m_{s_k},k)}} \sum\limits_{p\in\mathcal{P}_{s_k,t}} f_k^p & \text{if } m_{s_k} \neq m_{t_k} \quad \text{(flow from source to outnodes)}\\ \sum\limits_{s\in\text{InNodes}_{(m_{t_k},k)}} \sum\limits_{p\in\mathcal{P}_{s,t_k}} f_k^p & \text{if } m_{s_k} \neq m_{t_k} \quad \text{(flow to target from innodes)} \end{cases} \qquad \forall k \in \mathcal{D}\text{(net flow)} \tag{9}$$

$$f_k \leq d_k \text{(flow below volume)} \qquad \forall k \in \mathcal{D}$$

$$c_e \geq \begin{cases} \sum\limits_{k\in\mathcal{D}} \sum\limits_{p\in\mathcal{P},\ p\ni e} f_k^p & \text{if } m_{u_e} = m_{v_e} \quad \text{(intra-cluster edges; note: } k \text{ goes over all demands)}\\ \sum\limits_{k\in\mathcal{D}} f_{ke} & \text{otherwise} \quad \text{(inter-cluster edges)} \end{cases} \qquad \forall e \in \mathcal{E}, \tag{10}$$

$$\sum_{e\in\mathcal{E}\mid u_e=u,\ m_{u_e}\neq m_{v_e}} f_{ke} = \begin{cases} \sum\limits_{p\in\mathcal{P}_{s_k u}} f_k^p & \text{if } m_u = m_{s_k} \quad \text{(at cluster } m_u\text{, flow from } s_k \text{ to } u)\\ \sum\limits_{v\in\text{InNodes}(m_u,k)} \sum\limits_{p\in\mathcal{P}_{v,u}} f_k^p & \text{otherwise} \quad \text{(at cluster } m_u\text{, flow from all InNodes to } u) \end{cases} \qquad \forall u \in \mathcal{V}, k \in \mathcal{D} \tag{11}$$

$$\sum_{e\in\mathcal{E}\mid v_e=u,\ m_{u_e}\neq m_{v_e}} f_{ke} = \begin{cases} \sum\limits_{p\in\mathcal{P}_{u,t_k}} f_k^p & \text{if } m_u = m_{t_k} \quad \text{(at cluster } m_u\text{, flow from } u \textbf{ to } t_k)\\ \sum\limits_{v\in\text{OutNodes}(m_u,k)} \sum\limits_{p\in\mathcal{P}_{u,v}} f_k^p & \text{otherwise} \quad \text{(at cluster } m_u\text{, flow from } u \text{ to all OutNodes)} \end{cases} \qquad \forall u \in \mathcal{V}, k \in \mathcal{D} \tag{12}$$

$$\sum_{p\in\mathcal{P}_{\text{agg}}\mid(x,y)\in p} f_K^p = \sum_{e\mid m_{u_e}=x,\ m_{v_e}=y,\ k\in K} f_{ke} \qquad \forall K \in \mathcal{D}_{\text{agg}},\ x,\ y \in \mathcal{V}_{\text{agg}} \quad \text{(flow b/w clusters = flow on inter-cluster path)} \tag{13}$$

Cost-Effective Cloud Edge Traffic Engineering with CASCARA

Rachee Singh Sharad Agarwal Matt Calder Paramvir Bahl
Microsoft

Abstract

Inter-domain bandwidth costs comprise a significant amount of the operating expenditure of cloud providers. Traffic engineering systems at the cloud edge must strike a fine balance between minimizing costs and maintaining the latency expected by clients. The nature of this tradeoff is complex due to *non-linear* pricing schemes prevalent in the market for inter-domain bandwidth. We quantify this tradeoff and uncover several key insights from the link-utilization between a large cloud provider and Internet service providers. Based on these insights, we propose CASCARA, a cloud edge traffic engineering framework to optimize inter-domain bandwidth allocations with non-linear pricing schemes. CASCARA leverages the abundance of *latency-equivalent* peer links on the cloud edge to minimize costs without impacting latency significantly. Extensive evaluation on production traffic demands of a commercial cloud provider shows that CASCARA saves 11–50% in bandwidth costs per cloud PoP, while bounding the increase in client latency by 3 milliseconds[1].

1 Introduction

Cloud wide-area networks (WANs) play a key role in enabling high performance applications on the Internet. The rapid rise in traffic demands from cloud networks has led to widespread adoption of centralized, software-defined traffic engineering (TE) systems by Google [19] and Microsoft [17] to maximize traffic flow *within* the cloud network.

In the quest to overcome BGP's shortcomings, recent efforts have focused on engineering *inter*-domain traffic, which is exchanged between the cloud WAN and other networks on the Internet [27, 33]. These systems can override BGP's best-path selection, to steer egress traffic to better performing next-hops. However, this focus on performance overlooks a crucial operating expenditure of cloud providers: the *cost* of inter-domain traffic determined by complex pricing schemes. While the prices of inter-domain bandwidth have declined in the past decade, the decrease has been outpaced by exponential growth in demand [29] from cloud networks serving high-definition video, music and gaming content. In fact, the inter-domain bandwidth costs incurred by the cloud provider we analyze increased by 40% in the March 2020 billing cycle as a consequence of the increase in demand fueled by work from home guidelines in various parts of the world. [2]

[1]Code and experiments at: http://cascara-network.github.io.

[2]We do not disclose the fraction of total cloud operation expenditure contributed by inter-domain bandwidth costs due to confidentiality reasons.

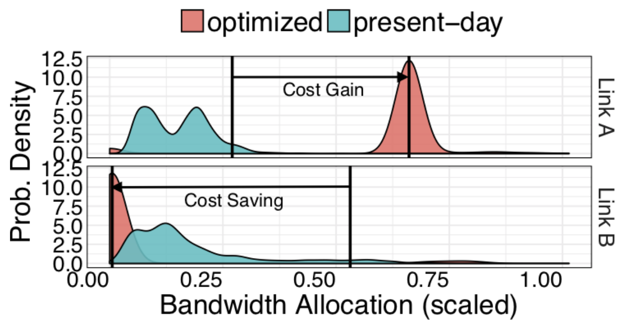


Figure 1: Present-day and CASCARA-optimized bandwidth allocation distributions for one week, across a pair of links between a large cloud provider and tier-1 North American ISPs. Costs depend on the 95th-percentile of the allocation distributions (vertical lines). CASCARA-optimized allocations reduce total costs by 35% over the present-day allocations while satisfying the same demand.

In this work, we show that recent increases in interconnection and infrastructure scale enable significant potential to reduce the costs of inter-domain traffic. These advances include the deployment of several new cloud points of presence (PoP) near clients and direct peering with an increasing fraction of the Internet's autonomous systems [5]. As a result, most clients are reachable over several short and *latency-equivalent* paths from the cloud provider [26]. We illustrate the cost saving potential due to latency-equivalent links with an example in Figure 1. We plot the distributions of bandwidth allocated over one week to links A and B, which connect a large cloud provider to tier-1 North American ISPs. Both links are located at inter-connection points within 30 km of each other, and offer comparable latency due to their geographical proximity. In this example, the bandwidth price per Mbps of Link B is 33% higher than that of Link A. Link costs are a function of the 95th percentile of the bandwidth allocations to each link. The *present-day* allocations (in blue) represent the current bandwidth assigned to the links by the cloud provider under study. In contrast, the CASCARA-*optimized* allocations (in red) meet the same or higher demand as the present-day allocations, while reducing total bandwidth costs by 35%.

Bandwidth allocations at the cloud edge impact both the client latency and inter-domain bandwidth costs to the cloud provider. At one extreme, traffic allocations may disregard the latency impact to drive bandwidth costs to near-zero while at the other extreme, allocations may incur very high bandwidth costs by greedily assigning traffic to the lowest latency peers. Balancing this *cost-latency tradeoff* is central to our work. However, it is made challenging by industry-standard pricing schemes that use 95th percentile of the bandwidth distribution over monthly time-periods. Complex relationships between bandwidth allocations, costs and client latency lead to computationally hard optimization problems.

We tackle these challenges by first analyzing the utilization of edge links from a large commercial cloud provider. We find that the majority of traffic from the cloud is exchanged with transit ISPs, with outbound traffic being twice in volume compared to inbound traffic. Thus, outbound traffic to transit ISPs dominates the inter-domain bandwidth costs of the cloud. Three such North American ISPs incur a majority of the total expenditure on inter-domain bandwidth in the continent (§3). Using these insights, we make three main contributions:

1. Quantify the opportunity of saving bandwidth cost. We formulate cloud edge TE as an optimization with the goal of minimizing percentile bandwidth costs. Despite the non-convex nature of the objective, the optimization is tractable in engineering outbound traffic to peer links with only the small number of ISPs that contribute majority of the costs. We show that cost-optimal allocations can **save up to 65% of the cloud provider's inter-domain bandwidth costs**, quantifying the upper bound on savings (§3) and offering a significant improvement over related approaches in [12, 20, 35].

2. Practical and cost-efficient online edge TE. Since optimizing percentile costs is NP-Hard [20], finding optimal solutions can take several hours. We leverage insights from the offline optimal solution to design an efficient, heuristic-based online TE framework, CASCARA. CASCARA leverages the cloud provider's rich diversity of latency-equivalent BGP peers to offer cheaper options to outbound traffic. Through extensive experiments we demonstrate that CASCARA provides near-optimal cost saving in practice and can be deployed **safely** and **incrementally** in cloud WANs (§4).

3. Flexibility to balance the cost-latency tradeoff. CASCARA incorporates the latency of primary and alternate peer paths from the cloud [4, 27] to strike a balance between bandwidth cost savings and client latency. CASCARA provides the flexibility to pick the operating point on this tradeoff and finds allocations that bound the increase in client latency by 3 ms while saving 11-50% of bandwidth costs per cloud PoP (§5).

Client latency requirements vary based on the types of application traffic, *e.g.,* software updates and large file transfers are more delay tolerant than live video. In fact, majority of all outbound traffic from the cloud provider is marked as best-effort, making it tolerant to small changes in latency. We conclude this study by discussing the generalizability of our results, the implications of CASCARA on peering contracts and bandwidth pricing models on the Internet (§6).

2 CASCARA controller overview

CASCARA's goal is to engineer outbound traffic allocations from the cloud edge to achieve near-optimal saving in inter-domain bandwidth costs. It does so by providing operational safety knobs to the operator: configurable variation in the traffic allocations to peer links, incremental deployability and bounded impact on client latency. Figure 2 shows the different components of CASCARA. At the core is the CASCARA WAN controller that allocates cost-optimized flow to outbound peer links of the cloud network.

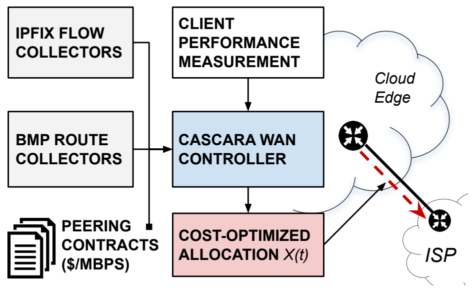

IPFIX FLOW COLLECTORS
CLIENT PERFORMANCE MEASUREMENT
Cloud Edge
BMP ROUTE COLLECTORS
CASCARA WAN CONTROLLER
PEERING CONTRACTS ($/MBPS)
COST-OPTIMIZED ALLOCATION X(t)
ISP

Figure 2: The design of CASCARA.

IPFIX Flow Collectors. We feed IP Flow Information Export (IPFIX) [31] logs to CASCARA to infer the utilization of edge links of the cloud network in five minute intervals of the billing cycle. These allocations to peer links are used both for offline cost analysis (§3) and online allocation to meet demands by CASCARA (§4 and §5).

BMP Route Collectors. We gather route announcements made by BGP peers at points of presence (PoP) of the cloud provider using BGP Monitoring Protocol (BMP) collectors. These routes inform CASCARA of the choices of peer links for outbound demand towards clients.

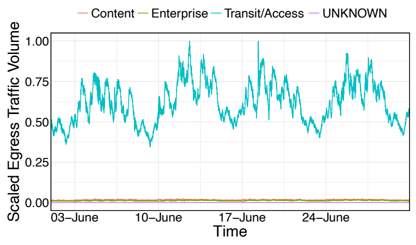
Peering Contracts. We feed the billing models and peering rates for all BGP peers of the cloud provider to CASCARA. Since peering rates remain stable over short durations of time, we use snapshot of this information from June 2019.

Client latency measurements. CASCARA makes latency-aware decisions limiting the impact of outbound traffic allocation on client latency. We feed CASCARA the median latency to all clients of the cloud provider over both the primary and alternate BGP paths at the PoPs.

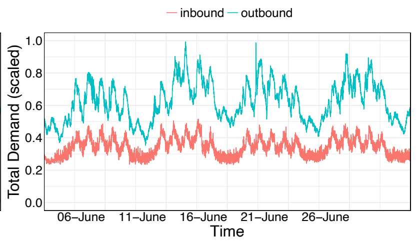
Cloud providers have developed software-defined edges for fine-grained control of outbound path selection from their networks [27, 33]. These systems provide the infrastructure to steer outbound traffic to desired peer paths. The CASCARA controller allocates flow to peer links in every 5 minute interval and can leverage the software-defined edge to optimize the inter-domain bandwidth costs. We first quantify the potential of bandwidth cost saving in a large cloud provider (§3), then develop an efficient, online and near-optimal algorithm for CASCARA to realize the saving potential (§4). Finally, we put CASCARA to test with realistic client performance and route availability constraints in §5.

3 Quantifying the Opportunity

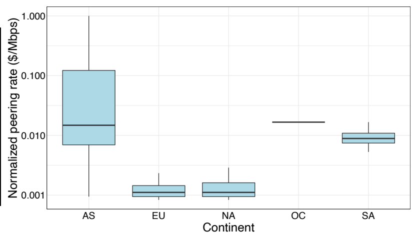
Cloud networks occupy a central position in the Internet ecosystem due to the large volume and variety of popular content they serve to users. To make this possible, cloud providers



(a) Outbound traffic types.



(b) Inbound vs. Outbound traffic.



(c) Norm. peering rates by continent.

Figure 3: (a) Outbound traffic from a large cloud provider towards BGP peers of different types; majority of outbound traffic is towards transit/access networks. (b) The distribution of inbound vs. outbound traffic volume from the cloud network. (c) Large differences in the cost per unit bandwidth in different parts of the world *e.g.,* the median peering cost in Asia is over 10X the median peering cost in North America.

peer with a large number of networks or Autonomous Systems (ASes) on the Internet, including transit ISPs and eyeball networks. The cloud provider we analyze has over 7,000 BGP peers, including transit networks, access networks, content providers and Internet Exchange Points (IXPs). These links span over one hundred geographical locations, collectively carrying terabits of traffic per second. We analyze the utilization and bandwidth costs incurred at the peering edge of the commercial cloud provider using IPFIX flow records collected from June 2018 to July 2019. Aggregated across all edge links, Figure 3a shows the outbound traffic volume per five minute interval from the cloud towards transit/access networks, cloud providers and enterprise networks, categorized by CAIDA's AS types classification [3].

3.1 Dominant contributors to bandwidth cost

A BGP peer of the cloud network charges for the traffic exchanged between them according to the billing model negotiated in their *peering contract*. There are three billing models for inter-domain traffic prevalent on the Internet today: (1) Settlement-free (2) Per-port and (3) Per-Megabit [9]. Settlement-free peers (SFP) agree to exchange traffic with each other at no cost (*e.g.,* between cloud providers). In per-port peering, a peer bills another for each network port used at their facility (*e.g.,* connections at IXPs). Per-Megabit is a utilization-based model where a network charges its peer based on the utilization of the link between them over monthly billing cycles. There can be a commit clause in this contract *i.e.,* regardless of the actual usage, the customer *commits* to pay at least some pre-arranged amount to the provider.

Utilization-based, per-megabit billing is the industry standard for paid peer and transit ISP contracts and it is the focus of our work. Our goal is to minimize bandwidth costs accrued on peering links billed by their utilization. To translate network utilization into the corresponding inter-domain bandwidth cost, ISPs measure the average utilization of peering links in five minute intervals in both inbound and outbound directions. Let the edge link from peer p_1 to peer p_2 have average outbound utilizations of $B = \{B_1, B_2, .., B_n\}$ megabits in 5-minute intervals of a given month. Let B_{out} be the 95^{th} percentile of the outbound utilizations, B. Similarly, B_{in} is the 95^{th} percentile of average inbound utilizations of the $p_1 - p_2$ link. The link cost for a billing cycle is given by, $B = c_i \cdot \text{MAX}\{B_{out}, B_{in}\}$, where c_i is the peering rate negotiated by p_1 and p_2 as part of their peering agreement. This model of billing bandwidth, also called *burstable billing*, has evolved as an industry standard on the Internet [9].

Bulk of the traffic is exchanged with Transit/Access ISPs. The large majority of traffic at the cloud edge is outbound to Transit/Access networks (Figure 3a). Therefore, traffic exchanged with transit ISPs is the main contributor to bandwidth costs incurred by the cloud provider.

Outbound traffic is twice the inbound. For the cloud WAN, outbound traffic volume is nearly twice the inbound (Figure 3b), highlighting that the cost computation based on link utilizations can be simplified to $c_i \cdot B_{out}$ for clouds networks.

Links with only three ISPs contribute majority of costs. Due to the large variance in peering rates (seen in Figure 3c) and skewed distribution of traffic towards a few large ISPs in the North American region of the cloud, edge links to three large networks incur a majority of the total spend on inter-domain bandwidth in North America.

3.2 Optimal inter-domain bandwidth costs

In this section we formalize the task of optimizing inter-domain bandwidth costs of a cloud network. As outbound traffic to paid peers is significantly higher than inbound (Figure 3b), we focus on engineering outbound traffic to minimize the overall inter-domain bandwidth cost. To quantify the potential cost savings, we formulate the offline version of the problem where traffic demands are known in advance.

Let $L = \{l_1, l_2, ..l_m\}$ be the set of all edge links from the WAN. Edge links to the same peer at different points of presence (PoP) are billed individually according to their percentile utilization. Let a five-minute interval in the monthly billing period be t_j where $j \in \{1, 2, .., n\}$. For instance, the month of January has $8,928$ five-minute intervals.

Decision variables. The traffic allocation scheme assigns network flow to peering links in L, for every time slot $t_j, j \in [1,..,n]$. Let x_{ij} be the decision variable, where x_{ij} is the flow assigned to peering link l_i in time slot t_j.

Objective function. The goal of our allocation scheme is to find a traffic assignment to edge links over the entire billing period such that the total inter-domain bandwidth cost is minimized. The cost incurred on peering link l_i is the product of the peering rate (c_i) and the 95th percentile utilization of that link (denoted by z_i). The goal is to minimize the total cost incurred across all links in the WAN:

$$\text{minimize } Z = \sum_{i=1}^{m} c_i \cdot z_i$$

Constraints. The traffic allocations are subject to constraints on link capacities. Since, the offline setting assumes knowledge of traffic demands, the traffic scheme must allocate flow in a way that the egress traffic demand is met in all time slots.

Formulating percentile cost as k-max. The cost function consisting of the sum of 95th percentile utilization of links is non-convex. Previous work has shown that optimizing percentile cost functions is NP-HARD [20]. We later show that techniques from previous work are not effective in saving bandwidth costs of edge links (§4.3). We formulate the exact 95th percentile of traffic allocations as part of the objective function. We note that the 95th percentile of a distribution of n numbers is the same as their **k-max** where $k = n/20$.

Key insight. The key insight of our formulation is that link utilization during 5% of time slots do not contribute to its 95th percentile cost. This means that 5% of time in any billing month is *free* regardless of the traffic it carries. We capture this insight in the optimization formulation using binary integer variables λ_{ij} for each decision variable x_{ij}. λ_{ij}s are also decision variables of the optimization which reflect whether their corresponding x_{ij}s contribute to the link cost. This is expressed with the indicator constraint:

$$(\lambda_{ij} == 0) \implies z_i \geq x_{ij}, \forall i, j \tag{1}$$

We note that only 5% of all $\{x_{i1}, x_{i2}, .., x_{in}\}$ can have their corresponding $\lambda_{ij} = 0$ since we can get away with considering 5% of allocations as *free*. This is expressed using Big-M constraints in the formulation [14]. The minimization objective ensures that of all λ_{ij}s, the ones corresponding to the top $k-1$ of the allocations (x_{ij}) at a link do not contribute to its cost.

Implementation details. Algorithm 1 formulates the traffic cost optimization problem as a Mixed Integer Linear Program (MILP), which is computationally hard to solve. We implement the formulation using the CVX [8] framework and solve it with the commercial optimization solver, GUROBI [15] on a machine with 12 cores and 48 Gb RAM. Our choice of solver is motivated by the computational complexity of Algorithm 1. Commercial solvers like GNU LPK [11] and CPLEX [18] were orders of magnitude slower than GUROBI in solving our formulation.

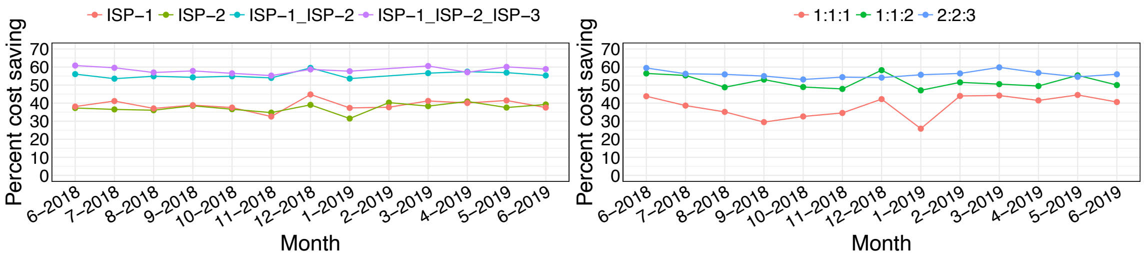
```
Algorithm 1: WAN Egress Traffic Allocation
  Inputs:
        $n$: number of five-minute time slots in a month
        $m$: number of peering links in the WAN
        $l_i$: Peering link $i$
        $C_i$: capacity of peering link $l_i$
        $c_i$: peering rate (USD/Mbps) for link $l_i$
        $d_j$: egress demand from the WAN in time slot $t_j$
        $k = \frac{n}{20}$
        $M$: large integer constant
  Outputs:
        $x_{ij}$: traffic allocation to link $l_i$ in time slot $t_j$
        $\lambda_{ij}$: binary variables that discount top-k $x_{ij}$s
        $z_i$: billable bandwidth on link $l_i$
  Minimize: $\sum_i z_i \cdot c_i$
  subject to:
        $0 \leq x_{ij} \leq C_i, \quad \forall i, \forall j$
        $\sum_i x_{ij} = d_j, \quad \forall j$
        $\sum_j \lambda_{ij} = k - 1, \quad \forall i$
        $z_i > x_{ij} - M \cdot \lambda_{ij}, \quad \forall i, \forall j$
```

3.3 Generalizable and large saving potential

Using the set of peering links (L), peering rates (c_i), link capacities (C_i) and real egress traffic demands (d_j) from a large commercial cloud network, we formulate instances of Alg. 1. The egress traffic demands (d_j) are collected from June 2018 to June 2019 and consist of flow (megabits) that traversed the BGP peering links in each 5-minute interval. Peering rates remain constant during the course of our study. This provides 12 instances of Alg. 1, one for each 1-month billing period. We discuss the implementation details and assumptions in §3.4 and offer a preview of the results here. We compare the cost of allocations computed by Alg. 1 with the real allocation cost incurred by the cloud provider and find that Algorithm 1 reduces the combined cost of the three ISPs that contribute a majority of the bandwidth cost (ISP-1, ISP-2 and ISP-3 peer links) by 65% on average (Figure 4a).

Impact of participating links. When the input to the optimization is a *single* peer's links and traffic matrix, we observe lesser, yet significant, cost savings. This can be seen in the trends for ISP-1 and ISP-2 in Figure 4a. This shows that our cost optimization techniques can be deployed incrementally in the cloud WAN by engineering the traffic flow to a few ISPs at first. The fraction of savings increase as more outbound links are included in the optimization.

Impact of peering rates. We show the impact of relative peering rates of the three participating ISPs in the cost optimization. For the optimization instances demonstrated in



(a) Cost savings with different sets of participating links.

(b) Impact of peering rates on cost saving.

Figure 4: (a) Cost savings for 12 billing cycles using traffic matrices of ISP-1, ISP-2, ISP-3 and their combinations. (b) Cost saving with ISP-1, ISP-2 and ISP-3 for different peering rate ratios.

Figure 4a, the ratio of peering rates of the ISPs is 2:2:3. While the exact peering rates are confidential, their ratio shows that links belonging to two ISPs cost less than the third ISP. To ensure that the cost savings are not simply a function of this specific cost ratio, we compare the savings from the optimization when the peering rates are in 1:1:1 and 1:1:2 ratios and demands are the same as before. Figure 4b shows that savings are significant ($\approx$ 40%) even when all links have the same peering rate. Significant cost savings with different peering ratios demonstrate the generality of our results.

Impact of engineered traffic volume It may not be desirable to allow all traffic from edge links to be engineered for saving network costs. For instance, it may be important to egress some portion of the traffic on the same edge link where the client request entered the cloud for performance or geo-political reasons. We find the impact of the fraction of traffic that can be engineered on a per-link basis by computing the cost gains for the month of June 2018 when the fraction of engineered outbound traffic on the edge links of ISP-1, ISP-2 and ISP-3 is 50%. We find that the resulting cost savings are 37.5%. We note that the solution took longer than our time limit for the solver and therefore the LP gap was higher than 15%. Similarly, when the fraction of traffic engineered on a link is reduced to 40%, the overall cost saving is 28.6%.

3.4 Computing optimal traffic allocations

We now discuss the details of our implementation of Alg. 1.

Managing the scale of the problem. Due to the non-convex nature of the problem, even state-of-the art optimization solvers can take an impractical amount of time to approximately solve Algorithm 1. We take advantage of our findings from §3.1 and only engineer peer links to the three North American ISPs (ISP-1, ISP-2 and ISP-3, anonymized for confidentiality) which incur a majority of the inter-domain bandwidth costs to the cloud. Each of the 3 ISPs peers with the cloud provider at tens of locations in North America, contributing 56 peer links between the cloud network and the three ISPs. We solve Algorithm 1 for different sets of peering links: first considering links with ISP-1 and the egress demand (d_j) that gets served over links with ISP-1. Similarly, we solve problem instances with links and demands of ISP-2, ISP-3, ISP-1 and ISP-2 and ISP-1, ISP-2 and ISP-3 as input.

Efficient computation of the lower-bound. Cutting-edge optimization solvers use a combination of techniques to solve general Mixed Integer Programs (MIPs). At a high level, the first step is *relaxing* the MIP to an efficiently solvable Linear Program (LP) by removing the integral constraints. If a feasible solution to the LP is not found, the MIP, in turn, is also infeasible. If a feasible solution is found, the solution of the LP is a lower bound to the solution of the original MIP. Therefore, in a minimization problem like Algorithm 1, the LP solution provides the lower bound on bandwidth cost without having to solve the MILP.

Running time of the optimization solver. We note that Algorithm 1 has $O(mn)$ Real decision variables and just as many binary variables. Predicting the difficulty of Integer programs in terms of the number of variables and constraints is hard. Indeed, *increasing* the number of links (size of set L) *reduces* the algorithm's running time. The rationale behind this counter-intuitive behavior is that higher number of peering links make it easier for the optimization to meet demands without raising the 95^{th} percentile utilization of the links.

Once the LP relaxation has been solved, MIP solvers use a branch-and-bound strategy to find feasible solutions to the MIP from an exponential number of possibilities. As a result, some instances of the optimization can take several hours to solve. We use two techniques to bound the time of the solver. First, using the efficiently computable LP relaxation, we compute the proximity of the MIP solution to the theoretical lower bound. Second, we configure the branch-and-bound algorithm to return the current-best feasible solution after a fixed amount of time has elapsed. We configure the solver to stop if the current best feasible solution to the MIP is within 15% of the LP optimal or if the solver has run for 15 hours.

Some instances of the optimization problem took 1-2 hours to find solutions while for others, the solution space had to be explored for 15 hours. On average, instances of Algorithm 1 took 6 hours to finish. The variance in run-time is due to differences in traffic demands of months. One strategy that was effective in speeding the optimization involved using the values of decision variables from the previous month as initial values of the corresponding decision variables for next month's model. We found that using this *warm-start* strategy reduced the running time by 3X with instances taking 2 hours to solve on average. We describe other approaches that did not reduce the running time in Appendix (§A.1).

Gap from LP optimal. While the optimal solution to the LP relaxation provides a lower bound on the minimum cost of allocations, this lower bound is not always feasible. To improve the run time, we set a break condition while solving the problem instances to either reach within 15% of the LP optimal or spend 15 hours in solving the MIP using branch-and-bound. For the instances we solved, the average gap of the final MIP solution from the LP optimal is 9% *i.e.,* the solutions are very close to the theoretical lower bound.

4 Online cost-optimization with CASCARA

Results of the offline allocation scheme (3.2) show that there is significant potential for optimizing bandwidth cost at the cloud edge. There are two caveats to the scheme's use: first, it assumes knowledge of outbound demand for every time slot of the billing cycle. In practice, an online algorithm that can allocate network flow to peer links without the knowledge of future demands is required. Second, the optimization formulation (Algorithm 1) takes two hours on average to provide optimal traffic allocations for the entire month. However, state-of-the-art TE controllers compute traffic allocations every 5-10 minutes, making it crucial to have an online solution that is efficient and effective. In this section we develop a heuristic-based online traffic allocation framework that uses insights from the offline optimal solutions to Algorithm 1. Despite the complexity of the cost optimization problem, we show that a simple and efficient algorithm with few hyperparameters governs the closeness of the heuristic solution to the offline optimal. The heuristic allocations achieve bandwidth costs savings within 5% of the optimal.

Consider the set of edge links from the cloud, L = $\{l_1, l_2, ..l_m\}$. Let L_i be a subset of L, such that links in L_i are each priced at p_i per Mbps. For example, the setup in (3.2) has two such subsets, L_1 and L_2 where links in L_1 are priced at p_1 and those in L_2 are priced at p_2. Since the peering rates of links to ISP-1, ISP-2 and ISP-3 are in the ratio 3:2:2, $p_1 = \frac{3}{2}p_2$. From the results of Section 3.4, we derive three key insights about the optimal traffic allocations:

Lower utilization of expensive links. When $p_2 < p_1$, the optimal traffic allocations use links in L_1 minimally. This means that barring capacity considerations, it is always cheaper to use links in L_2 to meet the demand and only use links in L_1 for their *free* 5% time slots.

Maximize the utilization of free slots. Figure 5 shows the density distribution of optimal allocations on an edge link by Algorithm 1. We note that the optimal allocations reduce the link's 95th percentile utilization to ≈ 15% of its capacity.

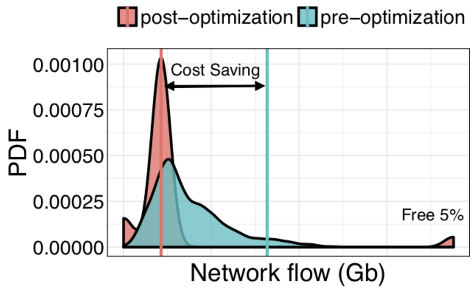


Figure 5: Optimized allocations on an edge link for a month. The vertical lines show the pre- and post-optimization 95th percentile utilization on the link. (X-axis labels removed.)

However, during 5% of time slots the link is utilized nearly at full capacity without contributing to the billable-bandwidth. Optimal allocation on all links show similar patterns.

Link utilization below the 95th percentile is uneconomical. Let u_j be the 95th percentile utilization of an egress link l_j. Assigning less than u_j flow to link l_j in any time slot is wasteful, *i.e.,* the link will get billed for u_j even if its utilization in other time slots is lower (Appendix Figure 13a).

4.1 Online traffic allocation

Using insights derived from the optimal allocations, we propose an online traffic allocation scheme, CASCARA, for the cloud edge. CASCARA *pre-decides* the fraction of the total network capacity that will be the billable bandwidth for the month (C_f). Given the billable bandwidth, finding the optimal pre-decided 95th percentile utilization of link l_j (u_j) is a special case of the bin-packing problem. Thus, greedy assignment of C_f to links in the increasing order of their peering rates minimizes the total bandwidth cost of the network. Since subsets of links ($L_i \subset L$) have the same peering rate, we assign u_j to links in the same subset using the progressive filling algorithm to ensure max-min fairness [2] within link subsets.

When a billing period begins, every link has a 95th percentile utilization (u_i) assigned to it. As new outbound demands arrive, if they can be met with $\sum u_i = C_f$ capacity, CASCARA allocates corresponding flows to the links. However, if the outbound demand exceeds C_f, CASCARA chooses to utilize one or more links at near full capacity to meet the demand. Since 5% of billing slots do not contribute to the links' costs, CASCARA ensures it only runs a link at near capacity for 5% or fewer billing time slots.

Parameters to the online algorithm. It is crucial to select C_f such that all demands in the billing period are met within C_f or by augmenting C_f with the extra capacity of links in their 5% free time slots. Once a link's 95th percentile utilization has been chosen to be u_i, using it for any lesser makes no difference to its final cost. The choice of C_f is critical to making a feasible allocation. If C_f is too low, the allocation may be infeasible or if it is too high, the bandwidth cost can be sub-optimally high. We discuss the choice of initial C_f and how CASCARA improvises when the chosen C_f is too small to meet the demand during the billing cycle.

Order of choosing peer links. CASCARA decides the order of links to be augmented above their allocation u_i to meet 5-minute demands higher than C_f. Using a configurable parameter, CASCARA can allocate how close the augmented allocation is to the link's capacity to prevent sudden link performance degradation. The time slots in which CASCARA augments the allocation to a link are called *augmented* slots. The augmented slots are limited to 5% for each link, making the order in which links are augmented relevant to the feasibility of an allocation. CASCARA uses a priority queue of all edge links where a link's priority is a combination of the time since it was last augmented and its capacity. If a link was augmented in the previous slot, it must also be augmented in the following slot, if required, so that the allocations do not change sharply. By prioritizing links with lesser capacity for augmentation, CASCARA ensures that free slots of links with higher capacity are not used pre-maturely.

Link augmentation order does not impact feasibility. If CASCARA's assignment of u_is and the order of link augmentation leads to an infeasible allocation problem, any change to the order of link augmentation does not render the allocation feasible (Proof in Appendix A.2). Since u_is are derived from C_f, the key input parameter to CASCARA is C_f. Algorithm 2 shows the online traffic allocation scheme of CASCARA in brief (details in Appendix Algorithm 3).

Insufficient C_f and infeasible allocation. If the initial capacity fraction assigned by CASCARA ends up being insufficient to meet the demand in a time slot, despite augmenting the allocations to all edge links that have free slots remaining, we consider the allocation infeasible. This means that it is no longer possible to limit the billable bandwidth of this month to C_f and the C_f value must be increased. CASCARA increases the value of C_f by step size (β) to meet the demand. Until it becomes necessary to increase C_f in a billing cycle, CASCARA has under-utilized the links stay under C_f. Increasing C_f to $C_f + \beta$ renders the past efforts to keep C_f low, futile. Indeed these efforts may have wasted the augmentation slots of links before C_f is incremented. However, there is no choice but to increase C_f as traffic demands must always be met. In the ideal case, initial value of C_f is just enough to meet demands in the entire billing period using augmentation slots when needed. On the other hand, starting the billing cycle with a C_f that is higher than required leads to sub-optimally high bandwidth costs. We show that the *ideal* C_f value is sufficient in ensuring that CASCARA finds optimal cost allocations.

Improvising billable bandwidth preemptively. When CASCARA finds that the demand is too high to accommodate in the current C_f, it increases C_f by β. Increasing the billable bandwidth estimate, C_f is a tradeoff – increasing too late in the billing cycle leads to wasteful use of links' free slots until the increase and increasing it too early reduces the cost saving potential. We capture this tradeoff by introducing the third and final parameter of CASCARA: α. α is the increase in C_f during the monthly billing cycle before an infeasible allocation is encountered. The goal is to preemptively increase C_f if such an increase is inevitable later in the month.

Algorithm 2: Online Traffic Allocation Per-Timestep

```
Function allocate_timestep($d, f$):
    if $d \le C_f$ then
        allocate $C_f$ to links in $L$
        return true
    else
        $d = d - C_f$
        while linkqueue do
            $l$ = pop(linkqueue)
            augment $l$
            decrement $l$'s priority and free slots
            decrement $d$ by $l$'s augmented capacity
            if $d \le 0$ then
                return true
        return false
```

4.2 Finding CASCARA's hyperparameters

We show that by setting C_f effectively, CASCARA's online traffic allocation (Algorithm 2) can be nearly as effective as the offline solutions of Algorithm 1. We set C_f to different fractions of the total network capacity, ranging from 0 to 1, in steps of 0.01. We compare the cost saving from the feasible allocation using the smallest C_f with the optimal cost saving[3] and find that on average, CASCARA with the optimal initial C_f achieves savings within 2% of the offline optimal allocation.

Setting C_f. CASCARA with the optimal C_f is called CASCARA-offline since it has prior knowledge of the lowest C_f for feasible allocations. CASCARA-online assumes no such knowledge and uses the optimal C_f of the previous billing cycle as the current month's initial C_f. This choice is motivated by strong daily, weekly and monthly seasonality in the outbound traffic demands. Previous month's C_f is the optimal value for the next month 64% of the time. For the rest, the average difference between optimal C_f and its initial setting is of the network capacity. When the initial C_f is not optimal, the allocation becomes infeasible and CASCARA has to increase the C_f to meet the traffic demands.

Finding α and β with grid search. Increase in C_f is a definite increase in the bandwidth cost for the billing cycle. The step size by with C_f is increased (β) is also important: too high and it would wastefully increase the cost, too low and it would mean having to increase C_f again in the future. Once increased, there is no cost saving advantage to reducing C_f. Incrementing C_f later is worse than having started with the

[3]For confidentiality reasons, we cannot not share the capacity fractions.

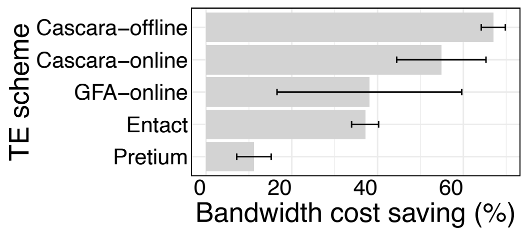


Figure 6: Costs savings from Cascara and related approaches. Bars show the mean and wiskers show the standard deviation.

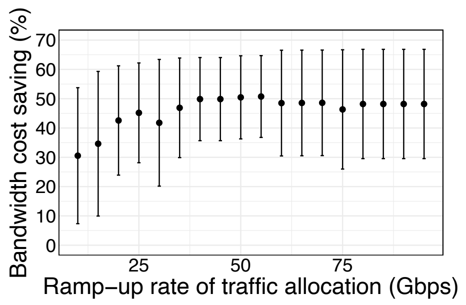


Figure 7: The impact of ramp-up rate on cost saving potential (mean and std. deviation calculated over 12 billing cycles).

optimal C_f since links' augmentation slots are wasted before the increment is made. Thus, preemptively increasing C_f by α during the billing cycle mitigates the issue of wasteful use of link augmentation slots. The hyperparameters, α and β are important to select. We perform a grid search to find the ones best suited for the cloud network. Details of the grid search are in Appendix A.5. The best values of α and β are used to for the following discussion.

4.3 Comparison with previous work

We now discuss the cost savings enabled by Cascara-online over twelve billing months from June 2018 to June 2019 (Figure 6). As before, we use the production network's traffic demands, topology and peering rates to measure the cost savings that Cascara-online would provide. We first show that Cascara-online achieves 55% cost saving, within 10% of the savings from Cascara-offline which knows the optimal C_f in advance. Then, we evaluate existing approaches that have focused on similar objective functions as Cascara. We exclude approaches that delay traffic to future time slots [13, 21] as these are not viable for the cloud provider we study (§7). The three main systems from related work are:

Pretium for dynamic file transfers in the WAN [20]. Pretium focuses on optimizing percentile costs of internal WAN links for dynamic transfers within the WAN [20]. They proposed to use the average of top 10% utilizations as a proxy for 95^{th} percentile cost of links. We find that Pretium offers modest cost saving of 11% on average compared to Cascara's 55% savings for egress WAN traffic. Pretium assumes that the 95^{th} percentile of a link's utilization is linearly correlated with the *average of top k utilizations* [20]. We evaluate this assumption using the utilizations of over 50 peering links from the cloud WAN to large ISPs in N. America. Figure 13b shows the Pearson correlation coefficient to measure the extent to which the average of top 10% utilizations can be used as a proxy for 95^{th} percentile utilization of inter-domain links. We find that the correlation coefficient for over 25% of the links is less than 0.5. Since previous work's hypothesis was derived from the data of a *single* WAN link measured a few years ago, the correlation between average of top 10% and 95^{th} percentile utilization may exist for some links but not all. Ever-changing traffic patterns from WANs due to new services like gaming also explain this difference.

Entact for cost minimization in clouds [35]. Entact shares a lot of the goals with Cascara, including finding cost optimal traffic allocations constrained by client latency. However, Entact chose to optimize *linear bandwidth prices* since percentile pricing is hard to optimize [35]. In a linear pricing scheme, greedy traffic allocation to cheapest links is optimal. However, the greedy algorithm does not fare well in percentile pricing schemes, as show in Figure 6's comparison between Cascara and Entact. The reason is that allocations in *every* time slot contribute towards the billable bandwidth in linear pricing schemes (*e.g.,* average and sum of allocations) but in percentile pricing, some percent of the allocations are *free*. Greedy allocations fail to take advantage of this phenomenon.

Global Fractional Allocation (GFA) for multihoming [12]. Finally, authors of [12] analyzed cost optimizations in the setting of multi-homed users. GFA comes closest in its approach to Cascara and this is also reflected in the cost saving comparison in Figure 6. However, Cascara outperforms GFA by 17% in the average case. There are two main reasons for this: GFA assumes a much smaller scale of the problem where the options for allocations are 3 to 4 upstream ISPs. This makes their naive estimation of cost lower bound ineffective: by using *only* 5% of the timeslots of peer links to meet demands was a viable option, traffic allocation would be *free*. Secondly, when GFA runs into an infeasible allocation, it assigns *all* remaining flow to a single link. This is often impractical at the cloud scale where the demand is too high for one peer link to handle the slack.

And finally, there are several realistic factors that need careful consideration: latency from peer links to clients and existence of routes at the peering router to engineer traffic. Cascara not only performs better in idealized environments by achieving higher cost saving that existing systems, it also takes real-world constraints of a large production WAN into account. We describe these in further details in §5.

4.4 Operational safety checks in Cascara

We discuss the safety checks built into the Cascara algorithm to ease the process of operating it in production.

Stable traffic allocation. One concern with algorithms as-

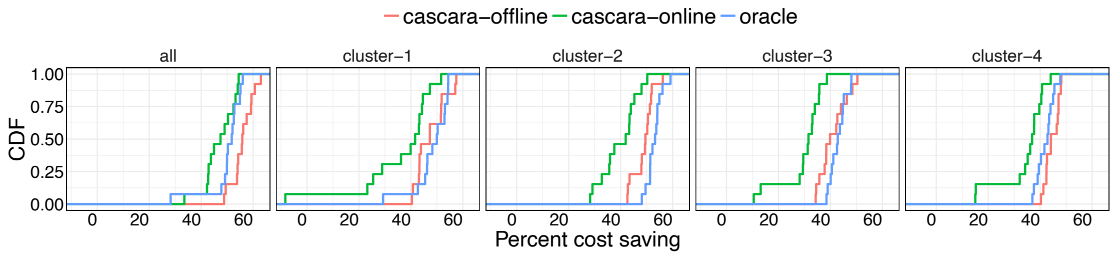


Figure 8: shows distributions of percent cost saving by CASCARA-offline, CASCARA-online and the oracle (Algorithm 1) over 12 billing cycles. CASCARA-online achieves near-optimal saving with different sets of links and corresponding demands as input.

signing traffic flows on peer links is that the allocation must be mindful of the performance impact on the inter-domain paths. CASCARA ensures that allocation to backup BGP paths does not change too rapidly by using a *maximum ramp-up* rate parameter that controls the maximum increase in the allocation to any peer link in the network. This ramp-up rate paces traffic allocation to links and allows CASCARA to incorporate path performance feedback into its decision making. We discuss how CASCARA incorporates performance metrics in its control loop in the next section. Figure 7 shows the cost saving potential of CASCARA as a function of the ramp-up rate. Very slow shifts which use a maximum ramp-up rate of 10 Gbps restrict the cost savings of CASCARA. However, at 30 Gbps ramp-up rate, CASCARA has reached its full saving potential and more rapid shifts of traffic do not offer much improvement in cost savings percentage.

Predictable traffic allocations on edge links. CASCARA's traffic allocation to edge links are more stable than present-day allocations which are driven by user-facing demands. There are two reasons for this. First, CASCARA selects a pre-decided fraction of a link's capacity as the utilization on the link for 95% of billing slots and changes are made to this fraction only when it is essential for meeting demand over C_f. Secondly, even when the allocation to a link has to be augmented, CASCARA ensures that a link, once augmented, is used until its free slots have been exhausted. Predictable allocations on edge links allow network peers to provision capacity appropriately in place of being prepared for arbitrary spikes in traffic demands.

Incremental deployability. CASCARA can be incrementally deployed across edge link groups in the cloud. To show this, we divide the peer links of ISP-1, ISP-2 and ISP-3 into four geographical clusters based on their PoP. These four clusters correspond to links at PoPs in north-central, south-central, East Coast and West Coast regions of North America. We compute the cost savings within each cluster by engineering the demands of the cluster onto its links. Figure 8 shows that CASCARA-online can achieve near-optimal cost (CASCARA-offline) savings across all peer links (cluster *all*) and also within the 4 geographical clusters. We note that in some cases CASCARA-offline achieves higher cost saving than the oracle (Alg. 1) due to the LP gap in the solution of the MILP (§3.4).

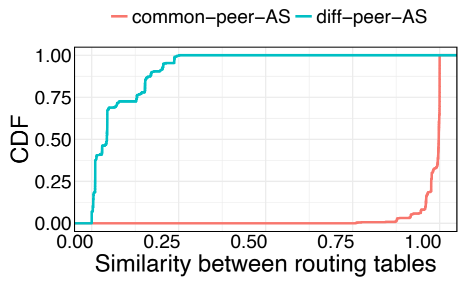


Figure 9: shows the distribution of address space similarity between peer links of ISP-1, ISP-2 and ISP-3 at different PoPs of the cloud. Each ISP announces nearly the same address space at different PoPs but the overlap in address space across ISPs is very small.

5 Performance-aware cost saving

We have demonstrated that there is significant potential of saving inter-domain bandwidth costs in a cloud network (§3) and CASCARA's efficient online algorithm can realize this potential by achieving near-optimal cost saving (§4). In this section we discuss practical aspects of achieving cost savings, namely, feasibility of engineering egress traffic in a WAN and the impact of CASCARA on client latency.

5.1 Availability of client routes at peer links

CASCARA engineers outbound traffic demand to peer links to achieve cost optimality over the billing cycle. However, it must ensure that peer links have the routes required for traffic shifted onto them. Otherwise, traffic to clients could get black-holed at the peering edge router. Using the routes announced by the three ISPs we focus on, we measure the address space overlap between peer links and find that ISPs announce the same address space across different peering locations (*e.g.,* Dallas vs. Seattle) but the overlap of address space across peers (*e.g.,* ISP-1 vs. ISP-2), even at the same PoP is minimal (Figure 9). Thus, CASCARA needs a mechanism to track the existence of relevant routes at peer links.

Tracking prefix route announcements by ISPs at different cloud PoPs in CASCARA leads to an explosion of the problem size since there are over 600,000 prefixes on the Internet. Aggregating clients to their corresponding geographic metropolitan area (metro) and autonomous system (AS) pair

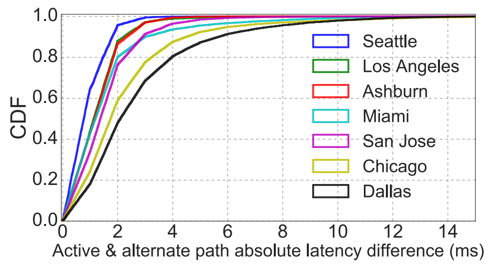


Figure 10: The difference in median latencies between primary and alternate BGP paths calculated from client measurements. The difference between latencies of primary and alternate paths is small.

significantly reduces the scale of the problem. This grouping of client prefixes within the same AS and small geographical locality has been used effectively in previous work [4]. We find that the points of presence where the cloud provider peers with ISP-1, ISP-2 and ISP-3 serve approximately 40,000 (metro, AS) pairs, reducing the scale of the mapping required to capture the existence of relevant BGP routes at peer links. Thus, we construct a bi-partite mapping between clients and peer links *i.e.,* an edge between client c to peer link p implies p has the relevant routes to c. We then constrain the traffic allocation in each timestep by the client to peer link mapping. We compute this allocation efficiently with a linear program (LP) within Alg. 2 that maps clients demands to peer links.

5.2 Bounded impact on client performance

Next, we tackle the challenge of limiting the performance impact of CASCARA's cost optimization. For this, we continuously measure the performance of alternate BGP egress paths to destination prefixes by directing a small amount of traffic over alternate peer links at eight PoPs [26,27,35]. We selected these PoPs as they carry high traffic volume – approximately 47% of all the cloud provider's North American traffic, and have high capacity alternate links.

Links at the same PoP have equivalent client latency. We analyze over 300 million measurements to the cloud PoPs for the month of August 2020, spanning 40,000 client metro and AS pairs, each with thousands of latency measurements towards the cloud on any given day. We first show the existence of latency equivalent peer links at the same PoP. Borrowing from existing methodology [26], we measure the difference in median latency between the BGP best path (*primary*) and the alternate BGP path for all clients that are served by the PoPs over 15 minute time buckets. Figure 10 shows that 80% of the time the difference in the latency is less than 3 ms. This implies that shifting client traffic to links at the same PoP, impacts the client latency by 3 ms or less.

Shifting traffic to peer links at a PoP different than the one where it ingressed introduces two challenges. First, it can inflate latency as the traffic would traverse the cloud backbone to reach the second PoP. The second PoP could be further from the client than the original, also inflating PoP to client latency. Second, traversal of the cloud backbone can congest backbone links but cloud providers often overprovision backbone capacity [6] and manage intra-WAN link utilizations with centralized controllers like SWAN [17] and B4 [19] to mitigate hot spots. Thus, we focus on the latency impact of CASCARA in this work. We find the *primary* PoP and peer ISP which historically has been the preferred egress for a client. This primary link defines the baseline for our experiments – any changes in client latency are measured in comparison with the primary peer and PoP.

Bound the latency impact in egress link selection. To limit the degradation to client latency, we inform CASCARA's allocation (Algorithm 2) of the most recent latency from a peer link to the client. In every timestep, while fulfilling demands to a client, CASCARA enforces that traffic is allocated along the primary and other sets of links. We select the set of links to empirically construct the relationship between latency impact and saving of CASCARA. We consider the set of links for each client to include ISPs with route towards the client – including a transit ISP, at the client's primary PoP. This means that along with the links to its primary ISP, the client's demand could be carried over the transit ISP link at the same PoP. This can increase the set of outbound link options for a client by two links in the best case. Since, links at the same PoP have equivalent latency, this configuration of CASCARA does not cause significant latency degradation (Figure 10).

We use CASCARA to engineer traffic at each PoP and compute the offline cost optimal solutions (Figure 11) for comparison. At some PoPs (PoPs 0, 2 and 13), there are up to five latency-equivalent peer links to most clients. *e.g.,* two interconnections with ISP-1, one with ISP-2 and two with the transit ISP. CASCARA-offline shows the potential to save up to 50% of bandwidth costs at such PoPs. At other PoPs (PoPs 4, 5, 11), there are only 2 latency-equivalent peer links to most clients *e.g.,* one interconnection with ISP-1 and one with the transit ISP. Moreover, high diversity in demands across PoPs due to client population density leads to differing opportunities of cost savings across PoPs. We use CASCARA-online to engineer traffic in an online manner with route and latency constraints. In each five minute timeslot, CASCARA allocated traffic to clients on latency equivalent links at the client's primary PoP. On average, each iteration of CASCARA takes approximately 3 seconds to compute traffic allocations, including the construction of the LP and extraction of traffic allocations on links. We note that our implementation uses Python 2.7 and could be further optimized for running time. However, TE systems typically perform allocations once every 5-10 minutes, thus CASCARA's runtime of 3 seconds is reasonable. Across all PoPs, CASCARA achieves the overall cost saving of 21% while ensuring that client latency remains unaffected. The per-PoP configuration we have evaluated enforces the strictest possible latency bound on CASCARA. CASCARA allows cloud providers to configure the acceptable

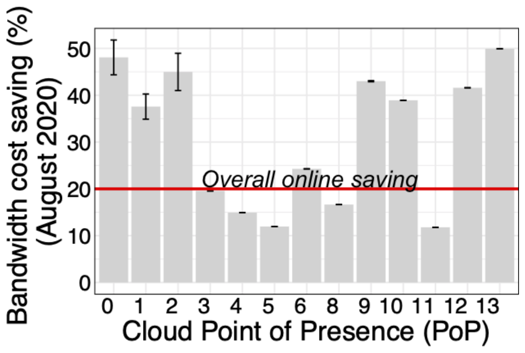


Figure 11: shows cost savings with CASCARA when engineering traffic on a per-PoP basis while limiting the impact on client latency to 3 ms in the worst case (mean and standard deviation computed over three runs of CASCARA).

worst-case latency degradation while saving bandwidth costs.

6 Discussion

In this section, we investigate the source of CASCARA's cost savings. We discuss implications of our findings on peering contracts with ISPs and bandwidth pricing on the Internet.

6.1 Where do the cost savings come from?

Network operators have historically used heuristics to limit their bandwidth costs. These include, load balancing traffic over equivalent links and preferring cheaper peer links in the BGP best path selection by setting localpref appropriately.

Localpref based cost saving is sub-optimal. We illustrate the cost savings from CASCARA with a small example using 2 links and 3 billing slots. There are two egress links from a network (Link 1 and Link 2), each of capacity 5 traffic units and unit peering rate. The traffic demand is assigned to Link 1 and 2 in any time slot (Figure 12). Traffic must not be dropped if there is enough capacity on the outbound links. For simplicity, the links are billed using the median (50^{th} percentile) utilization over three time slots. Since the peering rate of both links is the same, localpref-based cost minimization will simply balance traffic on the two links. Under this scheme, the link utilizations are : $1, 2.5, 1.5$ in time slots 1, 2 and 3 respectively (shown in red in Figure 12) for both links. The median utilization is 1.5 for both, the total cost of the links is 3 units. An alternate traffic assignment to the links is shown in blue in Figure 12, where the utilizations of link 1 and 2 are $\{1, 5, 0\}$ and $\{1, 0, 3\}$ respectively. The median cost in this case is 1 for both links, total cost being 2 units. This scheme saves one third of the traffic cost while meeting the same demand. We note that by extension, sending all traffic to a link that is cheaper would also be sub-optimal.

Free time slots for saving cost. The example shows that in case of median billing, one of the three time slots does not contribute towards the final cost of the link. Each link has one free slot that can absorb peaks in demands to reduce

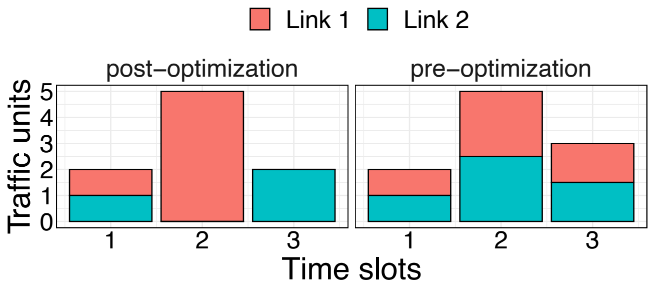


Figure 12: A toy example comparing CASCARA's cost-optimized allocations vs. load-balanced allocations.

costs. Similarly, bandwidth on the Internet is billed using 95^{th} percentile billing, meaning that 5% of 5-minute time slots in a month are *free* for each link. This implies that for roughly 36 hours in a month, traffic allocation on any link does not contribute to the final billed cost. While it may seem that the free slots provide little wriggle room for saving cost, cloud providers have a rich diversity of network peers in several PoPs. These peer links provide free slots in the billing context and enable multiple latency-equivalent ways to reach clients.

6.2 Do the findings generalize?

We believe our results generalize to any large global cloud, content provider, or content delivery network. The first reason is that the cloud provider network is not unique. These networks all share several critical properties in common with each other: **(1)** presence in hundreds of PoPs around the world to deliver traffic very close to users and **(2)** extensive peering and short AS paths [5, 32]. The second reason is that other large networks have shown that given such large deployments and peering, many of the alternate paths to users have similar latency [1, 26]; also allowing these networks to optimize bandwidth costs with stable performance.

Cost optimization is not a one-time effort. Traffic patterns across billing slots change – demands have been rising steadily at 30-40% per year. The surge in demand [7] from the COVID-19 pandemic has made traffic patterns more dynamic. We have evaluated CASCARA using over a year worth of demands, including evaluation in August 2020 to capture the post-pandemic traffic growth. Our findings show small month-to-month variation in saving but overall, the savings are significant and consistent. We note that cost savings compound over time as demand continues to rise exponentially.

6.3 How practical is CASCARA?

While CASCARA benefits from large sets of latency-equivalent peer links, it can be deployed incrementally over peer links (§5), allowing cloud operators to choose to expand CASCARA's purview over time. CASCARA can bound the amount by which allocations to links can change across time slots to prevent sudden changes in traffic (§4.4). Moreover, we foresee CASCARA as a component of a larger software-defined edge [27, 33] that already prioritizes successful traffic delivery based on the capacity and availability of the downstream path. During demand surges or outages, the high priority components of the TE system may take action to mitigate customer impact, putting CASCARA on hold for some types of traffic for short periods of time. Automated network build-out alerts limit the duration of persistent capacity crunches, enabling cost savings from CASCARA in the long term. Where cost-optimization falls among second-order priorities will vary across cloud providers and their business needs.

6.4 Implications for existing peering contracts

An important concern in optimizing the cost of inter-domain traffic is the long-term impact it may have on peering contracts. For instance, if free peers observe higher traffic volume from the cloud, they may reconsider their peering agreement or lean towards paid exchange of traffic [22]. Due to these factors, we evaluated CASCARA only on links with paid North American peers. We argue that the peering rate captures the value of the interconnection to both networks involved and thus optimizing the outbound allocations for cost, not exceeding the peering port capacity at the edge, is a reasonable strategy. Additionally, peering rates in certain regions of the world are disproportionately high due to monopolistic transit ISPs and complicated socio-political factors, making high bandwidth rates the cost of operating in the market.

6.5 Implications for bandwidth pricing

CASCARA shows that the abundance of latency-equivalent peer links has enabled networks to significantly reduce their expenditure on inter-domain bandwidth. With the findings of CASCARA, we encourage the community to revisit the classic problem of pricing inter-domain traffic effectively. A subject studied since the dawn of the Internet [24], inter-domain bandwidth pricing models and rates determine paths taken by traffic and subsequently the end-user performance. With the emergence of cloud and content providers as the source of disproportionately large volume of Internet traffic, current pricing models may not suffice in ensuring the harmonious existence of networks on the Internet [10, 30]. Today, a handful of networks (cloud and content providers) can take advantage of their rich connectivity to save inter-domain bandwidth costs, potentially taking a portion from the profits of ISPs. Some recent proposals suggest ways to better align the cost of Internet transit and the revenue gained by networks [16, 34].

7 Related Work

In this section, we discuss important pieces of work related to CASCARA and set them in the context of our contributions.

Intra-WAN traffic engineering. Large cloud providers have embraced software-defined, centralized traffic engineering controllers to assign flow within their private WAN to maximize their utilization, guarantee fairness and prevent congestion [17,19,23]. Bandwidth costs in the context of WANs were considered in Pretium [20] (comparison with CASCARA in Section 3.4). Stanojevic *et al.* used Shapley values to quantify the value of individual flows under percentile pricing [28].

Engineering the WAN egress. Recent work has proposed a software-defined edge to manage outbound flows from their networks [27, 33]. The goal of these efforts has been to react to poor client performance by switching to better performing BGP next hops. The allocation decisions made by CASCARA can be implemented using a software defined edge like Espresso or EdgeFabric. The subject of TE in multi-hoped networks has been studied [12, 25] and we compare CASCARA with a representative set of work from this space (§3.4).

Engineering delay tolerant traffic. Previous work has explored the potential of delaying traffic across timeslots to save bandwidth costs at the end of the billing cycle [21]. However, the cloud provider we analyze does not consider delaying client traffic by several minutes as a viable option.

Performance-based routing on the Internet. Google's Espresso [33] implements performance-based routing on the Internet to improve client performance. Recently, other large global networks have shown limited potential in optimizing latency by routing [1, 26]. Our work effectively exploits this realization by optimizing cost while keeping latency stable.

Bandwidth pricing schemes. In the early years of the Internet, economists studied potential mechanisms to price bandwidth [24]. Congestion pricing was proposed to bill based on the use of network resources at times when they are scarce. These pricing schemes incentivize users to reduce consumption of network resources during peak utilization by pricing bandwidth higher when the network is congested.

8 Conclusion

In this work, we quantify the potential of saving inter-domain bandwidth costs in a large commercial cloud provider and find that optimal allocations can save up to 60% of current inter-domain bandwidth costs while meeting all traffic demands as they arrive. Inspired by this, we develop an efficient online TE framework, CASCARA, that achieves cost savings within 10% of the optimal. CASCARA's cost savings are robust to changes in traffic patterns and peering rates. Finally, we show that CASCARA can balance the cost-performance tradeoff by achieving 11-50% cost savings per cloud PoP without degrading client latency significantly.

9 Acknowledgements

We thank our shepherd, Srikanth Sundaresan and the anonymous reviewers for helping improve this work. We are grateful to Anuj Kalia, Akshay Narayan, Arvind Krishnamurthy, Dan Crankshaw, Emaad Manzoor and Phillipa Gill for their feedback. We thank Ingrid Erkman, Brett Lewan and the Azure WAN team for their feedback on CASCARA's design.

References

[1] Todd Arnold, Matt Calder, Italo Cunha, Arpit Gupta, Harsha V. Madhyastha, Michael Schapira, and Ethan Katz-Bassett. Beating bgp is harder than we thought. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, HotNets 2019, page 9?16, New York, NY, USA, 2019. Association for Computing Machinery.

[2] Dimitri P Bertsekas and Robert G Gallager. *Data networks*, volume 2.

[3] CAIDA. CAIDA AS Classification. http://data.caida.org/datasets/as-classification/, (Accessed on 2020-01-15).

[4] Matt Calder, Ryan Gao, Manuel Schröder, Ryan Stewart, Jitendra Padhye, Ratul Mahajan, Ganesh Ananthanarayanan, and Ethan Katz-Bassett. Odin: Microsoft's scalable fault-tolerant CDN measurement system. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 501–517, Renton, WA, April 2018. USENIX Association.

[5] Yi-Ching Chiu, Brandon Schlinker, Abhishek Balaji Radhakrishnan, Ethan Katz-Bassett, and Ramesh Govindan. Are we one hop away from a better internet? In *Proceedings of the 2015 Internet Measurement Conference*, pages 523–529, 2015.

[6] David Chou, Tianyin Xu, Kaushik Veeraraghavan, Andrew Newell, Sonia Margulis, Lin Xiao, Pol Mauri Ruiz, Justin Meza, Kiryong Ha, Shruti Padmanabha, et al. Taiji: managing global user traffic for large-scale internet services at the edge. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 430–446, 2019.

[7] Cloudflare. Cloudflare During the Coronavirus Emergency. https://blog.cloudflare.com/cloudflare-during-the-coronavirus-emergency, (Accessed on 2020-03-12).

[8] CVX Research. Software for Disciplined Convex Programming. http://cvxr.com/, (Accessed on 2019-10-02).

[9] Dr. Peering. The Art of Peering: The Peering Playbook. http://drpeering.net/white-papers/Art-Of-Peering-The-Peering-Playbook.html, (Accessed on 2020-08-01).

[10] Free Press. Net Neutrality. https://www.freepress.net/issues/free-open-internet/net-neutrality, (Accessed on 2020-01-19).

[11] GNU. GNU Linear Programming Kit. https://www.gnu.org/software/glpk/, (Accessed on 2019-10-02).

[12] David K Goldenberg, Lili Qiuy, Haiyong Xie, Yang Richard Yang, and Yin Zhang. Optimizing cost and performance for multihoming. *ACM SIGCOMM Computer Communication Review*, 34(4):79–92, 2004.

[13] L. Golubchik, S. Khuller, K. Mukherjee, and Y. Yao. To send or not to send: Reducing the cost of data transmission. In *2013 Proceedings IEEE INFOCOM*, pages 2472–2478, 2013.

[14] Igor Griva, S Nash, and Ariela Sofer. *Linear and Nonlinear Optimization: Second Edition*. 01 2009.

[15] Gurobi. GUROBI Optimization. https://www.gurobi.com/, (Accessed on 2019-10-02).

[16] Yotam Harchol, Dirk Bergemann, Nick Feamster, Eric Friedman, Arvind Krishnamurthy, Aurojit Panda, Sylvia Ratnasamy, Michael Schapira, and Scott Shenker. A public option for the core. SIGCOMM '20, page 377–389, New York, NY, USA, 2020. Association for Computing Machinery.

[17] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with Software-driven WAN. *SIGCOMM*, 2013.

[18] IBM. CPLEX Optimizer. https://www.ibm.com/analytics/cplex-optimizer, (Accessed on 2019-10-02).

[19] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a Globally-deployed Software Defined Wan. *SIGCOMM*, 2013.

[20] Virajith Jalaparti, Ivan Bliznets, Srikanth Kandula, Brendan Lucier, and Ishai Menache. Dynamic Pricing and Traffic Engineering for Timely Inter-Datacenter Transfers. In *SIGCOMM'16*, 2016.

[21] Nikolaos Laoutaris, Georgios Smaragdakis, Pablo Rodriguez, and Ravi Sundaram. Delay tolerant bulk data transfers on the internet. In *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, pages 229–238, 2009.

[22] Steven Levy. *In the plex: How Google thinks, works, and shapes our lives*. Simon and Schuster, 2011.

[23] Wenxin Li, Xiaobo Zhou, Keqiu Li, Heng Qi, and Deke Guo. Trafficshaper: shaping inter-datacenter traffic to reduce the transmission cost. *IEEE/ACM Transactions on Networking*, 26(3):1193–1206, 2018.

[24] Jeffrey K MacKie-Mason and Hal R Varian. Pricing congestible network resources. *IEEE journal on Selected Areas in Communications*, 13(7):1141–1149, 1995.

[25] B. Quoitin, C. Pelsser, L. Swinnen, O. Bonaventure, and S. Uhlig. Interdomain traffic engineering with bgp. *Comm. Mag.*, 41(5):122?128, May 2003.

[26] Brandon Schlinker, Italo Cunha, Yi-Ching Chiu, Srikanth Sundaresan, and Ethan Katz-Bassett. Internet performance from facebook?s edge. In *Proceedings of the Internet Measurement Conference*, IMC ?19, page 179?194, New York, NY, USA, 2019. Association for Computing Machinery.

[27] Brandon Schlinker, Hyojeong Kim, Timothy Cui, Ethan Katz-Bassett, Harsha V Madhyastha, Italo Cunha, James Quinn, Saif Hasan, Petr Lapukhov, and Hongyi Zeng. Engineering egress with Edge Fabric: Steering oceans of content to the world. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 418–431. ACM, 2017.

[28] Rade Stanojevic, Nikolaos Laoutaris, and Pablo Rodriguez. On economic heavy hitters: shapley value analysis of 95th-percentile pricing. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 75–80, 2010.

[29] TeleGeography. The State of the Network. `https://www2.telegeography.com/hubfs/assets/Ebooks/state-of-the-network-2019.pdf`, (Accessed on 2020-01-19).

[30] TIME. Netflix's Disputes With Verizon, Comcast Under Investigation. `https://time.com/2871498/fcc-investigates-netflix-verizon-comcast/`, (Accessed on 2020-01-19).

[31] Brian Trammell and Benoit Claise. Specification of the IP flow information export (IPFIX) protocol for the exchange of flow information. *RFC7011*, 2013.

[32] Florian Wohlfart, Nikolaos Chatzis, Caglar Dabanoglu, Georg Carle, and Walter Willinger. Leveraging interconnections for performance: the serving infrastructure of a large cdn. In *SIGCOMM*, pages 206–220, 2018.

[33] KK Yap, Murtaza Motiwala, Jeremy Rahe, Steve Padgett, Matthew Holliman, Gary Baldus, Marcus Hines, TaeEun Kim, Ashok Narayanan, Ankur Jain, Victor Lin, Colin Rice, Brian Rogan, Arjun Singh, Bert Tanaka, Manish Verma, Puneet Sood, Mukarram Tariq, Matt Tierney, Dzevad Trumic, Vytautas Valancius, Calvin Ying, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Taking the Edge off with Espresso: Scale, Reliability and Programmability for Global Internet Peering. In *SIGCOMM'17*, 2017.

[34] Doron Zarchy, Amogh Dhamdhere, Constantine Dovrolis, and Michael Schapira. Nash-peering: A new techno-economic framework for internet interconnections. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 403–408. IEEE, 2018.

[35] Zheng Zhang, Ming Zhang, Albert Greenberg, Y. Charlie Hu, Ratul Mahajan, and Blaine Christian. Optimizing cost and performance in online service provider networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, page 3, USA, 2010. USENIX Association.

A Appendix

A.1 Speeding the MIP solution

In this section we briefly describe the intuitive but ineffective methods we employed to speed the execution of the MILP. The methods did not yield a reduction in running time but we document these for completeness.

Solving the MILP in smaller time slices. Since link utilizations at the edge exhibit strong daily and weekly seasonality, we hypothesized that solving the cost optimization in smaller chunks of time, say, one week at a time, and then stitching together the resulting solutions would find the entire month's optimal allocations. While the smaller problems of weekly allocations could be solved in approximately ten minutes, when stitched together, the overall solution is very far from optimal. In fact, the allocations obtained via this process did not show any significant reduction in inter-domain bandwidth cost over the present-day traffic allocations. On investigating the reason why this approach does not work, we found that while there are regular trends in the traffic demand, bursts of traffic are not spread uniformly across all weeks of a month. Accommodating these bursts with a local, week-long view leads to overall poor cost saving from the stitched allocations.

Automated parameter tuning. The commercial solver we used (Gurobi), provides a tool for automatically tuning the parameters of the solver for a given optimization model. We attempted to use this tool to find parameters which gave the best performance in terms of running time and closeness to the optimal. However, our model was too large for the autotune to deliver any results. Thus, we selected the appropriate parameter values manually by several runs of the optimization. These parameters are documented in the code repository we have released.

A.2 CASCARA's link augmentation order

We show that changing the order of links that CASCARA augments during the billing cycle does not make an unfeasible allocation, feasible. We take the example of an unfeasible ordering, $O_{unfeasible}$ where the demand in timeslot k cannot be met even after augmenting the capacity of all links. Consider the following change in the position of link l_i in the ordering: if l_i is picked for augmenting in timestamp k in place of timestamp j where $j \leq k$. If this were possible, then CAPACITY(l_i) would be available for use in timestamp k. However, this is not possible since l_i had to be the *smallest* capacity link that met the excess demand of timeslot j, any other link that takes its place has to have a higher capacity. This means that by using another link in place of l_i, we would reduce the available capacity in timeslot k. Thus, a change in ordering of links for augmentation would not make a problem instance feasible. □

A.3 Traffic allocation with CASCARA

In this section we discuss details of the CASCARA allocation algorithm which were omitted in Section 4 for brevity. The complete algorithm, Algorithm 3, expands on Algorithm 2. L is the set of links in the network in the increasing order of thei peering rate. The algorithm shows how CASCARA allocates flow to links in every timestamp of the billing period. The solution to this algorithm are link allocations in all time steps. CASCARA maintains a priority queue of links and the priority of a link is decided based on two factors:

- Initial priority: all links have their initial priority set to the number of free time slots they have in the current billing cycle. We update the priority after augmenting the link. In any subsequent billing timeslots, if the demand is higher than C_f, links with lower priority *i.e.,* ones which were used in the previous slots are re-used again. This ensures that the link augmentation is not spread across many links.
- Link capacity: We prefer to augment lower capacity links to save the higher capacity links for the remaining billing cycle. If the demand is too high, high capacity links are more likely to absorb it with augmentation.

We also kep track of the remaining free slots for each link. When all links have exhausted their free slots, allocation in that timestep fails and we have to increment C_f. Let O be the order in which links got augmented. An example ordering of augmented links, O is like so:

$$O = \{[l_1, l_2, l_3], [l_1, l_2,], ..[l_k, l_{k+1}, l_{k+2,..}]\}$$

In timestamp 1, CASCARA augmented allocations to links l_1, l_2 and l_3. The starting priority of links is the same, so the priority queue returns links in ascending order of their capacity. In the next timeslot, CASCARA attempts to meet the demand by augmenting the same set of links to keep allocations stable.

CASCARA initializes C_f to the minimum value that produced a feasible traffic allocation for the previous month. If C_f is too low for the current month's demands, despite augmenting allocation to links, the traffic demand would not be met and C_f will be incremented by β. The augmented link ordering of an infeasible allocation would be like so:

$$O_{unfeasible} = \{[l_1, l_2..], ..., [l_k, l_{k+1}, .., l_m]\}$$

where $\sum_k^m \text{CAPACITY}(l_i) \leq demand - C_f$.

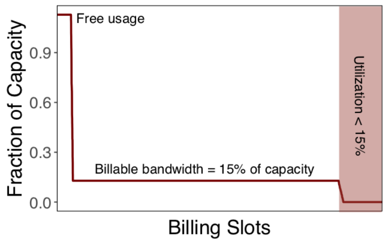
Additionally, CASCARA has a provision to proactively increment C_f by α (not shown in the algorithm). The goal is to proactively perform an inevitable increase in C_f to avoid wasting free slots of links. To do this, CASCARA checks if the number of links with free slots remaining is proportional to the amount of time left in the billing cycle. If the number of burstable links are too few,, C_f is incremented proactively.

Algorithm 3: Online Traffic Allocation (long version)

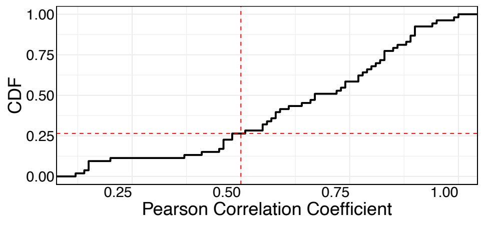
```
Result: Allocation of demand $d$ in every timestamp $t$
Input: $L, n, k, f,$ CAPACITY, $C, \alpha, \beta$
Initialization:
freeslots = $\frac{k}{100} * n$
$prio$ = freeslots          ▷ Initial priority of all links
$linkq$ = PRIORITYQUEUE()
for link ∈ $L$ do
    linkq.insert(link, CAPACITY(link), freeslots, $prio$)

Function allocate_timestep($d, f$):
    link_alloc = {}
    augmented_links = []
    $C_f = f * C$          ▷ Fraction $f$ of total capacity $C$
    if $d \leq C_f$ then
        link_alloc = bin_pack($L, C_f$)
    else
        $d = d - C_f$
        while $d \geq 0$ do
            b_link = linkq.pop()
            if !b_link then
                return {}
            augmented_links.add(b_link)
            if b_link ∈ $L_1$ then
                $d = d -$ (1 - f) $*$ CAPACITY(b_link)
            else
                $d = d -$ CAPACITY(b_link)
            link_alloc [b_link]= CAPACITY(b_link)

        for link ∈ augmented_links do
            link.prio = link.prio $-1$
            link.free_slots = link.free_slots $-1$
    return link_alloc
End Function
while not allocate_timestep($d, f$) do
    $f = f + \delta$
```

A.3.1 Link utilization below the billable bandwidth

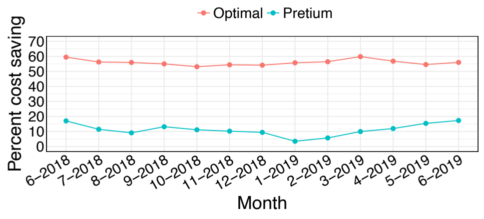
CASCARA chooses the *target* billable bandwidth (C_f) for a month. Given the billable bandwidth, it can be packed on to links by greedily assigning traffic ot cheaper links. [4] Given the minimum feasible C_f, this strategy is optimal. In fact, utilizing any link below its 95th percentile utilization is uneconomical – the link gets charged at the 95th percentile anyway. Figure 13a shows that while the utilization in some billing slots was below the 95th percentile (shaded red), yet, the link was billed for 15% of its capacity, making the period of utilization below 15%, wasteful.



(a) Link utilizations.



(b) Pearson's correlation.



(c) Pretium vs. Optimal saving

Figure 13: (a) Utilization of a link as fraction of its capacity, sorted from high to low across billing slots in a month. (b) 95th %-ile and avg. of top 10% correlation.(c) Cost saving by CASCARA vs. Pretium [20] on a month-by-month basis.

A.4 Details on the implementation of previous systems

We implement the optimization formulation from previous work using top 10% of utilizations in a month as the bandwidth cost of a link. We use CVXPY's implementation of sum of largest decision variables for this purpose. Since this formulation is a linear program, GUROBI solves it in less than a minute. While fast to compute, the allocations from this formulation are ineffective in saving the 95th percentile cost. Figure 13c compares the cost savings per-month between our solutions from Algorithm 1 and previous work. We note that the cost saving from the sum of top-k formulation are modest, 11% on average for all instances. We believe this is because of two assumptions made by previous work:

Assumption 1: 95th percentile of a link's utilization is linearly correlated with the *average of top k utilizations* [20]. We evaluate this assumption using the utilizations of over 50 peering links in a cloud WAN. These links connect the cloud WAN to large ISPs in N. America. We compute the Pearson correlation coefficient to measure the extent to which the average of top 10% utilizations can be used as a proxy for 95th percentile utilization of inter-domain links. We find that the correlation coefficient for over 25% of the links is less than 0.5. Since previous work's hypothesis was derived from the data of a *single* WAN link measured a few years ago, the correlation between average of top 10% and 95th percentile utilization may exist for some links but not all. Ever-changing traffic patterns from WANs due to the advent of new services like gaming also explain this difference.

Assumption 2: The correlation between average of top-k and 95th percentile of a link's utilization holds even after a new traffic allocation scheme replaces the current one. There is no guarantee that assumptions about allocation distributions hold in a newly proposed traffic engineering scheme. In fact, traffic engineering schemes *change* the allocation of flow along network links, modifying how links are utilized.

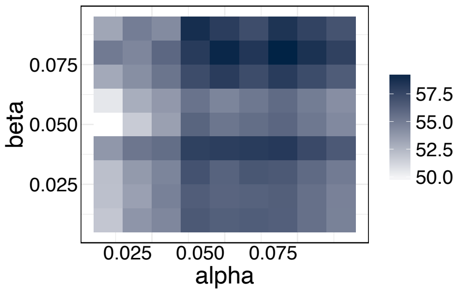


Figure 14: Average monthly bandwidth cost saving with CASCARA as a function of the parameters α and β. We choose the best values of α and β for the evaluation in §5.

A.5 Selecting CASCARA's hyperparameters

We sweep through potential values of α and β to find the ones that fit CASCARA the best. The range of values for α and β is $[0,1]$ since they represent increments to fraction of total network capacity. We sweep the space in steps of 0.01 to find the parameters that lead to the highest cost savings in the average case across all billing cycles (Figure 14).

[4]We have discussed the additional routing and client latency constraints on CASCARA in §5.

A Social Network Under Social Distancing: Risk-Driven Backbone Management During COVID-19 and Beyond

Yiting Xia[*§] *Ying Zhang*[§] *Zhizhen Zhong*[†§] *Guanqing Yan*[§] *Chiun Lin Lim*[§]
Satyajeet Singh Ahuja[§] *Soshant Bali*[§] *Alexander Nikolaidis*[§] *Kimia Ghobadi*[‡] *Manya Ghobadi*[†]
[*]*MPI-INF* [§] *Facebook* [‡] *Johns Hopkins University* [†] *MIT*

Abstract

As the COVID-19 pandemic reshapes our social landscape, its lessons have far-reaching implications on how online service providers manage their infrastructure to mitigate risks. This paper presents Facebook's risk-driven backbone management strategy to ensure high service performance throughout the COVID-19 pandemic. We describe Risk Simulation System (RSS), a production system that identifies possible failures and quantifies their potential severity with a set of metrics for network risk. With a year-long risk measurement from RSS we show that our backbone resiliently withstood the COVID-19 stress test, achieving high service availability and low route dilation while efficiently handling traffic surges. We also share our operational practices to mitigate risk throughout the pandemic.

Our findings give insights to further improve risk-driven network management. We argue for incorporating short-term failure statistics in modeling failures. Common failure prediction models based on long-term modeling achieve stable output at the cost of assigning low significance to unique short-term events of extreme importance such as COVID-19. Furthermore, we advocate augmenting network management techniques with non-networking signals. We support this by identifying and analyzing the correlation between network traffic and human mobility.

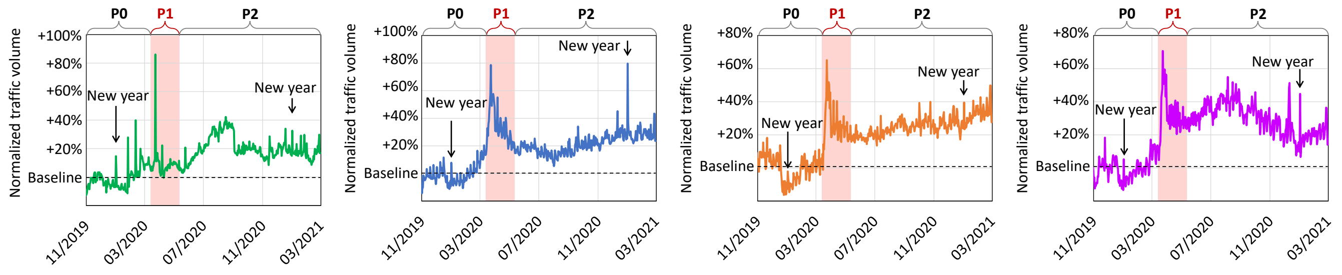
1 Introduction

COVID-19 fundamentally reshaped societal norms and human interactions by forcing most social activities to move online. The global network infrastructure was subjected to an unprecedented stress test as work, entertainment and education all had to be conducted via digital connections [37]. Over the past year, the networking community aimed to answer two fundamental questions about the impact of COVID-19 on different network environments [7,15,28]. First, how well has the current network infrastructure withstood the COVID-19 stress test? Second, how should the network infrastructure evolve to support a post-pandemic era likely to be permanently remodeled by the social distancing experience?

This paper supplements the recent COVID-19-centric research by sharing Facebook's experience emerging from the risk-driven backbone management strategy. Our work has two unique angles: the focus on the backbone network of a global online service provider and the use of network risk to quantify the robustness of the network infrastructure under adverse conditions. This study enriches previous observations made in different network environments, including the Internet [15], edge networks [7], and mobile networks [28]. Furthermore, it is a significant departure from prior work which uses only traffic measurement to quantify the impact of social events on the network infrastructure.

Our risk-driven backbone management is based on the fact that failures and disasters happen frequently and the backbone network should be equipped with sufficient protection capacity to mitigate the effects. Particularly, Facebook's backbone connects hundreds of Point-of-Presence (PoP) sites and tens of Data Center (DC) regions. At this scale, failures such as fiber cuts, router misconfigurations, and power outages happen on a daily basis [20], causing traffic congestion, packet loss, and latency increase which, in turn, negatively impact the network's availability and service-level agreements [8,21,27]. *Network risk* is an effective means to capture the impacts of potential failures in the network, before they actually occur, which is critical for identifying operational pain-points for long-term deployment planning, mid-term capacity augmentation, and short-term health monitoring.

This paper describes our Risk Simulation System (RSS), which performs comprehensive "what-if" analyses of network risk through traffic simulations under plausible failure scenarios. RSS has been in production for years. We introduce RSS in detail, showing key design decisions and engineering efforts to optimize the system over time. Specifically, we propose a set of risk metrics (demand loss, availability, latency stretch) to quantify impacts of potential failures from different aspects. Further, we introduce a high-fidelity failure model based on failure records from different data sources



(a) Asia (b) Europe (c) North America (d) South America

Figure 1: User-facing (DC-to-PoP) traffic traversing the global backbone.

and failure types. To scale our system, we provide different simulation granularities that trade failure count for simulation accuracy. Moreover, we discuss several techniques to accelerate computation such as system parallelization, routing simplification, and reduction of failure scenarios.

We conduct a year-long analysis of network risk using RSS, and in what follows, we share our operational experience of keeping risk at bay during the COVID-19 pandemic. Our risk analysis demonstrates our backbone network remained robust under the COVID-19 traffic surges. Although risk increased with traffic, even the most heavily affected class of service still achieved four 9s of availability and its flows only experienced 2.12% longer paths on average. We further discuss capacity enhancement and quality of service downgrade as two effective measures to reduce network risk.

Finally, we use case studies to show unusual network conditions caused by social distancing have challenged fundamental assumptions of the traditional network design, and we share our insights on future directions of risk-driven backbone management. We observe a large variation of optical and IP-layer failures triggered by changes of human activities. We thus suggest failure modeling to be more responsive to short-term failure statistics and discuss the tradeoff between model stability and agility for accurate failure predictions. We also identify the limitation of standard network management that only considers in-network signals and is blind to social impacts to the network. We find a negative correlation between traffic volume and population mobility rate during social distancing, and use it as an example to show opportunities for improving network management with external non-networking signals.

Our risk metrics, failure model and risk simulation approach generalize beyond the initially envisioned backbone network scenario and are readily applicable to other network environments. We believe that risk-driven network management has the potential to become the standard approach to disaster prevention, monitoring and recovery. Towards this goal, we hope that our experience can inspire future solutions and spur broader adoption of risk-driven network management. This work does not raise any ethical issues. We preserved user privacy and anonymity throughout this study.

Phase	Start date	End date	# Days
Pre-COVID (P_0)	11/4/2019	3/15/2020	133
Shelter-in-Place (P_1)	3/16/2020	5/3/2020	49
Re-opening (P_2)	5/4/2020	2/28/2021	301

Table 1: Measurement phases in this paper.

2 Traffic Surges During COVID-19

Recent news reports and measurement studies suggest significant traffic surges globally during the COVID-19 pandemic [7, 10, 15, 24, 28]. As a major social media platform, Facebook witnessed higher user engagement under social distancing. In this section, we measure Facebook's user-facing traffic to motivate the need for a risk-driven backbone management system.

Throughout the paper, we categorize our measurement period into three time phases (P_0 to P_2) listed in Table 1. The first phase, *Pre-COVID* (P_0), is our baseline to capture the state of the network before the global shut-down due to the pandemic. The second phase, *Shelter-in-Place* (P_1), marks the period when the US and European countries started to introduce extreme COVID-19 regulations, such as border closures, flight reductions, and school closures. The third phase, *Re-opening* (P_2), represents the slow re-opening phase when strict shut-down orders were relaxed [2].

Significant traffic increase. Figure 1 plots the traffic volume from our Data Center (DC) regions to Point-of-Presence (PoP) sites in four geographical regions. The traffic volume is normalized against average traffic during the pre-COVID phase (P_0). The figure shows a significant traffic surge starting mid-March 2020 in all regions, matching the timeline of global social distancing. In particular, we measure a traffic surge of 86% in Asia, 78% in Europe, 65% in North America, and 70% in South America in the P_1 phase.

Beyond the New Year traffic spike. Traffic volume spikes are not unusual. Today's service providers consider well-known flash-crowd events, such as Cyber Monday, in their traffic modeling [45] and network operation planning [49]. However, as Figure 1 depicts, the COVID-19 traffic increase has two unique differences. First, the traffic peak during phase P_1 was substantially higher than that of New Year's Eve (31

December 2019). The peak volume was 1.62× the 2020 New Year's Eve in Asia, 1.65× in Europe, 1.68× in North America, and 1.61× in South America. Second, flash-crowd events are usually short-lived, but the traffic surges remained high for several weeks during the pandemic.

The above observations highlight the challenges posed by social distancing on large-scale network operations. Under high traffic load, operators need to answer a natural question: "is my network at risk?" This question motivates us to quantify network risk and use it as a guiding signal to drive network management.

3 Risk-Driven Backbone Management

Satisfying Service Level Objectives (SLOs) is the ultimate goal of network management. Risk analysis is an indispensable and effective means to guarantee SLO compliance under different failure scenarios. In this section, we dive into the details of RSS, our risk-driven backbone management framework. We begin with the description of Facebook's traffic classification and routing schemes (§3.1), followed by our definition of risk metrics that align with SLO requirements of different service classes (§3.2). Next, we describe our failure modeling technique (§3.3). Finally, we present the design and implementation of RSS, our risk simulation system (§3.4).

3.1 Traffic Classification and Routing

Quality of Service (QoS). Facebook classifies the backbone traffic into four service classes. In this paper, we refer to them as QoS classes 1 to 4, where class 1 is the highest priority. Different classes of service use different queue assignments and routing policies. Flows with higher priorities have greater availability guarantees and can tolerate more failures compared to those in lower priority classes. This is often realized by over-provisioning extensive backup paths for redundancy. QoS class 1 contains essential network control traffic including network signaling and routing protocol messages to manage our network gear; class 2 is for critical services including most of our user-facing traffic; class 3 is our default class for most internal applications; and class 4 is for heavy, bulk data transfers. To reduce operational costs, we constantly look for opportunities to move traffic into class 4.

Routing. Our backbone uses a centralized network controller to make routing and Traffic Engineering (TE) decisions [22]. The centralized controller implements different traffic allocation algorithms for different QoS classes. To minimize the latency experienced by flows in QoS classes 1 and 2, we use a Constrained Shortest Path First (CSPF) approach that provisions TE tunnels for these flows up to the physical capacity of the network links. Flows are assigned to paths with the smallest round trip latencies. The bandwidth for QoS class 2 is allocated after class 1, and we reserve headroom on each link for potential traffic bursts for these two classes. QoS classes

Algorithm 1 Compute risk metrics for QoS class q

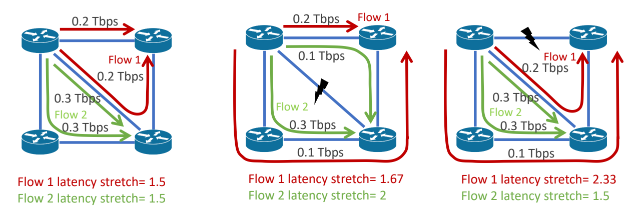
```
 1: procedure CALCULATE DEMAND LOSS, AVAILABILITY AND LATENCY STRETCH FOR QOS CLASS $q$ UNDER FAILURE SCENARIOS $S$
    ▷ Input: $S$: set of considered failure scenarios
    ▷ Input: $T$: set of Traffic Engineering tunnels on the IP topology $G$
    ▷ Input: $F^q = \{f\}$: set of flows in QoS class $q$
    ▷ Input: $d_f$: bandwidth demand of flow $f$
    ▷ Output: $V^q$: QoS class $q$'s availability
    ▷ Output: $L^q_f = \{< L^{s,q}_f, s.probability >\}$: for flow $f$, a distribution of latency stretch $L^{s,q}_f$ per failure scenario and its failure probability
    ▷ Output: $X^q$: QoS class $q$'s demand loss
 2:     Initialize flow $f$'s availability $V^q_f = 1, \forall q, f$
 3:     Initialize flow $f$'s demand loss in scenario $s$: $X^{s,q}_f = 0, \forall q, f$
        ▷ Iterate on all failure scenarios in S
 4:     for all $s \in S$ do
            ▷ TE bandwidth allocation $b^{s,q}_f$ and per-tunnel split ratio $a^{s,q}_{f,t}$
 5:         $\{b^{s,q}_f\}, \{a^{s,q}_{f,t}\}$ = TrafficEngineering($G, T, F^q, s$)
 6:         for all $f \in F^q$ do
                ▷ Flow $f$'s bandwidth-weighted latency
 7:             $l = (\sum_{t \in T_f} t.rtt \times a^{s,q}_{f,t}) / (\sum_{t \in T_f} a^{s,q}_{f,t})$
                ▷ Flow $f$'s latency stretch
 8:             $L^{s,q}_f = l / \min_{t \in T_f} t.rtt$
 9:             $L^q_f$.append(<$L^{s,q}_f$, $s.probability$>)
10:             if $b^{s,q}_f < d_f$ then
                    ▷ Flow $f$'s demand loss
11:                 $X^{s,q}_f = X^{s,q}_f + (d_f - b^{s,q}_f)$
                    ▷ Flow $f$'s availability reduction
12:                 $V^q_f = V^q_f - s.probability$
13:     $V^q = \min_{f \in F^q}(V^q_f)$
14:     $X^q = \max_{s \in S}(\sum_{f \in F^q} X^{s,q}_f)$
15:     return $V^q$, $L^q_f$, $X^q$
```

3 and 4 use a combination of K-Shortest Paths (KSP) and Multi-Commodity Flow (MCF) algorithms with the objective of minimizing the maximum link utilization in the network. We pre-assign TE tunnels for traffic flows between each router pair, then rely on a Linear Program (LP) to load-balance the traffic over all tunnels. Lower priority traffic uses the bandwidth left by higher priority traffic on each link. When failures bring certain links down, traffic is automatically re-distributed across remaining tunnels until the next TE execution where a new optimal traffic allocation is calculated.

3.2 Risk Metrics

This paper defines a set of key metrics to quantify a network's risk to potential failures. These risk metrics satisfy two requirements. First, they capture different aspects of failure events by quantifying the network's response from multiple dimensions. Second, since QoS classes have different levels of tolerance to failures, our risk metrics relate to QoS classes and reflect their SLO guarantees. As a result, we use the following metrics for each QoS class q:

(1) Demand Loss (X^q): For a failure scenario s, the demand loss is the total amount of lost traffic by all flows in q caused by s. The overall demand loss of QoS class q is the maximum, or worst-case, demand loss across all the considered failure


0.2 Tbps
Flow 1
0.2 Tbps
0.3 Tbps
Flow 2
0.3 Tbps
Flow 1 latency stretch= 1.5
Flow 2 latency stretch= 1.5
0.2 Tbps
Flow 1
0.1 Tbps
Flow 2
0.3 Tbps
0.1 Tbps
Flow 1 latency stretch= 1.67
Flow 2 latency stretch= 2
Flow 1
0.2 Tbps
0.3 Tbps
Flow 2
0.3 Tbps
0.1 Tbps
Flow 1 latency stretch= 2.33
Flow 2 latency stretch= 1.5

(a) Scenario 1 (no failures) (b) Scenario 2 (diagonal link fails with MTBF = 3 years and MTTR = 20 hours) (c) Scenario 3 (top link fails with MTBF = 1 year and MTTR = 10 hours)

Figure 2: An example with two flows (green and red arrows) from the same QoS class q under three failure scenarios. Risk metrics computed by Algorithm 1 are as follows: worst-case demand loss (X^q) = 0.3 Tbps, worst-case availability (V^q) = 99.81%, flow 1's latency stretch (L_1^q) = {<1.5, 99.81%>, <1.67, 0.18%>, <2.33, 0.11%>}, and flow 2's latency stretch (L_2^q) = {<1.5, 99.81%>, <2, 0.18%>, <1.5, 0.11%>}.

scenarios $s \in S$.

(2) Availability (V^q): The percentage of time that a flow's demand is completely satisfied (100% admitted) across all failure scenarios reflects the availability of that flow. Similar to demand loss, we compute our availability metric as the lowest availability among all flows in QoS class q.

(3) Latency Stretch (L_f^q): For a failure scenario s, the latency stretch $L_f^{s,q}$ of flow f in QoS class q is the ratio of the average tunnel latency (weighted by the tunnel bandwidth assignments) divided by the shortest TE tunnel latency without failure. We use Round-Trip Time (RTT) as a proxy for tunnel latency but other metrics such as hop-count and fiber length can also be used. The overall latency stretch L_f^q across all failure scenarios is a distribution, represented as the latency stretch $L_f^{s,q}$ of each failure scenario associated with its time probability of occurrence.

Algorithm 1 describes how these metrics are calculated by RSS in detail. We first initialize each flow's availability and demand loss to 1 and 0, respectively (lines 2-3). For all considered failure scenarios $s \in S$, we execute the Traffic Engineering (TE) formulation to find the per-flow satisfied bandwidth $b_f^{s,q}$ and the per-tunnel traffic allocations $a_{f,t}^{s,q}$ for each flow (line 5). Then, we calculate the three risk metrics using the outputs of TE. Flow f's latency stretch is calculated as the bandwidth-weighted latency l divided by the minimum latency across all tunnels (lines 7-8). Flow f's demand loss is captured by the difference between satisfied bandwidth $b_f^{s,q}$ and flow's demand d_f (line 11). If the demand is not fully satisfied, availability is reduced by the probability of the failure scenario s (line 12). Finally, we select the availability of QoS class q to be the worst availability experienced by all flows $f \in F^q$ (line 13), and demand loss to be the maximum loss across all scenarios $s \in S$ (line 14).

Figure 2 shows an example of the risk metrics computed by Algorithm 1 for two flows from the same QoS class under three failure scenarios. The healthy state is shown in Figure 2(a) and labeled as scenario 1. Figure 2(b) illustrates scenario 2 where flows 1 and 2 experience 0.1 Tbps and 0.2 Tbps of demand loss, respectively. Hence, the total loss for this QoS class adds up to 0.3 Tbps. In scenario 3, shown in Figure 2(c), only flow 1 loses 0.1 Tbps traffic, so the total loss is 0.1 Tbps. The demand loss for this QoS class is thus 0.3 Tbps — the highest loss across all scenarios. To obtain the availability for this QoS class, we first compute the availability of each flow. The demand of flow 1 is fully satisfied in only the no-failure case (scenario 1). As a result, its availability is computed as the probability (fraction of time) that the network is healthy: $1 - \frac{10\ hours}{1\ year + 10\ hours} - \frac{20\ hours}{3\ years + 20\ hours} = 99.81\%$. The demand of flow 2 is fully satisfied in scenarios 1 and 3, hence its availability is $1 - \frac{20\ hours}{3\ years + 20\ hours} = 99.92\%$. The availability of this QoS class is the lowest availability across both flows, which is 99.81%. To simplify the calculation of latency stretch in this example, we assume the latency of each link to be 1. Because the shortest tunnel latencies for both flows are 1, their bandwidth-weighted latency stretches in the healthy state (scenario 1) are both 1.5, as shown in Figure 2(a). In scenario 2 (Figure 2(b)) the latency stretch values for flows 1 and 2 are $\frac{0.2\times1+0.1\times3}{0.2+0.1} = 1.67$ and $\frac{0.1\times2+0.3\times2}{0.1+0.3} = 2$, respectively. Similarly, in scenario 3 (Figure 2(c)) the latency stretch values for flow 1 and 2 are $\frac{0.2\times2+0.1\times3}{0.2+0.1} = 2.333$ and $\frac{0.3\times1+0.3\times2}{0.3+0.3} = 1.5$, respectively. We then associate each failure scenario's probability to the corresponding latency stretch value to construct the latency stretch distributions.

3.3 Failure Modeling

High-fidelity failure modeling is important for network planning to meet SLOs. Hence, modeling failure scenarios is an essential component in calculating the risk metrics that we defined in the previous section. The goal of failure modeling is to estimate the likelihood of a failure scenario as well as the duration of the failure event. In this section, we explain Facebook's production failure model.

3.3.1 Characterizing Failure Events

We use two main variables to characterize failure events in our backbone:

(i) ***Time Between Failures*** *(TBF)* represents the duration between the recovery and the occurrence of two consecutive failures. This metric captures how reliable a network component (such as switch, linecard, or a fiber path) is. For most components in our backbone network, TBF tends to be thousands or even tens of thousands of hours.

(ii) ***Time To Repair*** *(TTR)* measures how long each failure event lasts. This metric depends on the efficiency of the network operation. Some failures (e.g., subsea fiber cuts) are more difficult to repair than others (e.g., switch failures).

Our experience indicates that fiber-related failures in our backbone are the most devastating failure scenarios in terms of capacity loss and time to repair. As a result, our primary focus to model failures is on fiber-related issues.

Each fiber i under each failure scenario j is represented with a tuple $(TBF_{i,j}, TTR_{i,j})$. We use historical data analysis to estimate the values in each tuple. However, modeling every fiber in the backbone individually adds excessive complexity and will overwhelm the system. We need an intelligent clustering method to model fibers with similar features together. Moreover, we cannot completely rely on empirical observations. For instance, newly deployed fibers do not have historical failure data. As a result, we model $TBF_{i,j}$ and $TTR_{i,j}$ based on known features such as the length of the fiber and its supplier. The next section describes how we address these challenges.

3.3.2 Capturing Common Features and Data Sources

A naive approach to model failures is to use past failure events to compute TBF and TTR from historical data. However, there are practical challenges with this approach. First, rare failure events may not have enough historical data to faithfully compute their TBF and TTR. Second, data can be noisy. In particular, repair times are often recorded manually in our ticketing system, which may not be completely accurate. Third, data sources may belong to different administrative domains. For instance, leased fibers are operated by third-party vendors and we may not have access to the complete failure data. To address these challenges, we use a combination of common features and several data sources to model the failure characteristics of each fiber as accurately as possible.

Common Features. Each fiber is different, but there are common features that we can use to characterize a fiber without having its exact TBF and TTR. Below are the failure features we use in our system.

- *Fiber length:* Longer paths are more likely to experience fiber cuts due to greater surface area.
- *Vendor:* Fibers from certain vendors are more reliable than others, depending on their physical characteristics, operation quality, and contractual obligations to us.
- *Operational ownership:* Some fibers are purchased from the builder directly, while others may be leased or bought from indirect parties. We expect that without direct access to the fibers, subcontracted fibers have longer repair times.
- *Install type:* Subsea fibers are known to have longer repair time due to the difficulty in accessing the fiber or limited supply of maintenance ships. Similarly, aerial fibers are expected to have higher failure rates compared to buried fibers because the fiber is exposed to disasters and accidents.
- *Geographical region:* Failure rates can be higher in certain areas with frequent natural disasters, e.g., hurricanes. Repair times vary based on the weather condition, and can grow because of catastrophic events.

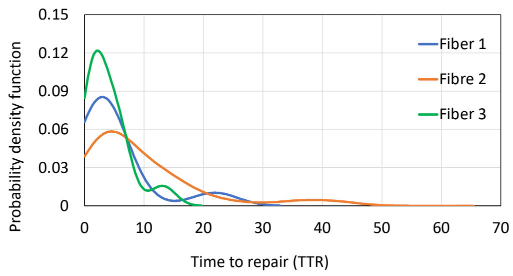


Figure 3: Distribution of time to repair for subsea fibers.

- *Urban density:* Fibers are more likely to be impacted in urban settings due to more frequent human activities, and hence accidental fiber cuts.
- *Shared Risk Link Group (SRLG):* Some fibers may fail together due to shared conduit or geographical proximity. An SRLG is considered as a single entity, hence, in our risk simulation system, we consider each SRLG as a single failure scenario.

Data sources. The impact of each feature can be analyzed using real-world data. We use three data sources for this purpose. (*i*) *Operational tickets:* Our Network Operation Center (NOC) maintains hundreds of incidents ticketed to our vendors. Each ticket contains confirmed failure information such as failed links, downtime, and failure root causes. We use this service as a historical benchmark for availability. However, the data is manually maintained, hence, the accuracy and coverage are both limited. (*ii*) *Continuous measurements:* We monitor counters from both IP-layer switches and optical-layer transponders and ROADMs. IP counters are collected every minute via the standard monitoring protocol SNMP. Optical counters are collected every three minutes using our optical-layer monitoring protocol TL1. To identify failures, we look for the Loss of Signal (LOS) on the Optical Service Channel (OSC) accompanied by the loss of IP links. (*iii*) *Fiber lifetime:* The above data sources both report discrete failure incidents, yet some fibers may not have failure events in recent years. Thus, we use the fiber lifetime dataset from our fiber inventory to compute the uptime of each fiber.

3.3.3 Failure Modeling Framework

We develop a failure modeling framework to best utilize the above data for estimating the failure model parameters. In particular, we take a two-step process.

1. Clustering. We start with a list of fibers and their failure characteristics from the data sources described in the previous section. Each record contains $< f_1, f_2, \ldots, f_k, TBF_t, TTR_t, TBF_d, TTR_d >$, where f_k is the k^{th} element in the feature set, TBF_t and TTR_t are the TBF and TTR values from the tickets, and TBF_d and TTR_d are those from the continuous measurements. We then use a Bayesian clustering algorithm to identify groups of fibers that share

	MTTR	MTBF
Subsea fibers vs. non-subsea fibers	90×	36×
Leased fibers vs. non-leased fibers	1×	0.4×
Fibers in the most different region vs. fibers in other regions	2×	5×

Table 2: MTTR and MTBF of different fiber categories.

similar failure characteristics. The output of this step is a set of clusters $(C_1, \dots, C_g)$, where each C_i contains a set of fibers.

2. Bayesian Hierarchical Model. Next, for each cluster, we use an exponential hierarchical model to fit the distribution of TBF and TTR separately. We find the mean TBF (MTBF) and mean TTR (MTTR) from both fitted curves and use them in RSS (§3.4). The accuracy of this model is evaluated in §4.1.

Operational observations. In the following, we summarize some of our empirical measurement results that have provided inspirations for our failure modeling.

Subsea TTR follows arbitrary distribution. Figure 3 shows the TTR distributions of three subsea fibers from our empirical failure data source. Each subsea fiber has a unique TTR distribution, due to its physical properties such as the length and placement under the ocean that determines the accessibility for repair endeavors. This observation deviates from a common technique to use a simple exponential distribution for TTR in two major ways. First, unlike exponential distribution, there is a lower bound for the TTR. This lower bound corresponds to the physical time constrains such as the time to secure permits to enter the water and the sailing time. Second, the distribution is multi-modal since each subsea fiber has distinct parts with different failure profiles depending on the depth under water.

Impact categories. We categorize three key factors that have significant impacts on the failure model: whether a fiber is subsea, leased, or belongs to a particular region. Table 2 shows the relative impact of different fiber categories on MTTR and MTBF. If a fiber is subsea, its MTTR is 90× longer than that of non-subsea fibers, and its MTBF is 36× longer. This is because subsea fibers are less frequently cut, but once they are cut, they will take much longer to be repaired. Leased fibers have similar MTTR as non-leased (Facebook-owned) fibers, but are 2.5× more likely to fail in terms of MTBF. For the region factor, we select the region with the largest difference from the rest ones, and we observe a 5× difference in MTBF and 2× difference in MTTR. These results show the drastic differences between fiber types and indicate the importance of clustering fibers into appropriate failure groups.

3.4 Risk Simulation System (RSS)

3.4.1 System Design

RSS performs periodic simulations (e.g., every 30 minutes) to report the risk metrics defined in Section 3.2. Figure 4

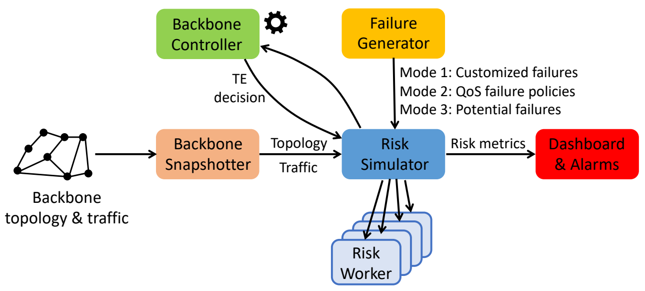


Figure 4: Risk Simulation System architecture.

depicts our risk simulation pipeline. For each simulation run, the *Backbone Snapshotter* polls the backbone routers for the latest IP topology and traffic demand. The *Failure Generator* generates hypothetical failure scenarios to be simulated. The *Risk Simulator* takes in such information to simulate routing on the residual topology under different failures. To simplify system implementation, we reuse the binary of our centralized *Backbone Controller* which computes the TE solution using a global optimization formulation. Since routing simulations for different failure scenarios are independent, we shard them onto a number of *Risk Workers* for parallelization. The risk metrics are calculated from the worker instances and displayed on a real-time risk dashboard. Risk values higher than pre-defined thresholds will raise production alarms.

Calculation of the risk metrics requires failure probabilities, which can be derived from MTBFs and MTTRs — the mean values of the failures' TBF and TTR distributions in Section 3.3. For most failures, the TBF and TTR follow exponential distributions. However, the TTR for subsea fibers is arbitrary and hard to model. As a result, we estimate their metrics from our empirical failure observation in production.

Suppose a failure scenario includes n failed fibers $\{f_1, f_2, ..., f_n\}$. For a particular fiber f_i, the probability of being available is $A(f_i) = \frac{MTBF_i}{MTBF_i + MTTR_i}$, and the probability of being under failure is $P(f_i) = 1 - A(f_i)$. Therefore, the probability of the entire failure scenario is $P(f_1, f_2, ..., f_n) = \Pi_{i=1}^{n} P(f_i)$, and the available probability is $A(f_1, f_2, ..., f_n) = 1 - \Pi_{i=1}^{n} P(f_i)$. Given the failure probability of each failure scenario, the risk metrics can be calculated using Algorithm 1.

RSS supports three modes of operation. Mode 1 is for fine-grained simulation of customized failures that are expected to happen in the near future. For example, it is part of our decommission workflow where capacity is removed, or migrated from one fiber to another, according to the backbone expansion plan. Before decommission tasks are carried out, RSS is used to generate failure scenarios that reflect the decommission plan. The risks associated with these scenarios are then taken into account to ensure there is sufficient protection capacity in the network. Mode 1 also serves for risk monitoring and mitigation under natural disasters. For instance, in response to a hurricane forecast, we simulate failure scenarios relating to the hit regions and shift traffic as necessary. Similarly, as COVID-19 goes on, we plan to simulate failure scenarios for specific geological locations based on the severity of the pandemic.

Mode 2 is fine-grained simulation of pre-defined failure scenarios for different QoS classes given their protection policies. In production, the protection policies include four categories of critical fiber failures: (1) single fiber failures, (2) SPOFs where multiple SRLGs use fibers in the same conduit or have the same geographical proximity, (3) dual subsea failures where two subsea fiber paths fail simultaneously, and (4) dual DC failures where two fiber paths from the same DC fail simultaneously. These four categories include over 6000 failure scenarios in the Facebook backbone. As described in Section 3.1, different QoS classes protect against different failure categories. QoS classes 1 and 2 carry our critical services and have full protection against all the above failure categories. QoS class 3 (default class for our internal traffic) relaxes on dual DC failures, because they account for over 50% of the failure scenarios but are less likely than single fiber failures and SPOFs and have less severe consequences compared to dual subsea failures. QoS class 4 (background bulk data transfers) is best-effort service without failure protection. We use Mode 2 in RSS to validate the QoS performance and guide network maintenance.

Mode 3 is coarse-grained simulation of a large number of potential failures in the backbone, where the exact number is determined by a cutoff threshold. The cutoff threshold can be defined in different forms, such as by failure probability, the number of concurrent failures, MTTR, or the protection cost (in terms of the protection capacity, construction cost, and maintenance work), under the intuition that we value failures that are more likely to happen, take a longer time to repair, or are affordable to protect against. We typically have a quick scan of the network health considering *millions* of failure scenarios. This simulation mode must be coarse-grained given the large number of failure scenarios. We bypass the computation-heavy global TE optimization with efficient routing approximations, which will be discussed in Section 3.4.2. This mode of operation offers a tradeoff between simulation accuracy and runtime, and the choice depends on the number of failure scenarios and how close to production the simulation needs to be (e.g., replaying production situations in Mode 1 and 2 vs. a big picture of the network in Mode 3).

3.4.2 System Optimizations

RSS is implemented using around 18,000 lines of C++ code. This system is highly optimized for fast execution time. Today, it can finish a fine-grained simulation of one failure scenario in an average of 250 seconds and a coarse-grained simulation of a failure scenario in 0.1 second. Important performance improvements attribute to the following optimizations.

Parallelization. Our risk simulation is highly parallelizable by nature. Our first implementation was based on a two-layer master-slave architecture where the failure scenarios were distributed across the slave nodes and the simulation results were aggregated to the master. The master node was overwhelmed with the aggregation load when we scaled to 50 slaves, hence we added another layer in the middle to aggregate the intermediate results generated by slaves and then transmit them to the master. Today, we use tens of aggregators and hundreds of slave nodes to optimize the execution time of RSS.

Routing simplification. Our fine-grained simulation emulates the production backbone by executing the TE algorithm when a failure happens. This process is computationally expensive, especially when we simulate a large number of failure scenarios. Thus, for coarse-grained simulations, we simplify the TE implementation with shortest-path routing of small units of sub-flows. Specifically, we split each traffic flow in the backbone demand matrix, usually hundreds of Gbps or several Tbps big, into minimal sub-flows around 1 Gbps and pack them one by one onto the shortest path until there is no remaining capacity in the network. The result is close to production TE when the sub-flows are sufficiently small.

Merge duplicate failures. Different fiber failures can result in the same failure scenarios on the IP layer, which can be effectively merged during risk simulation. For example, failures of different SRLGs may cause different fiber spans to be down, but they create the same failure scenario for the IP links over the fiber paths traversing any of these fiber spans. Because the risk is ultimately simulated on the IP-layer network, RSS translates the fiber failures into IP-layer failures and merges duplicate failure scenarios. The failure probability of a merged failure scenario equals to the sum of the probabilities of each individual failure event.

Identify dominating failures. We further reduce the simulation time by only simulating failures with severe consequences. We define dominating failures as the ones that contain subsets of other failures. For example, the failure scenario with fiber cuts $\{f_1, f_2\}$ is a dominating failure of single fiber failures f_1 and f_2 alone. Note that this simplification only applies to the calculation of demand loss, which does not rely on the probabilities of failure scenarios (the other two risk metrics, latency stretch and availability, need to factor in failure probability). This is because the failure probability of a dominating failure is much smaller than the probabilities of its subset failures. In production, we usually use this approach for the Mode 1 simulation, where the demand loss of expected failure events is a critical signal for network maintenance.

4 Evaluation

In our daily operation, we keep monitoring the health of the backbone network with the risk metrics (Section 3.2) produced by RSS (Section 3.4). Here we report the risk measurement results from November 2019 to September 2020

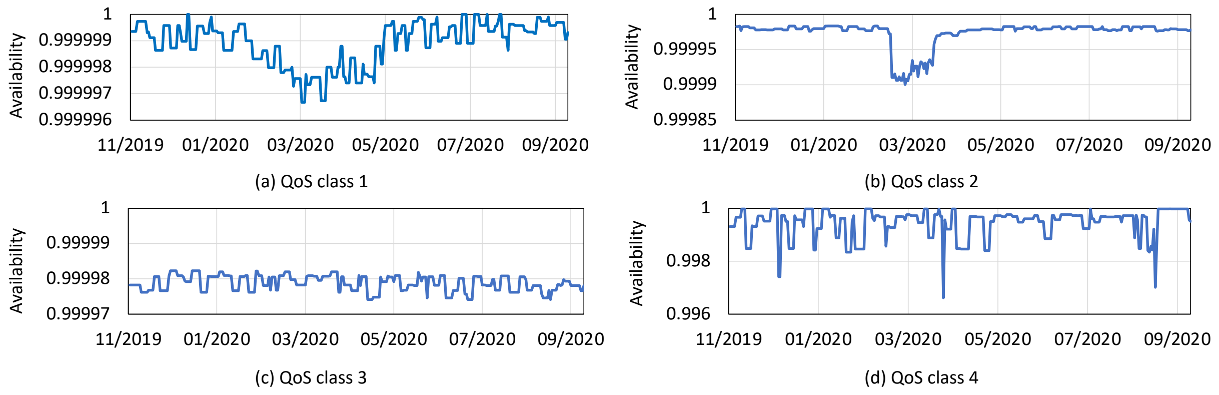


Figure 5: Availability over time for each QoS class.

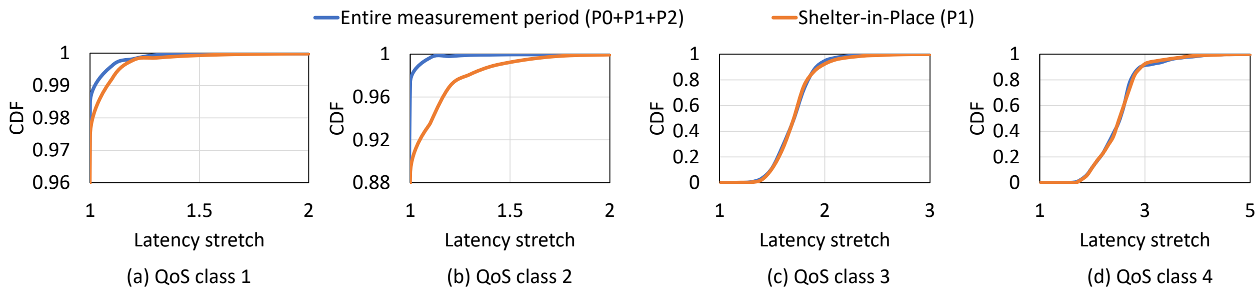


Figure 6: CDF of per-flow latency stretch for each QoS class.

and share our operational experience to survive the extreme conditions under COVID-19.

4.1 Observations with RSS

An important input into RSS is production traffic. During COVID-19, we observed an increase in network risk triggered by significant traffic surges (Figure 1). However, most risk metrics remained at a reasonable level, indicating that our backbone was robust under the COVID-19 stress test.

High availability. As shown in Figure 5, different QoS classes all achieved high availability over time, constantly reaching the SLO goals. The traffic increase during social distancing mostly related to the user traffic in QoS class 2, causing its availability to drop sharply from around 0.99998 to 0.9999. QoS class 1 also experienced a minor availability reduction, as the traffic for highly critical user services increased as well. The change was smoother compared to QoS class 2, because QoS class 1 also contains system control traffic irrelevant to user behaviors. QoS classes 3 and 4 that mostly comprise machine-to-machine computational traffic showed no obvious decrease in availability. These results suggest that availability is highly sensitive to traffic volume. On the positive side, our backbone infrastructure is over-provisioned, making it robust against the unprecedented traffic surge.

Low latency stretch. From Figure 6, we see a minimal change of latency stretch during the shelter-in-place period (P_1). Recall from Section 3.1 that QoS classes 1 and 2 use CSPF routing. As shown in the figure, over 97% of the flows in QoS classes 1 and 2 had a latency stretch of 1 throughout the entire measurement period ($P_0 + P_1 + P_2$), meaning they went through the shortest paths. The latency stretch of QoS class 2 was slightly higher than that of class 1, because the bandwidth for QoS class 2 is allocated after class-1 flows are fully accommodated. Similar to the availability results, latency stretch degraded the most in QoS class 2 during the shelter-in-place period (P_1) due to the traffic increase. Yet, the stretch still remained low regardless of the COVID-19 increase: it stops at 1.7 for most flows in QoS class 2, and at 1.4 for QoS class 1, though with a long tail not shown in the figures. QoS classes 3 and 4 use a combination of KSP and MCF routing, so they generally take longer paths. The mean latency stretch for QoS class 3 was 1.71, and 2.53 for QoS class 4. COVID-19 caused little increase of latency stretch for these two traffic classes, which is consistent with the trend of traffic growth and availability change.

Accurate failure modeling. We evaluate the accuracy of our failure model (§3.3) by comparing the TTRs and TBFs of observed fiber failures in North America against our model's predictions. As shown in Figure 7, our failure model is close to the actual observed values. To quantify the difference between prediction and observation, we perform a Kolmogorov-Smirnov (KS) test [1] on the null hypothesis that the measurements and the model-generated samples are drawn from the same distribution. We report the KS statistic and p-value in Table 3. Both p-values are large, meaning the two distributions match. Lastly, we directly compute the prediction accuracy, as the difference between each observed and predicted value,

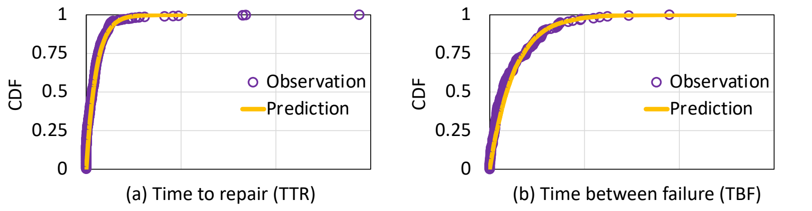


Figure 7: CDF of TTR and TBF from failure prediction vs. observation (the values on the x-axis are anonymized).

	KS stats	$p-$value	accuracy
TTR	0.05	0.25	94%
TBF	0.03	0.47	98%

Table 3: Kolmogorov-Smirnov (KS) test statistics and accuracy of TTR and TBF models.

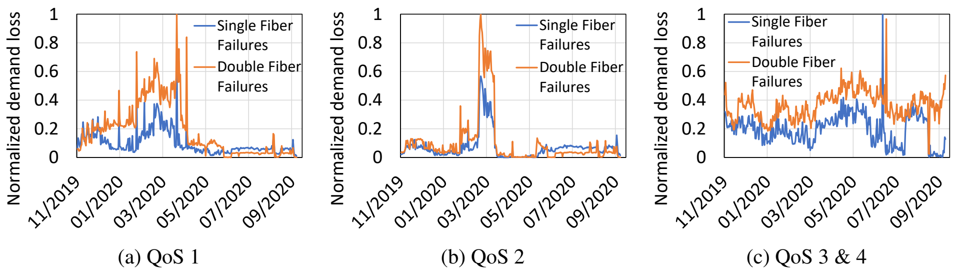


Figure 8: Normalized demand loss per QoS class.

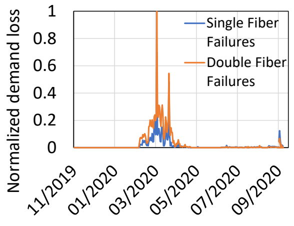


Figure 9: QoS class 2 downplayed into class 3&4.

divided by the observed value. Our model achieves 94% accuracy for TTR and 98% accuracy for TBF prediction.

4.2 Risk Mitigation

Demand loss as a guide. Demand loss is a rigid risk metric, which captures the highest traffic loss across all simulated failure scenarios. It guides our operations for mitigating potential risk. Figure 8 shows the demand loss for each QoS class over time. For confidentiality reasons, we normalize the numbers in each QoS class against the highest loss value. We group different types of failures that we track into two categories: single fiber failures and dual fiber failures. These categories capture the major failures we protect against in production. The figure shows that the demand loss increased during the P_1 phase for QoS classes 1 and 2. In particular, the mean value of risk during the shelter-in-place phase (P_1) increased by 80% compared to the pre-COVID period (P_0) in QoS class 1, and by 3.6× in QoS class 2. QoS classes 3 and 4 did not have a significant change in their demand loss values during the pandemic, and the loss increase in March 2020 was due to traffic migration between regions because of an internal policy change. Note that dual failures, though less common in practice, induce 2.14× higher loss on the fabric than single failures, on average. This worst-case analysis makes us operate the network conservatively, which was especially beneficial during the pandemic period.

QoS downplay. Another key technique to save capacity for the most critical traffic is to adjust the QoS assignment. By default, all traffic from a service is assigned the same QoS class. However, a service usually contains traffic from both user requests and system metadata, whose importance should be differentiated. This coarse-grained performance isolation causes over-protection of unimportant traffic, and the resulting capacity waste should be recycled for traffic increase. We have developed an internal system that leverages inference mechanisms to identify the true traffic priorities and correct their QoS labeling. For example, we find that on-off and periodic traffic patterns are common signals for machine-created traffic, which can be downgraded to a lower class. Figure 9 zooms into QoS classes 3 and 4 in Figure 8(c) and shows the demand loss of the downplayed traffic. The loss only appeares during the shelter-in-place phase (P_1) as the result of traffic shift to alleviate the stress from QoS class 2.

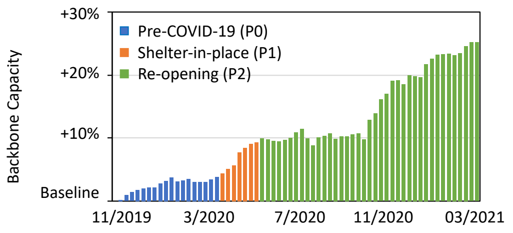
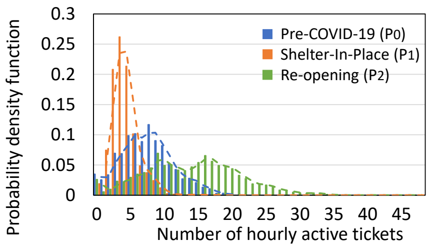
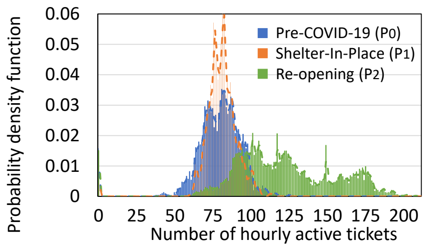


Figure 10: Backbone capacity measured per week.

Proactive capacity enhancement. We deploy optical wavelengths periodically to augment the capacity of our backbone. Figure 10 shows the weekly measurement of backbone capacity over a year, normalized by the capacity value before P_0. We observe an aggressive capacity increase starting from the P_1 phase compared to the pre-COVID times (P_0). This capacity increase is also visible in Figure 8, leading to a significant drop in demand loss in April and May 2020. We observe a continued capacity increase during the re-opening phase (P_2). Although social distancing during the COVID-19 pandemic paused most of our site work for deploying new



(a) Optical layer



(b) IP layer

Figure 11: Impact of COVID-19 on hourly active failure tickets.

Failure category	Mean # of tickets			p−value for
	P_0	P_1	P_2	P_0 and P_1
IP	80.56	80.28	121.4	0.44
Optical	7.67	3.72	14.0	$\ll 0.001$

Table 4: Statistical comparison on the number of active tickets during our measurement phases.

fibers, we had sufficient dark and under-provisioned fibers from previous planning cycles. Our "plan ahead" strategy gave us enough headroom for emergency capacity enhancement, and our fully-automated optical management system allowed us to provision wavelengths remotely.

5 Insights on RSS Improvement

Besides a stress test of our network infrastructure, COVID-19 and the resulting social distancing also created unusual situations beyond normal assumptions of network operations. These edge cases have given the networking community a unique opportunity to rethink fundamental design assumptions of networks. In this section, we share our recent progress on failure modeling and traffic forecasting to shed light on future evolution of risk simulations.

5.1 Responsive Failure Modeling

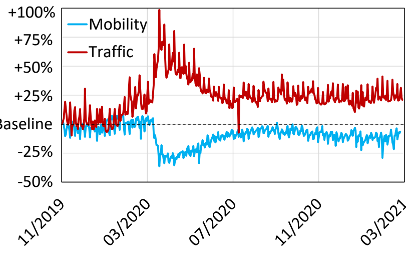
Our failure modeling (Section 3.3) uses years of failure measurement data to ensure model accuracy and stability, and it is accurate in the long run (Figure 7). However, prior studies claimed network failures are often caused by human activities and network operations [19, 20, 29, 30, 38]. Indeed, we confirm that the lack of human activities during the shelter-in-place phase (P_1) as well as the increase of network upgrade and capacity augmentation activities during the re-opening phase (P_2) changed the failure characteristics. In response to the change, we recalibrated our failure model to increase the weight of failure statistics during the P_0, P_1 and P_2 periods.

The recalibration relies on Facebook's centralized failure ticketing system, which automatically detects network failures and infers possible root causes. For this purpose, we categorize failure tickets into two groups: (*i*) optical-layer failures (e.g. fiber/amplifier/transponder issues) and (*ii*) IP-layer failures (e.g. router/interface down). For each group, we record the number of active failure tickets every hour and plot the probability density function of each phase in Figure 11. Our observations are as follows.

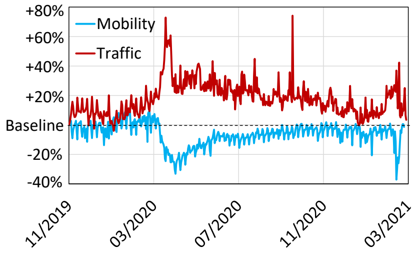
Optical-layer failures. Figure 11(a) compares the probability density function of hourly active optical-layer failure ticket numbers among the pre-COVID (P_0), shelter-in-place (P_1) and re-opening (P_2) phases. The dashed lines in the figure represent the moving average of each phase. We observe reduced optical-layer failures during P_1 compared to P_0. We attribute this finding to the significant reduction in human activities at our backbone facilities during global social distancing. For instance, limited construction work can lead to fewer fiber cuts, and fewer human contractors on-site may result in fewer accidental link flaps, as suggested in prior work [19]. We observe more active optical failure tickets in the re-opening phase (P_2), as our network operation team was actively augmenting the capacity of our backbone.

IP-layer failures. Figure 11(b) shows the probability density function of hourly active IP-layer ticket numbers during the P_0, P_1 and P_2 phases. In contrast to optical-layer failures, we find no significant changes in IP-layer failures between P_0 and P_1, and the two distributions largely overlap with each other. This is likely because IP-layer tickets, such as router/interface hardware failures, are mostly mechanical issues and do not correlate with human activity. However, similar to the optical-layer failures, in the re-opening (P_2) phase, the IP layer also experienced more active failure tickets due to our aggressive capacity provisioning operations. Moreover, since the pandemic continued to enforce limitations on our failure repair staff, the P_2 phase suffered from longer repair times.

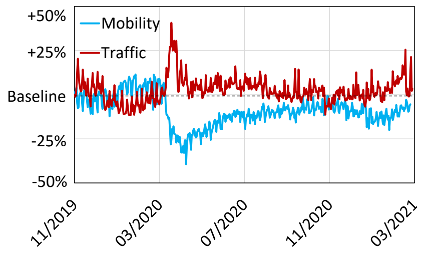
Confirmation with statistical hypothesis test. The results in Figure 11 suggest that optical and IP-layer failure distributions behave differently during the P_1 phase. To confirm this observation, we apply a statistical hypothesis test on the time-series distribution of active number of tickets between the P_0 and P_1 timelines and set the null hypothesis to be: *the means of the distributions are the same*. We apply Welch's t-test on the optical and IP-layer categories separately to validate the null hypothesis. Table 4 reports the mean number of active tickets and the corresponding p−values for each category. Considering a p−value threshold of 0.01, the null hypothesis cannot be rejected for IP-layer tickets, suggesting the pandemic did not have a significant impact on the IP-layer failures during P_0 and P_1. In contrast, the null hypothesis is rejected for optical tickets suggesting that the average number of active optical tickets changed in a statistically significant manner. Note that we do not run hypothesis tests over P_2 because its mean values in both the IP and optical layers differ a lot from those of P_0 and P_1, which already indicates



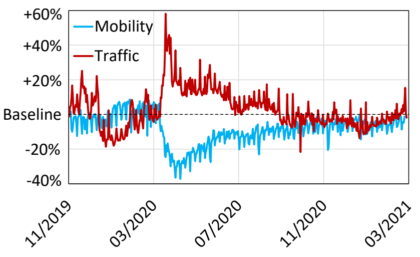
(a) Chicago



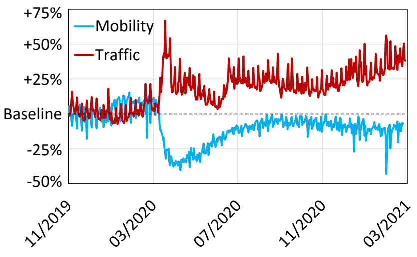
(b) Dallas



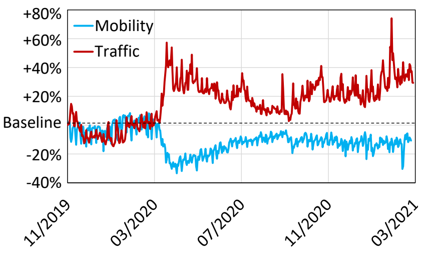
(c) Los Angeles



(d) Miami



(e) New York City



(f) Seattle

Figure 12: Traffic volume and mobility patterns in six US cities during the COVID-19 pandemic.

differences in the probability distributions.

Insights. These findings call for responsive failure modeling. When special events cause failure characteristics to change, the failure prediction model should be adjusted to rely more on recent failure measurement points. However, the model stability might be at stake with short-term data collection, and the challenge lies in balancing stability and agility to have an accurate model. The COVID-19 crisis required us to respond quickly, and the fast development of the pandemic gave us little time for drastic redesigns of the failure model. The various failure generation modes in RSS make it adaptive to different failure models. For instance, our customized failures in Mode 1 are designed for failure scenarios of particular interests that may deviate from the failure model. We leveraged this feature to feed RSS with short-term failure statistics for close monitoring of the network health during COVID-19. The failure model is hard to change in Modes 2 and 3, hence, we applied a scaling factor to the failure distributions to generate more failures. These false positives helped us operate the network more cautiously during the pandemic. Moving forward, we are working on more responsive failure modeling with a sliding window that automatically assigns weights to different measurement periods.

5.2 Traffic Prediction with External Signals

The risk observations we report throughout this paper use current production traffic as input, yet RSS can also take in projected traffic to forecast future risk. At Facebook, we perform demand forecasting every quarter to predict the traffic volume in the next 6 months to one year. Our prediction used to be accurate, but the traffic grew beyond the predicted upper bound since the pandemic started. At the peak, we saw a 26% difference between actual and predicted upper-bound traffic.

In our operational experience, we have seen rich examples of how external non-networking signals can be leveraged to aid network management. For instance, it is common practice to strengthen the guard on PoP or DC regions that have received hurricane warnings, and we keep a close watch on traffic blackholing in areas with frequent armed conflicts. In this section we demonstrate that human *mobility* metrics can also be used to better predict the traffic volume.

To demonstrate this finding, we use population mobility data from the SafeGraph [5,41] database built from 20 million mobile devices. As an approximation of mobility, we sum the total number of trips that take place in a geographical region based on aggregated cellphone GPS data, and normalize it by the population of the trip origin. We consider six major US cities and plot the variations of traffic and mobility over time in Figure 12. Interesting findings imply opportunities and challenges in our proposal of mobility-aided traffic prediction.

Negative correlation between traffic and mobility. Figure 12 shows the traffic volume and mobility rate normalized to their corresponding averages during the P_0 phase. While there is a fair amount of overlap between traffic and mobility in P_0 across the cities, we observe a strong negative correlation between traffic and mobility since the start of P_1. We see the general trend that when mobility drops, traffic increases; and as mobility increases slowly, traffic falls as well. The sporadic spikes of mobility and traffic also match well, forming zigzags in opposite directions. The gap between the traffic and mobility curves closes down in the P_2 phase when the cities started to re-open and social distancing reduced.

Variations across cities. Each city shows some uniqueness despite the same trend. Chicago, Dallas, Los Angeles, and Miami have similar patterns that network traffic gradually decreased while mobility continued to increase after the pandemic peak in mid-Mach 2020. In Chicago, traffic increased by 99% and mobility rate decreased by 36% during the shelter-in-place phase (P_1) compared to the P_0 phase baseline. Dallas, Los Angeles, and Miami had around a 40-70% traffic increase and a 35% mobility drop. Roughly, the drop in mobility rate corresponds to different levels of traffic reduction across cities. For example, in Miami and Los Angeles the traffic volume almost returned back to normal in the P_2 phase, while Chicago still showed a 25% average traffic increase, and Dallas had around 10% average traffic increase. On the other hand, New York City and Seattle have contrasting patterns, with both an uptake in traffic and a downtake in mobility appearing since November 2020. For Seattle, we also observe ups and downs in traffic volume, with occasional spikes up to 74%.

Insights. An intrinsic limitation of traditional network management is the complete reliance on in-network signals. It fails to track social influences on the network infrastructure, which has been proven to be heavily underestimated during COVID-19. Our mobility case study shows the potential benefit of embracing offline signals from the outside world. On occurrences of social events, we can make mobility the main signal for traffic prediction. As Figure 12 shows, traffic peaks can be inferred from the steep drops in mobility rate. However, it is challenging to estimate the traffic volume when it recovers, as mobility rate does not show significant changes in that case. We may need other signals to understand user behaviors better. Fortunately, though, risk management cares about worst-case scenarios. Thus, an accurate prediction of the peak traffic is already a big win for risk prevention.

6 Related Work

Risk-aware network management. There have been recent proposals to apply risk to capacity planning [4, 6] and traffic engineering [8, 12, 31, 46]. The definition of risk is different across proposals, including probabilistic models of failures [6, 8], revenue shortfall [31], early signs of failures (e.g., hardware abnormality and performance degradation) [46], user-specified undesirable events and their associated probabilities [12], and the likelihood of losing customer traffic during planned network changes [4]. In this paper, we describe Facebook's definition of risk as a set of SLO-related metrics that quantify the impacts of potential failures. We are also the first to apply risk simulation to backbone management and to develop a production system.

Internet under COVID-19. There are reports on the impacts of COVID-19 on the Internet [10, 13, 17, 24, 32, 33], as well as measurement studies on the PoP traffic [7], mobile traffic [28], Internet traffic [15], and cybercrime [47] during the pandemic. We share similar observations on the traffic increase, but we present a comprehensive study on the impact of COVID-19 from the perspective of risk management.

Backbone failures. Prior work on understanding backbone failures includes statistical modeling [3, 9, 42] and real-world measurements on both the optical layer [18, 19, 38, 43] and the IP layer [11, 14, 23, 25, 26, 29, 30, 35]. Our failure analysis confirms the observation in previous papers that a good proportion of failures are human-related [19, 20, 29, 30, 38].

Traffic classification with QoS. QoS can provide differentiated performance for different categories of traffic [16, 48]. There have been rich discussions on traffic classification methods in the prior work [36, 39, 40, 44]. In recent years, with the development of SDN, traffic classification can be deployed with centralized control on a private enterprise network [34]. Facebook follows these discussions and categorizes the backbone traffic into four classes of QoS. To maximize user satisfaction while considering network risks, our traffic classification scheme can dynamically adjust traffic flows' QoS categories so as to prioritize critical traffic flows and guarantee service level objectives.

7 Conclusion

This paper introduces RSS, a risk simulation system deployed at Facebook. We present our risk analysis with RSS during the COVID-19 pandemic period and beyond. Motivated by the surge of traffic volume, we define risk metrics to quantify the impact of COVID-19 and show our strategies to mitigate the risk. We keep the network running at low risk levels during this challenging time and propose that having responsive failure modeling and using external signals such as human mobility can help understand the social impacts on the network to further improve network management. Our experience and insights are useful for managing large-scale backbones in the post-pandemic world, where we are likely to face an ever-growing demand for online services. We hope that our experience can inspire and guide practitioners towards embracing risk-driven network management and ultimately making it a key strategy for ensuring high availability of networked services.

Acknowledgments. Many people in the Network Infrastructure team at Facebook have contributed to RSS over the years. In particular, we would like to acknowledge Max Noormohammadpour, Siyang Xie, Bob Kamma, Abhinav Triguna for their early work on RSS. We also would like to thank Arash Vakili, Ariyani Copley, Brian Bierig, Debottym Mukherjee, Lu Huang, Tyler Price, Sandrine Pasqualini for their operational support during the COVID-19 time period. We would like to acknowledge Facebook for the resource it provided for us. And finally, we thank Omar Baldonado, Gaya Nagarajan, our shepherd Haonan Lu, as well as our anonymous reviewers for their comments. This work was partly supported by NSF grants CNS-2008624 and ASCENT-2023468, as well as by SystemsThatLearn@CSAIL Ignite Grant.

References

[1] Kolmogorov-Smirnov test. http://www.mit.edu/~6.s085/notes/lecture5.pdf.

[2] State Shelter-in-Place and Stay-at-Home Orders, 6 2020. https://www.finra.org/rules-guidance/key-topics/covid-19/shelter-in-place.

[3] Anuj Agrawal, Vimal Bhatia, and Shashi Prakash. Network and Risk Modeling for Disaster Survivability Analysis of Backbone Optical Communication Networks. *Journal of Lightwave Technology*, 37(10):2352–2362, 2019.

[4] Omid Alipourfard, Jiaqi Gao, Jeremie Koenig, Chris Harshaw, Amin Vahdat, and Minlan Yu. Risk based Planning of Network Changes in Evolving Data Centers. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 414–429, 2019.

[5] Nick Altieri, Rebecca L. Barter, James Duncan, Raaz Dwivedi, Karl Kumbier, Xiao Li, Robert Netzorg, Briton Park, Chandan Singh, Yan Shuo Tan, Tiffany Tang, Yu Wang, and Bin Yu. Curating a COVID-19 data repository and forecasting county-level death counts in the United States, 2020. https://arxiv.org/abs/2005.07882.

[6] Ajay Kumar Bangla, Alireza Ghaffarkhah, Ben Preskill, Bikash Koley, Christopher Albrecht, Emilie Danna, Joe Jiang, and Xiaoxue Zhao. Capacity planning for the Google Backbone network. In *ISMP (2015)*, 2015.

[7] Timm Boettger, Ghida Ibrahim, and Ben Vallis. How the Internet reacted to Covid-19. In *Proceedings of the 2020 ACM SIGCOMM conference on Internet measurement conference*, 2020.

[8] Jeremy Bogle, Nikhil Bhatia, Manya Ghobadi, Ishai Menache, Nikolaj Bjørner, Asaf Valadarsky, and Michael Schapira. Teavar: Striking the right utilization-availability balance in WAN traffic engineering. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM 19, page 29?43, New York, NY, USA, 2019. Association for Computing Machinery.

[9] Graham Booker, Alexander Sprintson, E Zechman, C Singh, and S Guikema. Efficient traffic loss evaluation for transport backbone networks. *Computer Networks*, 54(10):1683–1691, 2010.

[10] Joseph Brookes. Akamai data shows 30 percent surge in Internet traffic, April 2020. https://which--50-com.cdn.ampproject.org/c/s/which-50.com/an-extraordinary-period-in-internet-history-akamai-data-shows-30-per-cent-surge-in-internet-traffic/.

[11] Alberto Dainotti, Claudio Squarcella, Emile Aben, Kimberly C Claffy, Marco Chiesa, Michele Russo, and Antonio Pescapé. Analysis of country-wide Internet outages caused by censorship. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, pages 1–18, 2011.

[12] Ferhat Dikbiyik, Massimo Tornatore, and Biswanath Mukherjee. Minimizing the risk from disaster failures in Optical Backbone Networks. *Journal of Lightwave Technology*, 32(18):3175–3183, 2014.

[13] Fastly. How COVID-19 is affecting Internet Performance, 2020. [Online; accessed 20-May-2020].

[14] Nick Feamster, David G Andersen, Hari Balakrishnan, and M Frans Kaashoek. Measuring the effects of Internet path faults on reactive routing. *ACM SIGMETRICS Performance Evaluation Review*, 31(1):126–137, 2003.

[15] Anja Feldmann, Oliver Gasser, Franziska Lichtblau, Enric Pujol, Ingmar Poese, Christoph Dietzel, Daniel Wagner, Matthias Wichtlhuber, Juan Tapiador, Narseo Vallina-Rodriguez, Oliver Hohlfeld, and Georgios Smaragdakis. The Lockdown Effect: Implications of the COVID-19 Pandemic on Internet Traffic. In *Proceedings of the 2020 ACM SIGCOMM conference on Internet measurement conference*, 2020.

[16] Victor Firoiu, J-Y Le Boudec, Don Towsley, and Zhi-Li Zhang. Theories and models for Internet Quality of Service. *Proceedings of the IEEE*, 90(9):1565–1591, 2002.

[17] Forbes News. Apple Data Shows Shelter-In-Place Is Ending, Whether Governments Want It To Or Not, 2020. [Online; accessed 20-May-2020].

[18] Monia Ghobadi, Jamie Gaudette, Ratul Mahajan, Amar Phanishayee, Buddy Klinkers, and Daniel Kilper. Evaluation of Elastic Modulation Gains in Microsoft's Optical Backbone in North America. In *Optical Fiber Communication Conference*, page M2J.2. Optical Society of America, 2016.

[19] Monia Ghobadi and Ratul Mahajan. Optical Layer Failures in a Large Backbone. *IMC*, 2016.

[20] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Evolve or Die: High-Availability Design Principles Drawn from Google's Network Infrastructure. In *SIGCOMM*, 2016.

[21] Chi-Yao Hong, Subhasree Mandal, Mohammad Al-Fares, Min Zhu, Richard Alimi, Chandan Bhagat,

Sourabh Jain, Jay Kaimal, Shiyu Liang, Kirill Mendelev, et al. B4 and after: Managing Hierarchy, Partitioning, and Asymmetry for Availability and Scale in Google's Software-defined WAN. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 74–87, 2018.

[22] Mikel Jimenez and Henry Kwok. Building Express Backbone: Facebook's New Long-haul Network. *Facebook Engineering*, 2017. https://engineering.fb.com/2017/05/01/data-center-engineering/building-express-backbone-facebook-s-new-long-haul-network/.

[23] Ethan Katz-Bassett, Colin Scott, David R Choffnes, Ítalo Cunha, Vytautas Valancius, Nick Feamster, Harsha V Madhyastha, Thomas Anderson, and Arvind Krishnamurthy. Lifeguard: Practical repair of persistent route failures. *ACM SIGCOMM Computer Communication Review*, 42(4):395–406, 2012.

[24] Ella Koeze and Nathaniel Popper. The Virus Changed the Way We Internet, April 2020. https://www.nytimes.com/interactive/2020/04/07/technology/coronavirus-internet-use.html.

[25] Craig Labovitz, Abha Ahuja, and Farnam Jahanian. Experimental study of Internet Stability and Backbone Failures. In *Digest of Papers. Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (Cat. No. 99CB36352)*, pages 278–285. IEEE, 1999.

[26] Craig Labovitz, Roger Wattenhofer, Srinivasan Venkatachary, and Abha Ahuja. Resilience characteristics of the internet Backbone routing infrastructure. In *Proceedings of the Third Information Survivability Workshop, Boston, MA*, 2000.

[27] Hongqiang Harry Liu, Srikanth Kandula, Ratul Mahajan, Ming Zhang, and David Gelernter. Traffic engineering with forward fault correction. In *ACM SIGCOMM (2014)*, 2014.

[28] Andra Lutu, Diego Perino, Marcelo Bagnulo, Enrique Frias-Martinez, and Javad Khangosstar. A Characterization of the COVID-19 Pandemic Impact on a Mobile Network Operator Traffic. In *Proceedings of the 2020 ACM SIGCOMM conference on Internet measurement conference*, 2020.

[29] Athina Markopoulou, Gianluca Iannaccone, Supratik Bhattacharyya, Chen-Nee Chuah, and Christophe Diot. Characterization of failures in an IP Backbone. In *IEEE INFOCOM 2004*, volume 4, pages 2307–2317. IEEE, 2004.

[30] Athina Markopoulou, Gianluca Iannaccone, Supratik Bhattacharyya, Chen-Nee Chuah, Yashar Ganjali, and Christophe Diot. Characterization of failures in an operational ip backbone network. *IEEE/ACM transactions on networking*, 16(4):749–762, 2008.

[31] Debasis Mitra and Qiong Wang. Stochastic Traffic Engineering for Demand Uncertainty and Risk-aware Network Revenue Management. *IEEE/ACM Transactions on networking*, 13(2):221–233, 2005.

[32] NCTA. National upstream peak growth — observed increase in peak consumer usage — observed increase in peak consumer usage, 2020. [Online; accessed 20-May-2020].

[33] Network World. Why didn't COVID-19 break the internet?, 2020. [Online; accessed 20-May-2020].

[34] Bryan Ng, Matthew Hayes, and Winston KG Seah. Developing a traffic classification platform for enterprise networks with SDN: Experiences & lessons learned. In *2015 IFIP Networking Conference (IFIP Networking)*, pages 1–9. IEEE, 2015.

[35] Ramakrishna Padmanabhan, Aaron Schulman, Alberto Dainotti, Dave Levin, and Neil Spring. How to find correlated Internet failures. In *International Conference on Passive and Active Network Measurement*, pages 210–227. Springer, 2019.

[36] Junghun Park, Hsiao-Rong Tyan, and C-C Jay Kuo. Internet traffic classification for scalable QoS provision. In *2006 IEEE International conference on multimedia and expo*, pages 1221–1224. IEEE, 2006.

[37] Paul Poast. Changing the rules of International Relations, April 2020. https://news.uchicago.edu/videos/covid-2025-changing-rules-international-relations-paul-poast.

[38] Rahul Potharaju and Navendu Jain. When the network crumbles: An empirical study of cloud network failures and their impact on services. In *Proceedings of the 4th annual Symposium on Cloud Computing*, pages 1–17, 2013.

[39] Shahbaz Rezaei and Xin Liu. Deep learning for encrypted traffic classification: An overview. *IEEE communications magazine*, 57(5):76–81, 2019.

[40] Matthew Roughan, Subhabrata Sen, Oliver Spatscheck, and Nick Duffield. Class-of-service mapping for QoS: a statistical signature-based approach to IP traffic classification. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 135–148, 2004.

[41] SafeGraph. Footprint data. https://www.safegraph.com/covid-19-data-consortium.

[42] Jane M Simmons. Catastrophic failures in a Backbone Network. *IEEE communications letters*, 16(8):1328–1331, 2012.

[43] Rachee Singh, Manya Ghobadi, Klaus-Tycho Foerster, Mark Filer, and Phillipa Gill. RADWAN: Rate Adaptive Wide Area Network. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18.

[44] Lakshminarayanan Subramanian, Ion Stoica, Hari Balakrishnan, and Randy H Katz. OverQoS: offering Internet QoS using Overlays. *ACM SIGCOMM Computer Communication Review*, 33(1):11–16, 2003.

[45] Sean J Taylor and Benjamin Letham. Forecasting at scale. *The American Statistician*, 72(1):37–45, 2018.

[46] Bruno Vidalenc, Laurent Ciavaglia, Ludovic Noirie, and Eric Renault. Dynamic risk-aware routing for OSPF networks. In *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*, pages 226–234. IEEE, 2013.

[47] Anh Viet Vu, Jack Hughes, Ildiko Pete, Ben Collier, Yi Ting Chua, Ilia Shumailov, and Alice Hutchings. Turning Up the Dial: the Evolution of a Cybercrime Market Through Set-up, Stable, and Covid-19 Eras. In *Proceedings of the 2020 ACM SIGCOMM conference on Internet measurement conference*, 2020.

[48] Xipeng Xiao and Lionel M Ni. Internet QoS: A big picture. *IEEE network*, 13(2):8–18, 1999.

[49] Zhizhen Zhong, Nan Hua, Massimo Tornatore, Jialong Li, Yanhe Li, Xiaoping Zheng, and Biswanath Mukherjee. Provisioning short-term traffic fluctuations in elastic optical networks. *IEEE/ACM Transactions on Networking*, 27(4):1460–1473, 2019.

Staying Alive: Connection Path Reselection at the Edge

Raul Landa, Lorenzo Saino, Lennert Buytenhek and João Taveira Araújo
Fastly

Abstract

Internet path failure recovery relies on routing protocols, such as BGP. However, routing can take minutes to detect failures and reconverge; in some cases, like partial failures or severe performance degradation, it may never intervene. For large scale network outages, such as those caused by route leaks, bypassing the affected party completely may be the only effective solution.

This paper presents Connection Path Reselection (CPR), a novel system that operates on *edge networks* such as Content Delivery Networks and edge peering facilities [52, 64] and augments TCP to deliver transparent, scalable, multipath-aware end-to-end path failure recovery.

The key intuition behind it is that edge networks need not rely on BGP to learn of path impairments: they can infer the status of a path by monitoring transport-layer forward progress, and then reroute stalled flows onto healthy paths. Unlike routing protocols such as BGP, CPR operates at the timescale of round-trip times, providing connection recovery in *seconds* rather than minutes. By delegating routing responsibilities to the edge hosts themselves, CPR achieves *per-connection* re-routing protection for *all* destination prefixes without incurring additional costs reconstructing transport protocol state within the network. Unlike previous multipath-aware transport protocols, CPR is *unilaterally deployable* and has been running in production at a large edge network for over two years.

1 Introduction

Survivability is a core design tenet of the Internet, and a key factor in its enduring success. In reviewing the formative years of the DARPA Internet Protocols, Clark [19] listed survivability second in importance only to the top level goal of interconnecting existing networks:

> It was an assumption in this architecture that synchronization would never be lost unless there was no physical path over which any sort of communication could be achieved. In other words, 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 today falls short of this assumption. Failures are not only common on the Internet, they are often visible [25, 28, 29], sometimes spectacularly so [39, 41, 42, 53, 54]. While large outages are rare, transient reachability fluctuations, colloquially referred to as *internet weather*, are frequent [25]. Attempting to prevent all sources of outages is an exercise in futility: failures are endemic to every component at every layer along every path on the Internet, and subsets of components interact to form complex failure conditions which cannot be anticipated. Instead, the most cost-effective way of improving reliability on the Internet is to *circumvent failures when they occur*. Traditionally, this task has fallen upon network providers, who rely on routing protocols such as the Border Gateway Protocol (BGP) and Open Shortest Path First (OSPF) to route around failures. Routing alone, however, is not enough.

Firstly, since the successful delivery of keepalive messages does not imply the successful delivery of client traffic, routing protocols can only detect a subset of failure conditions. For example, a BGP session may be hashed onto a healthy Link Aggregation Group (LAG) member, while other links in the same group falter. Misconfigurations, such as those that result in route leaks, can impact large swathes of the Internet [26, 35, 41, 53]. Such events, undetected by routing protocols, often require manual intervention to mitigate at significant cost to the stakeholders involved.

Secondly, the time required by routing to re-establish a consistent state after a failure increases with the size of the network and can take minutes [37], during which loops and blackholes can occur [28, 29]. Even detection itself can be slow. The recommended value for the *BGP hold timer* (the time after which a non-responsive BGP peer is marked as failed) is 90s [33, 48], but many implementations set the default value as high as 180s [4, 5, 11, 18] or even 240s [23]. Even with sophisticated monitoring infrastructure [17, 32, 51], transient performance degradations have often disappeared by the time corrective updates can fully propagate.

Rather than relying on routing reconfiguration, protocols such as SCTP [55] and MPTCP [24] propose pushing the responsibility for mitigating path failures to the transport layer. Unfortunately, multipath transport protocols have struggled with adoption, and are today still circumscribed to niche use cases. Although multipath-aware protocols provide compelling reliability improvements, the current Internet architecture provides limited means (and incentives) for network providers to push multipath options out towards clients. With deployments limited to subsets of traffic or client population [16, 36, 46], their promise remains largely unfulfilled.

Edge networks [51, 52, 64] provide a natural vantage point from which to improve reliability, acting as a critical intermediary between application/content providers, hosted on highly centralized cloud infrastructure, and a globally distributed set of clients. By providing caching, security and compute functions as close to clients as possible, edge networks have positioned themselves to carry most of the customer facing traffic on the Internet [51, 52, 64]. Further, because they are expected to commit to SLAs guaranteeing the successful processing of end user requests, they end up bearing the costs of network layer failures and have a tremendous economic incentive to improve reliability. Conveniently, they also have the means. Unlike access/transit providers, edge networks have end-to-end visibility of traffic, and can therefore detect and react to failures faster and at finer granularity. By design, edge networks are multihomed and have access to better path diversity than end clients.

This paper presents Connection Path Reselection (CPR), a software-based approach improving the end-to-end reliability of edge network traffic. The key intuition behind it is that edge networks do not need to rely on BGP to learn of path degradation: they can infer the status of a path by monitoring transport-layer forward progress, and then reroute stalled flows onto healthy paths. This not only improves connection recovery, but also allows traffic to be shifted on a per-flow basis, greatly reducing the likelihood of load-induced cascading failures. Unlike previous proposals, CPR is unilaterally deployable, applicable to all flows, and simple to configure. Its implementation is entirely contained in a server-side kernel patch; it does not require programmable switches or any extra infrastructure. CPR has been in production for over two years at Fastly, a multi-Tbps edge cloud provider, where it successfully mitigates ~120 degradation events every day, each ~8 minutes long (over ~16 hours per day).

Having described our motivation, the remainder of this paper is organized as follows. First, we discuss the background of this work (§2), explain first how CPR detects path impairments (§3) and reroutes traffic as a result (§4). We then share results from production measurements (§5), followed by some operational considerations (§6). Finally, we compare CPR with related work (§7) and present our conclusions (§8).

2 Background and motivation

Edge networks (as understood in *e.g.,* [52, 64]) have unique characteristics that must be considered when designing a mechanism to improve end-to-end customer traffic reliability.

Edge networks support a diverse and changing set of applications. Early edge networks such as Content Delivery Networks (CDNs) were designed to support a narrow segment of Internet traffic: large, static content that could benefit from caching. This narrow traffic profile allowed for a wide set of potential optimizations. Edge networks have since evolved to support a much wider set of use cases (security, edge compute) and applications which no longer fit a neat traffic profile: a video client may need to retrieve small manifest files before requesting video chunks, or a browser session may download cached assets while maintaining a long-polling connection over which it receives update notifications. As such, edge networks today represent a microcosm of Internet traffic, not a segment. While it may be tempting to focus our efforts on improving reliability for a subset of traffic, performance degradation on any flow can adversely impact an entire application. A further complication is that it is not always a given that end clients will retry. For example, packaging a container image can involve retrieving potentially hundreds of individual assets. Failure to acquire any single one of these assets can result in the entire build process failing, at which point a user must decide whether to retry. This implies that we must detect *any* potential source of failure, for *every* type of flow, independently of its source, length, or capabilities of the end-client. We must also take into account that most traffic towards end-users will likely flow through middleboxes.

Edge networks are constrained by physical capacity. Points of Presence (POPs) are limited by physical space, and are designed to maximize the number of requests per second (RPS) they can serve [59]. Peak RPS is primarily dictated by storage and compute capacity - not bandwidth. Unlike traditional cloud environments, we cannot increase the physical footprint of network hardware, since that would necessarily reduce the amount of hardware dedicated to serving requests. Given our motivation for improving reliability is to reduce costs, we can not do so at the expense of efficiency.

Edge networks have unpredictable traffic patterns. Edge networks are subject to sudden fluctuations in demand due to flashcrowds or DDoS attacks. While physical capacity at any given POP is fixed, operators can shift traffic between POPs by adjusting DNS and BGP anycast configurations. This traffic engineering wreaks havoc on any potential solution that acts only on a set of heavy-hitter prefixes which is periodically updated. For instance, a POP located in Los Angeles may only observe traffic from prefixes in southern California in normal conditions, but this can change abruptly if *e.g.,* a POP in San Jose undergoes maintenance, or if a DDoS attack targets POPs in Japan. Traffic patterns shift dramatically during significant congestion and routing events, which is precisely when reliability suffers the most. Our design takes these constraints into account, and presents a system which strives to:

- detect and react to failures affecting *any flow, at any point in its lifetime*;
- minimize operational and infrastructure costs;
- interact safely with concurrent traffic engineering and routing processes.

Such a system is possible because edge networks support routing architectures which expose multipath capability to end servers [57,64]. By maintaining multiple routing tables in the kernel and allowing transport sockets and userspace applications to select which one to use on a per-packet basis, edge servers can override standard BGP route selection and instead implement objective-driven routing policy by themselves.

The benefits of path diversity have been amply studied (*e.g.,* [21,60]), in particular for stub networks [27]; even when performance gains are not forthcoming, cost benefits can be achieved [6]. Previous work on path-switching revealed that it is possible to improve average path loss performance by an order of magnitude on average by dynamically switching paths [56]. Previous work on CDN multihoming demonstrated 25% performance improvement for 3 out of 4 metro areas simply by selecting the best of two transit providers, with comparable reliability improvement [7]. One specific type of simple path diversity is highly prevalent: *path load balancing*. 72% of source-destination network pairs explored in [12, 13] show evidence of load balancing; for ~12% of these, load balanced paths are asymmetric and explicit selection can significantly improve end-to-end latency. A more recent study [47] showed even more significant benefits: not only do paths from large cloud providers show latency differences between load-balanced paths exceeding 20ms to 21% of public IPv4 addresses; 8 pairs of datacenters were found to have latency differences between load-balanced paths exceeding 40ms. Path diversity is high for edge networks: in the CPR deployment presented here each POP typically connects to a few transit providers and several peers[1].

This is the starting point for Connection Path Reselection (CPR). *Given the multipath capabilities of edge servers, how can we extend TCP to circumvent failures?* Focusing on TCP is appealing not only because its congestion control and loss recovery mechanisms are sufficient on their own for a large class of end-to-end impairments, but also because *any* automated, short-timescale re-routing of large volumes of traffic can override traffic engineering and create undesirable traffic distributions. By performing path selection decisions at the *connection* (rather than the *route/prefix*) level, CPR minimizes its impact on traffic volumes.

CPR is embedded within TCP and implemented as a server-side patch to the Linux kernel, and relies on extensions to the `tcp_sock` struct to keep essential re-route-related state. Because the kernel itself abstracts the IP address version at the socket level, CPR naturally covers both IPv4 and IPv6 traffic. Since CPR is parsimonious with both its execution and instrumentation state, it remains scalable even under extreme situations such as flashcrowds or DDoS attacks. The operation of CPR can be decomposed into two independent sub-problems: how to detect genuine path failures through *impairment detection*, and how to circumvent an impaired route through *path reselection*. We will tackle each of these problems in turn.

[1] See §5.3 and appendix A.1 for further details.

3 Impairment detection

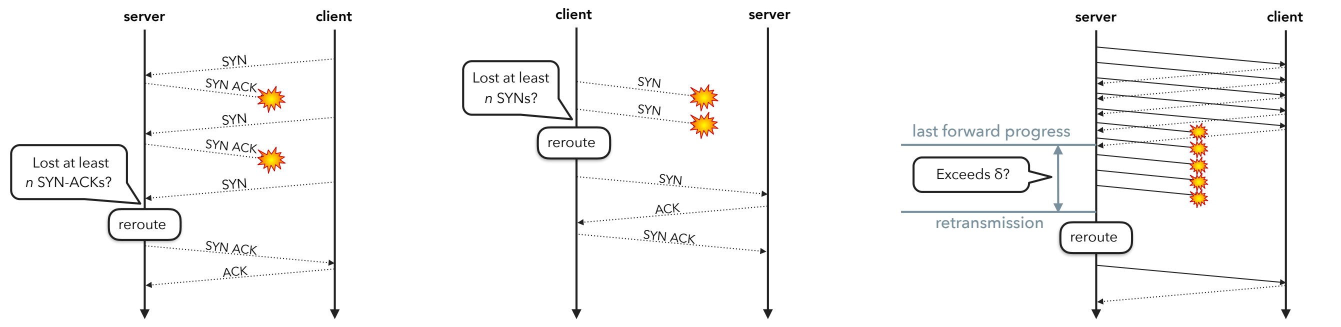
The main task of CPR's *impairment detection* algorithms is to accurately identify path failures based on transport layer performance in a timely manner. A key challenge is to distinguish between spurious packet loss, which should be recoverable with retransmission over the same path, and persistent failures, which are better addressed by selecting a different one. From a transport layer perspective, connections routed over an impaired path experience *stalls*, *i.e.,* fail to make forward progress for some amount of time in spite of retransmissions. CPR works by detecting stalls and using them as a signal to select an alternate path, with the objective of eventually resuming forward progress.

Two different impairment detection mechanisms are necessary, each addressing complementary stages in the TCP connection lifecycle. The first (§3.1) deals with path impairments before connection establishment; the second (§3.2) deals with impairments arising after connections have successfully established.

3.1 Pre-establishment impairments

A TCP connection is initiated by a client transmitting a SYN packet to commence a three-way handshake, to which the server will reply with a SYN-ACK. If the server's outbound path has failed before a connection is established, the SYN-ACK transmitted by the server will be lost. The client then retransmits the SYN after a predefined interval (1s on Linux [3]), to which the server will reply again with a SYN-ACK that will also be lost. This retry behavior continues until the path becomes available again or the client stops retrying (a Linux client retries 6 times by default). In such cases (fig. 1a) CPR will declare a stall upon exceeding the threshold of n presumed lost SYN-ACKs. Upon detecting a stall, route reselection is triggered on every subsequent SYN-ACK retransmission until the connection is successfully established or the client times out.

This detection mechanism is quite coarse because its ability to detect a failure is limited by the frequency of SYN retransmissions at the client. The precision and speed of failure detection could be improved if the server proactively retransmitted SYN-ACKs at a higher frequency without waiting for



(a) *Inbound* pre-establish (b) *Outbound* pre-establish (c) *Established*

Figure 1: Impairment detection and reroute

SYN retransmissions from the client. This would however create an amplification vector that could be exploited by SYN flood attacks and therefore would not be safe to deploy in untrusted environments.

CPR's pre-establishment impairment detection algorithm cannot be used when SYN cookies are enabled, because then the kernel will not keep any state for incoming pre-establish connections. This is a desirable feature. By default, the Linux kernel sends SYN cookies upon listen queue overflow, which is typically triggered by SYN floods. This will result in SYN-ACK stall detection being disabled during an attack, and SYN-ACKs using the preferred path.

Finally, we note that the same mechanism could be applied to connections initiated by edge servers (see fig. 1b), whereby we reroute after a given number of lost SYN packets rather than SYN-ACK packets. This case is less relevant in practice however, since most edge server traffic results from inbound connection requests.

3.2 Post-establishment impairments

For established connections (see fig. 1c) CPR verifies, before a retransmission, whether the connection has failed to make forward progress for a time threshold δ. If so, it declares a stall and selects a new egress path. CPR marks a connection as making forward progress whenever an (S)ACK is received for data that has been sent, but not yet acknowledged. This requires storing a timestamp variable in each TCP socket to keep track of forward progress. The algorithm operates as follows:

- clear the timestamp as long as there are no outbound segments in flight, and set it to the current time when the segment that is at the front of the transmit queue is transmitted *for the first time*;
- update the timestamp to the current time whenever the connection makes forward progress, *i.e.,* receives an (S)ACK that acknowledges data byte ranges previously transmitted but not yet acknowledged;
- clear the timestamp when the last outstanding byte range has been fully acknowledged;
- declare a stall when (re)transmitting a TCP segment if 1) the timestamp is set, and 2) the time elapsed since the timestamp exceeds a threshold δ.

Because this algorithm declares stalls *on retransmission*, connections that become idle whilst using paths that subsequently fail cannot declare a stall until their first retransmission. This behavior minimizes spurious path reselection.

Algorithm parameters. As with n, setting an appropriate value of δ needs to strike an appropriate tradeoff between reactivity and accuracy. The key challenge is to ensure that the threshold works consistently well across connections, regardless of their RTT. Setting δ to a fixed, global value would lead to either spurious stalls for high RTT connections, or sluggish response for low RTT connections. This can be addressed by defining δ in terms of path properties already estimated by TCP. We could, for instance, define δ as a multiple μ_{rto} of the connection retransmission timeout (RTO) T_{rto}, so that $\delta = \mu_{\text{rto}} T_{\text{rto}}$. As usual, $T_{\text{rto}} = \texttt{srtt} + 4 \times \texttt{rttvar}$, where `srtt` is the smoothed round-trip time and `rttvar` is the round-trip time variance [50]. Unfortunately, this solution on its own could be problematic for connections with very low `srtt` and `rttvar`, because, given a low value of δ, temporary router queue build-ups and subsequent increased latency may be misidentified as stalls and trigger spurious reroutes. We guard against this issue by defining δ_{min}, a lower bound for δ, and setting $\delta = \max(\delta_{\text{min}}, \mu_{\text{rto}} T_{\text{rto}})$.

Using a small value for μ_{rto} (*e.g.,* $\mu_{\text{rto}} = 1$) may spuriously trigger reroutes during the slow start phase of the connection. For paths with moderate background packet loss, RTO expiration is more likely to happen when there are few TCP segments in flight, *e.g.,* during slow start: once the connection has filled its *bandwidth-delay product*, a constant stream of incoming ACKs makes the triggering of RTOs less likely.

Rate limiting reroutes. Re-routing onto a new path does not necessarily result in recovery: the new egress path could share a failure with the original one, or the impairment responsible for the stalling could be on the inbound path. When this occurs, CPR will simply continue to probe paths until forward progress is made. If the reroute threshold δ of the connection is low, the connection may end up being rerouted multiple times in rapid succession. This could result in CPR not being able to gather enough data about the state of a new path to make an accurate decision about its suitability before trying yet another path. We addressed this by implementing a *rate limiting* over periods of length `wait`, so that connections will not be re-routed more than once in every period.

Triggering stall detection logic. Since CPR performs stall detection on retransmission with a timeout expressed as a multiple of the RTO, it is natural to ask whether stall detection logic could be co-located with the RTO triggering logic of the TCP state machine. The answer is negative, because not all retransmissions relevant to connection forward progress are triggered by RTO expiration. Consider the case where the outbound path used by a connection has failed and both the local and the remote ends have outstanding (unacknowledged) data. It is possible for retransmissions by the local end to keep being triggered by the reception of retransmissions from the remote end, before a local RTO can elapse. For this reason, stall detection logic *reuses* the RTO value, but is evaluated *independently* of RTO logic at relevant points in the TCP state machine (*e.g.,* when sending retransmissions).

4 Path reselection

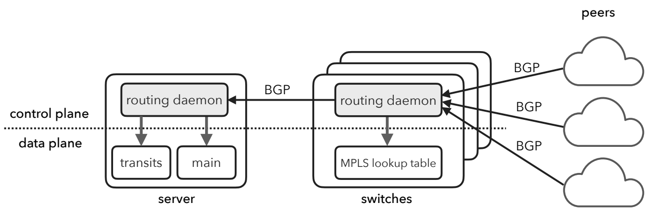


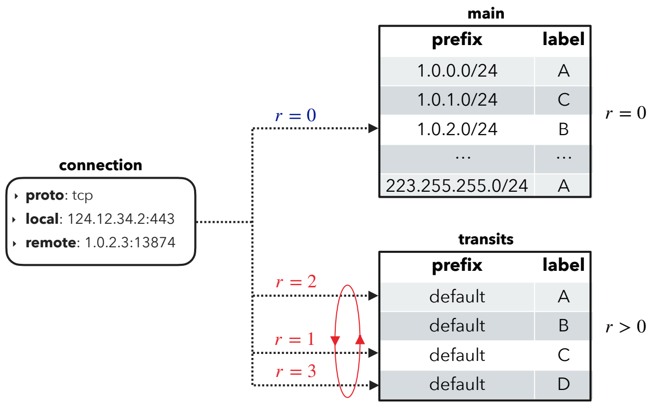
Figure 2: Routing architecture

CPR leverages a routing architecture similar to Espresso [64] and Silverton [14, 57] which push visibility of all available routes down to edge hosts. In this architecture (depicted in fig. 2) each host is connected to a number of switches, which are in turn connected to a number of upstream providers. These include both transit providers and settlement-free peers, connected directly through Private Network Interconnects (PNI) or Internet Exchange Points (IXP).

Each switch performs two tasks. First, an MPLS label is configured for each upstream provider, and a corresponding nexthop entry is inserted into the local routing table. Second, BGP route updates received from upstream providers are tagged with the associated MPLS label, and forwarded to routing daemons on the host. The routing daemon on end hosts populates two routing tables:

- The **main** table contains all policy-preferred routes among those learned from all peers, and is used for routing traffic under *normal* circumstances *i.e.,* when path reselection has not been requested. It contains routes preferred under performance, capacity and cost constraints.
- The **transits** table is populated with all *default routes* (*i.e.,* 0.0.0.0/0 or ::/0) learned from upstream providers. Since settlement-free peers do not provide universal reachability (*i.e.,* export a full routing table), only default routes provided by transit providers are included.

Given the routing architecture in fig. 2, and having access to the *main* and *transits* table, the next objective of CPR is to provide a mechanism to allow the stall detection algorithms in §3 to select a new path for a stalled connection. We achieve this by associating a reroute counter r with each connection, and incrementing it every time a stall is declared for that connection. This counter is stored in the 4 most significant bits of the firewall mark (*fwmark*) of a connection, a 32 bit value that can be used to tag packets traversing the Linux network stack and make routing decisions about them. To make rerouting based on r possible, we made two relevant changes. First, we changed the Equal Cost Multipath (ECMP) hashing function used by the Linux kernel. The standard Linux implementation of ECMP selects a nexthop by hashing the connection *5-tuple* (*i.e.,* source and destination IP addresses, source and destination ports and protocol number). *CPR includes the value of the reroute counter r into the hash computation*. Second, we configure an *ip rule* to ensure that, for any connection for which $r > 0$, a next hop is looked up from the *transits* table rather than from the *main* table. Hence, *we use r as a flag that triggers CPR-specific routing for connections that have suffered stalls*. The combined effect of these two changes is that simply incrementing the reroute counter will force a new route lookup.



main

prefix	label
1.0.0.0/24	A
1.0.1.0/24	C
1.0.2.0/24	B
···	···
223.255.255.0/24	A

transits

prefix	label
default	A
default	B
default	C
default	D

Figure 3: Path reselection upon stall detection

As an illustration, consider a hypothetical connection that has not yet experienced a stall, as shown in fig. 3. At this point in its lifetime, $r = 0$, route lookups are performed using the *main* table, and the BGP-defined egress path is used. When the stall detection algorithm (§3.2) declares a stall, it increments r. From that point on, since $r > 0$, route lookups are performed using the *transits* table, and the next hop for IP packets forming this connection is pseudorandomly selected among all the nexthops of the default route present in the transits table according to ECMP(5-tuple, r). Since each increment of r forces a new route lookup, as r increases the stalled connection will follow a unique, pseudorandom sequence of egress paths which will depend on both its 5-tuple and the ECMP hash used. The aggregate effect of this process is that rerouted connections are homogeneously "load balanced" among all available egress paths.

We now show that CPR should be able to resolve recoverable stalls with relatively few retries. If n_p is the number of egress paths available, of which n_s lead to *stalled* connections, each path reselection is a Bernoulli trial with a success probability $p = \frac{n_p - n_s}{n_p}$. The number of re-routes k required until recovery will be geometrically distributed [2] so that $P(X \leq k) = 1 - (1-p)^k$. Hence, the expected maximum number of re-routes that will be required to find a good path with a given probability β is $k^* = \frac{\log(1-\beta)}{\log(1-p)}$. Even a conservative[2] $p = 50\%$ and $\beta = 95\%$ results in only $k^* \approx 4$ re-routes.

We note that, although fig. 2 states that the *main* table should contain a full routing table, CPR does not *require* this to be the case. As noted above, path reselection only relies on the *transits* table. Further, although our implementation uses MPLS to steer specific flows towards a given provider, this can also be done using GRE tunneling [64] or DSCP marking [52]. Routes can be pushed down to the host using BGP add-path [62] or proprietary mechanisms. Our architecture is just one of many that could support CPR deployment; the basic primitives of CPR are applicable to many scenarios.

5 Evaluation

This section evaluates the performance of CPR in a large edge cloud production deployment with daily traffic peaks on the order of tens of Tbps. All results were collected through passive measurements of production traffic.

5.1 Parameter tuning

Tuning CPR involves resolving a tradeoff: whereas unnecessarily rerouting connections could place them on paths with potentially lower performance and higher cost, failing to react to a recoverable path impairment increases its potential to harm client connections. This section discusses how we tuned the CPR parameters (§3) to resolve this tradeoff between accuracy and reactivity.

Tuning pre-establishment impairment detection. The only parameter involved in detecting impairments prior to connection establishment is n, the number of presumed lost SYN-ACKs after which a reroute is executed (§3.1). Hence, at this stage tuning involves determining the value of n after which timely connection establishment becomes unlikely without CPR intervention.

We proceeded by instrumenting servers in three distinct geographical regions (North America, Europe and Asia) with CPR disabled. We then measured how many SYN retransmissions occurred for all connections that were eventually

[2]See §5.3 and appendix A.1 for further details.

n	APAC	EU	NA
0	.63	.64	.70
1	.13	.17	.28
2	.05	.09	.12
3	.03	.06	.08
4	.01	.05	.05
5	.01	.03	.03

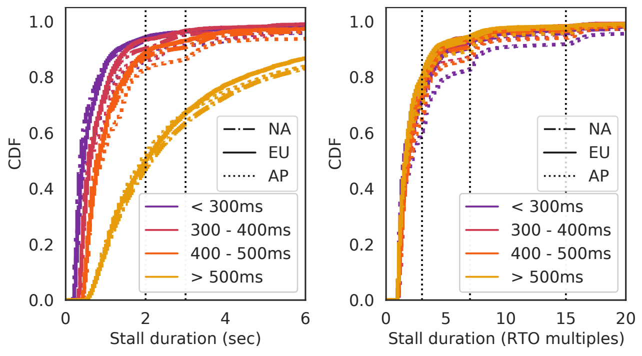
Table 1: Proportion of connections not establishing after n consecutive SYNACK losses (%)

Stall duration lower bound	APAC	EU	NA
RTO	1.79	.57	2.18
2s	.34	.36	.10
3s	.23	.24	.06

Table 2: Proportion of connections experiencing at least one stall during their lifetime, as a function of the stall duration (%)

established[3]. As shown in table 1, only ~0.63% to ~0.7% of connections experience impairments before establishing, depending on the region. This means that more than ~99.3% of connections establish without any retransmissions. Further retransmissions help connection establishment, but with noticeable diminishing returns. For example, after two consecutive retransmissions without a reroute, the probability of a connection being successfully established was between ~0.05% to ~0.12%. Based on these findings, we set $n = 2$ in our production configuration.

Tuning post-establishment impairment detection. We addressed the tuning of δ_{min} and μ_{rto} (§3.2) in two steps. First, we followed a similar measurement methodology as that used to tune n, this time focusing on connections experiencing at least one RTO expiration during their lifetime. Our results, reported in table 2, provide evidence of significant regional variability. Whereas ~0.1% of connections in North America experience stalls lasting for 2 seconds or more, this increases to ~0.35% for connections in Europe or Asia-Pacific[4]. However, these results also show that the probability that a connection recovers after experiencing a stall for 2 or 3 seconds was very low irrespective of geographical region.



(a) (b)

Figure 4: Duration of TCP stalls by region and RTO

[3]This was done during a period free of obvious biases such as outages, high-load client events, DDoS attacks, etc.

[4]These values constitute a lower bound; during the active phase of an impairment event the proportion of affected connections can rise by an order of magnitude or more (see §5.2).

The next step was to investigate the time required for a connection to resume forward progress after an RTO (fig. 4). We begin by noting the two vertical lines in fig. 4a: these correspond to the two candidates found in the previous study. For connections with RTO $\leq$ 500ms, most of the probability mass in the dataset lies to the left of the first line, providing evidence that for these connections the length of a stall is largely independent of RTO and $\delta_{\text{min}} = 2\text{s}$ is sufficient to ensure adequate impairment detection.

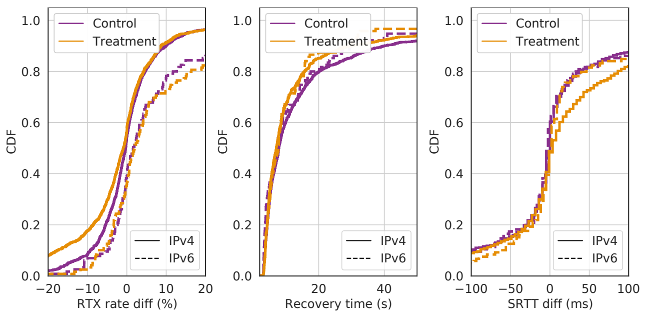
To understand how best to handle connections with RTO > 500ms, we first note the probability mass "bumps" in fig. 4b. These intervals of 3, 7 and 15 RTOs stem from TCP retransmission behavior under exponential backoff. Since the majority of the probability mass lies to the left of the first line, a large proportion of connections will recover on their own before $\mu_{\text{rto}} = 3$, irrespective of their RTO. This addresses the high RTO connections that were not already covered by $\delta_{\text{min}} = 2\text{s}$, and provides a rationale to set $\mu_{\text{rto}} = 3$.

Finally, we set `wait` (§3.2) by observing day-to-day operation of the system in production. We select `wait` $= 1\text{s}$ sec on the basis of keeping the volume of steady-state rerouted traffic to a sufficiently low level.

5.2 Evaluating benefit and non-harm

Methodology. To ascertain the impact that CPR has on ongoing connections we follow an *experimental* approach, in which we (pseudo)randomly label some connections as part of a *treatment* group, and the remainder as part of a *control* group for which path reselection logic is disabled. While the state and output of the impairment detection algorithms are maintained for both groups, network-layer path changes are triggered only for treatment connections. This setup allows us to explore the degree to which the benefits of CPR during path outages outweigh its potential costs when no impairments are present. We begin by exploring whether rerouting a stalled connection could make its performance significantly worse than doing nothing. To the extent that the answer to this question is negative, CPR will be innocuous when triggering due to stalls not associated with path impairments, and hence, non-recoverable by rerouting. We then move on to analyze the benefits that CPR provides during path impairment episodes.

Evaluating non-harm in the steady state. For both treatment and control we quantify the *reroute effect* on connection properties such as RTT or retransmission rate. Since connection properties need time to settle to their new values after a reroute in order to be meaningful, we focus on long-lived connections. We define the *onset* of a stall as the TCP sending event immediately prior to an RTO, and the *full resolution* as the first sending event after forward progress is restored and 30[5] segments have been sent. We define the reroute effect on a connection property as the difference between its value at



(a) (b) (c)

Figure 5: Steady-state reroute effect experiment results

onset and at *full resolution*.

The CDF of the reroute effect on the TCP retransmission rate[6] is presented in fig. 5a. Whereas it seems to be negligible for IPv6, for IPv4 the treatment connections have a greater proportion of their probability mass on the negative effect sizes, implying that rerouted IPv4 connections tend to have lower retransmission rates after the reroute than before it. This points towards CPR having *some* small benefit during the measurement period. Otherwise, the curves are very close to one another, providing evidence that CPR is not introducing significant retransmissions during steady state.

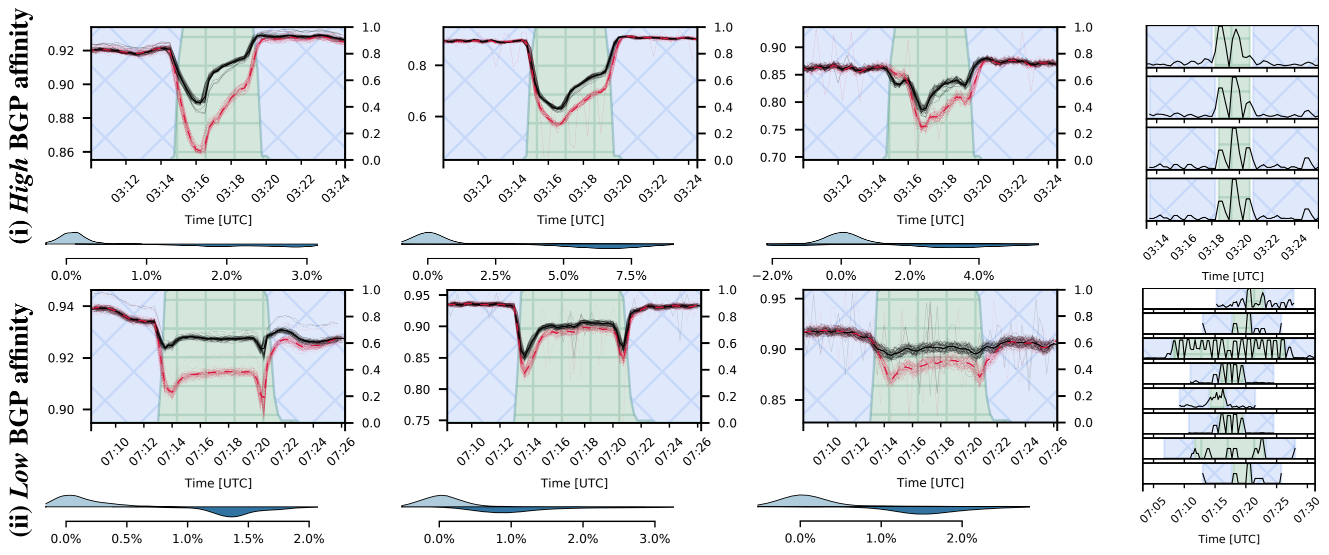
We can also use the results of the previously described reroute effects experiment to understand the effect of CPR on stall recovery speed. From fig. 5b we can see that treatment connections tend to have shorter resolution times compared to control connections, both for IPv4 and IPv6. The effect during steady state is small, as expected: ~80% of control connections recover within 20s of a reroute, compared to ~85% of treatment connections.

Finally, fig. 5c shows the effect of reroutes on `srtt` (§3.2). Again, a seemingly negligible effect for IPv6 is accompanied by a clear effect for IPv4, this time demonstrating higher `srtt` values after the reroute (usually by less than ~30 ms, but sometimes more than ~100 ms). This performance penalty is expected since our BGP traffic engineering policy optimizes for latency, and is more finely tuned in IPv4 than in IPv6 due to operational maturity. Since CPR explicitly reroutes away from paths selected by this policy, we are more likely to experience an increase in RTT than not. In this light, our configuration of CPR is an expression of the extent we are willing to subvert local routing policy in an attempt to recover from failure. Since this is a matter of policy, we observe there is no single correct configuration, but a range of potentially acceptable outcomes.

Evaluating benefit during stall events. During path impairments, the most common outcome for control connections is *failure* (rather than *e.g.,* increased RTT or retransmission

[5]This number was arbitrarily selected to provide enough samples for TCP connection properties to stabilize.

[6]Every connection in this dataset experienced a stall during the measurement period. Hence, retransmission rates are expected to be higher than the blended averages typically reported.



(a) $\pi_\bullet$(HEALTHY) (b) $p^o_\bullet$(HEALTHY|SYNRCVD) (c) $p^i_\bullet$(CLOSE|HEALTHY) (d) BGP events

Figure 6: CPR operation during two stall events (i and ii). The timeseries plots in figs. 6a to 6c show the *control* • (dashed) and *treatment* • (solid) group values (**left y-axis**). Each one of the *thin* lines corresponds to a single host; *thick* lines track the average for all hosts in a POP. In the same plot we also show the proportion of hosts in a POP for which the timestamp has been classified as part of an *active event* ⊞ or its *context* ⊠ (**right y-axis**). Beneath each timeseries plot we show a violin plot for the PDF of $\Delta\pi$ (fig. 6a) or $\Delta p^\bullet$ (figs. 6b and 6c), the difference between treatment and control values, for both *context* • (top) and *event* • (bottom) periods. Each subplot in fig. 6d shows BGP update/withdrawal rates for peering sessions which triggered anomaly detection in proximity to the active event span. Since only relative changes are relevant, y-axis tickmarks have been removed.

rates). Hence, to accurately assess the benefits of CPR we must look beyond the reroute effect measures presented above. Broadly, we resort to anomaly detection on the performance differences between treatment and control groups in order to identify *stall events*. The benefit of CPR for a given stall event can then be measured by comparing treatment and control performance differences during the event, with the corresponding performance differences during the immediately adjacent timespans. We refer to the union of timespans immediately preceding and following a stall event as its *context*.

In order to identify stall events, we instrumented the kernel to export additional metrics. First, we store per-connection information on the operation of the recovery mechanism as part of the socket metadata; this includes the reroute counter r, last forward progress timestamp, etc. Second, we maintain aggregate counters in the kernel which track state transitions as each connection traverses both TCP and CPR state machines. Each edge server aggregates separate counters for treatment and control group connections, allowing us to estimate the efficacy of CPR on each host according to multiple performance measures[7], including:

- $\pi_\bullet$(HEALTHY), the proportion of TCP connections in the HEALTHY state;
- $p^o_\bullet$(HEALTHY|SYNRCVD), the proportion of TCP connections that transition out from the SYNRCVD state towards the HEALTHY state, per unit time. This gives an indication of the instantaneous probability of connections connecting successfully; and
- $p^i_\bullet$(CLOSE|HEALTHY), the proportion of TCP connections that transition into the CLOSE state from the HEALTHY state, per unit time. This gives an indication of the instantaneous probability of connections closing while healthy (rather than stalled).

We present two CPR stall events in fig. 6, embedded in their *context* so that their impact is clearly visible. Each event is presented along its *BGP affinity score*, a synthetic measure of the degree to which observed routing plane events are correlated with transport layer anomalies (appendix C). Our examples were selected to juxtapose the case where there is a *high* BGP affinity score (i), and therefore an association between CPR and BGP behavior, and a *low* BGP affinity score (ii). In both cases we can observe that immediately after the start of the stall event (at the left context/event boundary) CPR treatment connections start experiencing better performance, as inferred from a higher $\pi_\bullet$(HEALTHY) (fig. 6a). Treatment connections also have better chances of establishing (fig. 6b), evidencing connection setup distress for control connections, and exhibit a greater chance of closing while HEALTHY, rather than when STALLED (fig. 6c). When taken together, these facts point towards a small but significant proportion of affected connections, both *established* and *pre-establish*, clearly benefiting from CPR, irrespective of the BGP affinity of the event. When a clear association with BGP is present, such as with fig. 6d (i),

[7]A more detailed overview of stall event detection is included in appendices B and C.

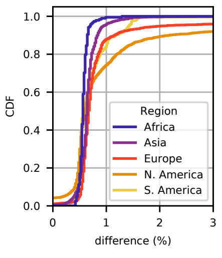
the benefits provided by CPR are materialized significantly before the associated BGP event is resolved. Conversely, fig. 6d (ii) demonstrates that CPR can recover just as effectively in cases for which BGP provides no remediation.

Each violin plot beneath a CPR time series event subplot in figs. 6a to 6c shows the benefit that CPR provided to connections assigned to the treatment group during the event. For state occupancies we rely on $\Delta\pi = \pi_t - \pi_c$, the difference between treatment and control values (similarly, for state transitions $\Delta p^{\bullet} = p_t^{\bullet} - p_c^{\bullet}$). For instance, in fig. 6a we can see that although the distribution of $\Delta\pi(\texttt{HEALTHY})$ was centered at zero outside the event period, it moved to the right during the active phase of the event, irrespective of BGP affinity. For (i) we see that CPR allowed an additional ~3% of connections (over the entire POP and to all destinations) to remain in the HEALTHY state compared with the control group; for (ii) this is reduced to ~1.4%. On the other hand, from $\Delta p^{o}(\texttt{HEALTHY}|\texttt{SYNRCVD})$ (fig. 6b) we see that during (i) CPR allowed an additional ~7% of connections in the POP moving out from a SYNRCVD state to enter the HEALTHY state; during (ii) this is reduced to ~1%.

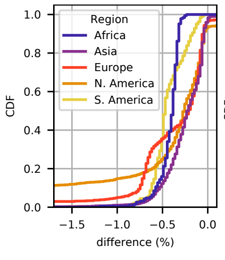
Evaluating benefit globally. Having explained the typical characteristics of stall events using individual examples, we now focus on 1) measuring CPR effectiveness at a global scale, and b) identifying the circumstances under which CPR is most effective. To this end we perform statistical aggregation on data collected from ~80 POPs over ~12 months. As before, we focus on *recovered events*, defined as stall events which show evidence of improvement in performance measures (*e.g.,* those presented in fig. 6). We do this for two reasons. First, whereas improved treatment group performance unequivocally points towards TCP stalls that can be resolved by path reselection, anomalies where CPR provides no benefit are uninformative: there are many possible causes of TCP stalls unrelated to resolvable path impairments (*e.g.,* wireless access roaming or dis-association) for which re-routing cannot be expected to help. The second reason is implementation-related. Since performance data such as presented in fig. 6d is only kept aggregated at host level (rather than at prefix or connection granularity) it is difficult to directly associate a CPR event with a set of destination prefixes/ASes, and hence, to a BGP-based ground truth. For recovered events this is not an issue: CPR itself provides direct confirmatory evidence of path impairment mitigation[8].

We will denote the daily per-POP average CPR state occupancy during the active phase of stall events (rather that their context) as $\mathbb{E}[\Delta\pi]$. This measure can be used to directly understand which geographies benefit the most from the deployment of CPR, as shown in figs. 7a and 7b. First, in fig. 7a we can see that most of the probability mass for $\mathbb{E}[\Delta\pi(\texttt{HEALTHY})]$ lies on the *positive* semi-axis regardless of geography, imply-

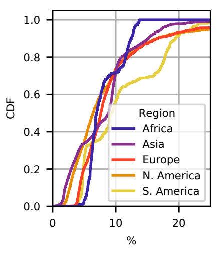
[8]Note however that, since our CPR deployment could *in principle* fail to recover from some stall events, our reported prevalence findings suffer from survivorship bias [63] and must be interpreted as lower bounds.



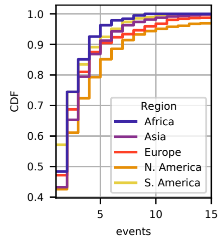
(a) $\mathbb{E}[\Delta\pi(\texttt{HEALTHY})]$



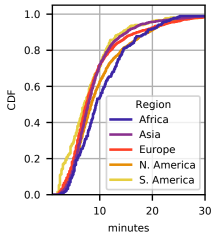
(b) $\mathbb{E}[\Delta\pi(\texttt{SYNRCVD})]$



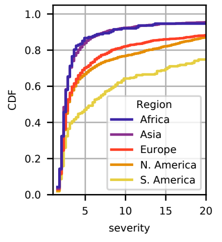
(c) $\mathbb{E}[\pi_t(\texttt{STALLED})]$



(d) Count



(e) Duration



(f) Severity Index

Figure 7: Per-day, per-POP recovered events

ing that $\pi_t \geq \pi_c$ and hence that the proportion of HEALTHY connections was higher for the treatment than the control group (median $\in [0.57\%, 0.68\%]$). Conversely, fig. 7b shows that most of the probability mass for $\mathbb{E}[\Delta\pi(\texttt{SYNRCVD})]$ lies on the *negative* semi-axis (median $\in [-0.46\%, -0.19\%]$), so that $\pi_t \leq \pi_c$. This means that CPR allows connections to both 1) connect faster and 2) spend less time stalled. We note that whereas $\pi_t(\texttt{HEALTHY})$ benefits are greatest in Europe (median = 0.68%) and Asia (median = 0.65%), $\pi_t(\texttt{SYNRCVD})$ benefit is particularly strong in South America (median = −0.46%) and Africa (median = −0.38%). However, as shown in fig. 7c, these benefits are accompanied by a cost: every day more than ~10% of connections on up to ~30% of PoPs experience stall events. Although the immense majority of these stalls are associated with abandoned connections and hence carry negligible traffic, we note that non-recoverable stalls induce a sustained level of background reroutes.

In contrast to figs. 7a to 7c, figs. 7d and 7e show weaker region dependence. Although fig. 7d provides some evidence that recovered stall events occur roughly proportionally to both 1) the aggregate traffic volume and 2) the interconnection density of a region, the difference between regions is small. Similarly, fig. 7e shows that most recovered events last under 20 minutes, irrespective of region (median $\in [7.1, 9.4]$). However, we note that this distribution can be heavy tailed: we have observed uncommon, longer events that span multiple hours. These can have "macroscopic" effects, triggering operational responses and re-routing over 20% of PoP traffic.

We finally turn our attention to the severity of recovered events. We define a heuristic *severity index* based on the normalized performance difference between treatment and control connections during the event and its associated context. The higher the number of standard deviations between treatment and control variables during the event itself, compared to before and after it, the higher the severity for the event (see appendix C). Perhaps unsurprisingly, fig. 7f shows that CPR routinely recovers from higher severity events in densely connected regions. However, the distinction between the North and South America curves in figs. 7d and 7f is instructive. Whereas in fig. 7d the former is always beneath the latter, in fig. 7f the opposite occurs. This shows that CPR helps recover from more events in regions with nontrivial BGP routing where complex business agreements and undersea cable topology can make BGP-only recovery more challenging.

As a complement to the per-POP statistics presented in fig. 7, we report that globally CPR mitigates on average ~120 impairment events every day with a median duration of ~8 minutes. Informally, this amounts to over ~16 hours per day. While this improvement may appear quantitatively modest, in practice these performance degradations bear a high cost in the absence of mitigation. Many of them exhibit no proximal cause, making them notoriously hard to debug. Even when an underlying cause is understood, many performance problems are short-lived and will be resolved by the time engineering or customer support resources have been mobilized to address them. In opportunistically mitigating either type of deterioration, CPR has an outsized impact in improving the reliability of services provided by edge networks, at virtually no cost.

5.3 Path coverage

Given that CPR deflects traffic among upstream providers when mitigating path impairments, it is informative to estimate the degree to which reroutes are likely to result in disjoint data plane paths. To this end, we analyze all routes learned from every provider in a given POP. Since all transits offer full routing tables, every destination prefix will have a set of available routes. In addition to these routes, we may have additional peering routes learnt from PNIs or IXPs. One of these paths (learnt from either peering or transits) will be *selected* as an outcome of BGP policy; the remaining routes (learnt over transits) represent valid *alternate* paths.

We proceed by defining each AS in the selected path to be *node-protected* if there is an alternate path for the same destination that does not include it. Then, we define *AS-node diversity* as the proportion of node-protected ASes in the selected path. Similarly, we can define an AS pair in the selected path to be *link-protected* if there is an alternate path that does not include the same AS pair in the same order, which leads to the entirely analogous definition of *Inter-AS-path diversity* as the proportion of link-protected AS pairs in the selected path. Since CPR will eventually explore every available path to every destination, these two measures can be used as proxies for the probability of recovery given a failure event involving any given destination prefix[9].

[9]Two practical qualifications are in order. First, since a packet can traverse different router-level paths within the same AS between given ingress and egress pairs, and multiple peering points between two adjacent ASes in an AS path, both quantities are lower bound estimates of actual dataplane path diversity. Second, under these definitions a number of high-volume destination prefixes advertised by directly connected peers will be counted as presenting zero diversity whilst in reality presenting negligible impairment risk as their path reliability is very high. These have been removed from the analysis in §5.3, but are reported in their entirety in appendix A.1.

An analysis of ~900k globally distributed routes revealed traffic-volume-weighted AS-node diversities of 90.8% and 86% for IPv4 and IPv6 respectively; these rise to 93.7% and 90.5% when considering Inter-AS-path diversity. The significant path diversity available to CPR evidenced by these numbers is a consequence of a strict requirement for edge networks: POPs must be interconnected to multiple transits in order to survive outages of individual providers or local exchange points. Since most client traffic is preferentially exchanged over peering, the addition of a transit provider has a limited impact in improving latency. Instead, multihoming is primarily a form of insurance. In this light, CPR is of great interest to edge network operators in that it extracts greater value from what is otherwise a necessary cost.

6 Operational considerations

CPR has been in use for over two years. This section revisits some of our original assumptions in the light of our accrued experience, as well as highlight some of the tweaks that were required along the way.

Per-route overrides. While it is beneficial to enable CPR by default on edge hosts, and the parameters derived in §5.1 are adequate for a wide range of conditions, there are cases where it is desirable to override configuration on a per-route basis. For example, rerouting intra-POP traffic onto transits is ineffective as a mitigation strategy, and costly due to the large volume of traffic exchanged between hosts within a POP.

Linux already allows per-route configuration of *features*, such as ECN or SACK. We expanded this method to allow the configuration of CPR on a per-route basis, and extended our edge host routing daemon to be able to inject routes into the *main* routing table accordingly. Over time, this has allowed us to selectively disable or experiment with CPR on a per-route basis as an integral part of our BGP policies. This evolution was gradual: what started exclusively as a server-side, transport layer extension has been progressively incorporated operationally as an extension of our routing infrastructure.

Middleboxes. Rerouting ongoing connections could in theory adversely interact with stateful middleboxes, which would be presented with packet streams for which the TCP three-way handshake happened in a different path (and hence was not observed). The expected outcome of this potential problem, increased TCP RST rates or silently dropped packets for treatment connections compared with control (see §5.2), has not been observed in practice after multiple years of sustained, worldwide, multi-Tbps usage. This is not unexpected, since

CPR does not modify the TCP header in any way. From the perspective of intermediate devices, CPR reroutes are indistinguishable from other reroutes events, such as those triggered by BGP or ECMP rehashes.

Prefix aggregation. It is natural to consider extending CPR by aggregating per-connection information and using it to influence routing. If many connections in a given prefix stall, only to recover successfully after a reroute, we could avoid future stalls by overriding the route for that prefix. Evaluating this approach is outside the scope of this paper, but the implementation is far from straightforward once we consider its risks. Firstly, as evidenced by §§5.1 and 5.2, not all transport-layer impairments will be correctable by path reselection. This means that there is always the danger of overriding routing and traffic engineering policy in an attempt to resolve a suspected problem that cannot be fixed that way. This risk is severely amplified by performing routing interventions on entire prefixes. A related threat is that, in addition to natural variability, stall signals are also susceptible to adversarial manipulation. Using stall recovery information to route entire prefixes would amplify the impact of such an attack, potentially burdening the edge network with higher transit costs or end-users with higher latency. Finally, there is no *a priori* reason to assume that the same routing intervention will have the same effect on every subprefix covered by an Internet-advertised prefix. Whereas this tradeoff is irreconcilable at the prefix level, it is trivially accommodated at the connection level.

Avoiding peering offload. As noted in §4, whereas routes in the *main* table are learned from *both* settlement-free peers and transits, routes in the *transits* table are *only* learned from transits. Therefore, CPR path reselection can re-route connections away from PNIs and IXPs and onto transits, which would typically result in greater costs and potentially less direct paths to destinations. During our design phase we accepted this limitation because the alternative is for connections to remain stalled; an opportunity to recover then makes the cost benefit positive. Although the nature of colocation facilities makes it relatively rare for a single POP to be connected to multiple IXPs, it would be beneficial to be able to failover from PNIs to IXPs. In our current architecture, this capability can only be provided at the cost greater routing complexity (defining the set of failover routes on a per-destination basis).

Path reselection. While one could envision more complex path selection algorithms than those presented in §3, in our experience the additional complexity does not translate to significant benefits. The simplicity of CPR is a core feature, providing quick recovery and effective load balancing without burdening operators with configuration overhead.

Userspace support. CPR need not be limited to the kernel, it can be leveraged or selectively disabled by any userspace application. To support this, we reserved bits in *fwmark*, which can be set via a `setsockopt` system call, to communicate application intent to the Linux kernel networking stack.

We reserve one bit in *fwmark* to signal that the connection for the given socket must not be rerouted. This allows CPR to be disabled on a per-connection basis for measurement and debugging purposes. For example, it may be necessary for a connection to be sent over an application-defined path to test path performance or to debug networking issues.

In order to support UDP based transport protocols, we allow userspace applications to directly manipulate the value of r. Surfacing control over the reroute counter through the *fwmark* effectively pushes route reselection out of the kernel.

QUIC. One of the primary drivers for userspace support of CPR has been the ongoing deployment of QUIC [38]. QUIC supports connection migration [31], and over time we expect this to be the primary mechanism for path failover. As of today, however, QUIC connection migration may be disabled by either peer, and can only be used after connection establishment. Furthermore, migration must be initiated by the receiver, who must send a probe packet from a new local address. CPR on the other hand covers connection establishment, and can be triggered unilaterally on the sender side, where stalls are more quickly detected. Since there is nothing intrinsic to CPR that makes it fundamentally incompatible with connection migration, we view both methods as complementary and are actively experimenting with CPR within QUIC. While validating the use of CPR within QUIC is the subject of future work, the implementation itself is straightforward and bears testament to the overall elegance and simplicity of CPR.

7 Related work

Although multiple solutions in the literature address the detection and mitigation of path impairments, none provides feature parity with CPR. To the best of our knowledge, the design of CPR is unique in its simplicity, which makes it extremely easy to deploy. It is the only solution providing path failover at a connection granularity, which does not require switches with programmable dataplanes or support from the client endpoint, and which makes it possible to identify and mitigate performance degradation even if a connection is not currently established.

Our work is most closely related in motivation and approach to INFLEX [58] and Blink [28]. Both use transport-layer triggers to reroute traffic and, unlike CPR, rely on programmable data plane switches. While INFLEX reroutes stalled flows on a per-connection basis, it does so by installing ephemeral route entries onto a switch. It can therefore not support failover for all connections at the scale edge networks operate. Blink on the other hand relies on the ability to monitor sequence numbers of ongoing TCP flows. This poses not only scale concerns, which Blink addresses by limiting itself to only monitoring high volume prefixes, but also cannot work on encrypted transport protocols such as QUIC. Finally, by detecting failures and re-routing on a route/prefix granularity, Blink suffers from the same pitfalls of traditional

Table 3: Comparison of path failover mechanisms

Feature	BGP [48]	MPTCP [24]	SWIFT [29]	INFLEX [58]	Blink [28]	CPR
Pre-establish connection support	✓	✗	✗	✓	partial	✓
Partial failures support	✗	✓	✗	✓	partial	✓
Granular failover	✗	✓	✗	✓	✗	✓
Support all connections and prefixes	✓	✓	✓	✗	✗	✓
No client support required	✓	✗	✓	✓	✓	✓
No switch support required	✓	✓	✗	✗	✗	✓
$O(\texttt{seconds})$ convergence	✗	✓	✗	✓	✓	✓
Production validation	✓	✓	✗	✗	✗	✓

routing protocols. Smaller failures can go undetected, and reroute decisions can result in substantial changes in traffic allocation, exacerbating the risk of cascading failures due to downstream congestion. CPR on the other hand can be enabled for all flows, and can detect outages and impairments on a per-connection basis. Re-routing is also performed on a per-flow basis, and as a result traffic can be offloaded and distributed in a more granular fashion. CPR can also afford more sophisticated failure detection implementations, for example based on RTT variation, that passive monitoring in switch dataplanes cannot support.

SWIFT [29] infers the extent of failures through the analysis of BGP updates. While it reduces the time to route convergence, its reliance on BGP route updates makes it oblivious to a wide array of failure scenarios. By the authors' own admission, it is also unable to improve the time taken to receive BGP updates, which itself is on the order of multiple seconds [28]. CPR bypasses routing messages altogether, and can more accurately detect more types of failure in less time by simply piggy-backing on existing transport mechanisms.

Although a number of egress routing control systems aim to mitigate some of the same issues as CPR (*e.g.,* Espresso [64] or Edge Fabric [52]), CPR again differentiates itself by operating on individual connections rather than prefixes. This makes it complementary to such systems: it is possible to use them to apply intelligent egress traffic engineering, whilst still relying on transport extensions like CPR to mitigate transient, sub-prefix path impairments. In fact, since CPR depends on specific routing primitives for its operation (*e.g.,* those provided by Silverton [14, 57]), CPR *requires* a subset of the functionality provided by these systems.

FlowBender [34], similarly to CPR, uses transport-layer metrics to trigger path reselection decisions, but for the purpose of intra-datacenter load balancing. However, in contrast to CPR, it 1) requires Explicit Congestion Notification (ECN) which is not safe to use with anycast traffic [49, 59]; and 2) requires control of the ECMP hashing configuration of routers along the path to implement its reroute mechanism. These two issues make it unsuitable for providing resilience against Internet path failures. Similarly, although SD-WAN solutions (*e.g.,* [22, 65]) can achieve for overlay networks results similar to CPR, they are neither unilaterally deployable nor a good fit to the business model of an edge cloud network.

Multipath TCP (MPTCP) [24] makes it possible to establish single connections over multiple paths and load balance traffic automatically to the best performing path. Inheriting from previous multipath-aware transport such as shim6 [44] and SCTP [55], MPTCP has slowly gained traction in niches such connection handoff [10]. In the same way the usage of CPR is not at odds with QUIC, we do not view CPR as inherently incompatible with MPTCP. CPR works well precisely where MPTCP often falls short, for example by improving reliability for short lived flows which dominate web traffic, protecting connection establishment, or exploiting path diversity without requiring clients to explicitly identify distinct paths (*e.g.,* with distinct addresses). In contrast with CPR, MPTCP provides no value when interacting with legacy TCP clients, since it is not unilaterally deployable. By tying itself to a legacy wire format, MPTCP is also less forward looking than QUIC. Given the significant economic investment required to deploy and support a new transport protocol, it is unsurprising that edge networks have collectively focused on the latter. Over time we expect QUIC to fully assimilate the explicit multipath capabilities of MPTCP, while still falling back to implicit multipath mechanisms such as CPR.

8 Conclusion

This paper presented Connection Path Reselection (CPR), a novel system that improves the reliability of *edge networks* [52, 64] by inferring the status of a path by monitoring transport-layer forward progress, and rerouting stalled flows onto healthy paths.

CPR is unabashedly simple, and follows in a long tradition of incremental improvements to the Internet which push the boundaries of best effort delivery. While it cannot protect flows against all failures, the cases in which it is effective come virtually for free. Our implementation is trivial to configure, effortless to operate, requires no additional hardware and exploits path diversity already available to edge networks. By operating at the transport layer, CPR provides faster recovery than is attainable by routing alone, as well as detecting a wider array of potential failures on a per-connection basis without incurring additional state.

Importantly, this paper is not a proposal - CPR has been deployed in production at a large scale edge network for over two years. We evaluate our design within this context, and document our assumptions for the benefit of a wider community. To the extent of our knowledge, CPR both complements existing routing and traffic engineering mechanisms, and can coexist with multipath enhancements to transport protocols in the future. As a bridge between the two, CPR is a step towards the collective goal of ensuring that only a complete partition is capable of stalling an Internet transport connection.

Acknowledgements

We would like to thank our shepherd Italo Cunha and the anonymous reviewers for their feedback. We are also very grateful to the engineers at Fastly who have contributed to the deployment, monitoring and operation of CPR over the years.

References

[1] The CAIDA UCSD AS classification dataset, 2020-07-01. https://www.caida.org/data/as-classification.

[2] Geometric distribution. In *Statistical Distributions*, chapter 23, pages 114–116. John Wiley & Sons, Ltd, 2010.

[3] Linux ip sysctls. https://www.kernel.org/doc/Documentation/networking/ip-sysctl.txt, Aug. 2020.

[4] SONiC default BGP hold timer. https://github.com/Azure/sonic-buildimage/blob/833584eff96fca37579d2d792807a8b69c47e701/dockers/docker-fpm-frr/frr/bgpd/templates/general/instance.conf.j2#L8-L9, Nov. 2020.

[5] FRRouting default BGP hold timer. https://github.com/FRRouting/frr/blob/ca7b6587b2a8d3d15e04c00b5201030340d5d1d2/bgpd/bgpd.h#L1756, Feb. 2021.

[6] F. Ahmed, M. Z. Shafiq, A. R. Khakpour, and A. X. Liu. Optimizing internet transit routing for Content Delivery Networks. *IEEE/ACM Transactions on Networking*, 26(1):76–89, 2018.

[7] A. Akella, B. Maggs, S. Seshan, A. Shaikh, and R. Sitaraman. A measurement-based analysis of multihoming. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '03, pages 353–364, New York, NY, USA, 2003. ACM.

[8] A. Akella, J. Pang, B. Maggs, S. Seshan, and A. Shaikh. A comparison of overlay routing and multihoming route control. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '04, pages 93–106, New York, NY, USA, 2004. ACM.

[9] A. Akella, S. Seshan, and A. Shaikh. Multihoming performance benefits: an experiment evaluation of practical enterprise strategies. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '04, page 9, USA, 2004. USENIX Association.

[10] Apple. Improving Network Reliability Using Multipath TCP. https://developer.apple.com/documentation/foundation/urlsessionconfiguration/improving_network_reliability_using_multipath_tcp.

[11] Arista. Border Gateway Protocol (BGP) - EOS 4.25.1F User Manual. https://www.arista.com/en/um-eos/eos-border-gateway-protocol-bgp, 2021.

[12] B. Augustin, T. Friedman, and R. Teixeira. Measuring load-balanced paths in the internet. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, IMC '07, pages 149–160, New York, NY, USA, 2007. Association for Computing Machinery.

[13] B. Augustin, T. Friedman, and R. Teixeira. Measuring multipath routing in the internet. *IEEE/ACM Transactions on Networking*, 19(3):830–840, June 2011.

[14] D. Barroso. Developing and evolving your own control plane. NANOG '71. https://pc.nanog.org/static/published/meetings/NANOG71/1438/20171002_Barroso_Developing_And_Evolving_v1.pdf, Oct. 2017.

[15] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, oct 2008.

[16] O. Bonaventure and S. Seo. Multipath TCP deployments. *IETF Journal*, 2016, November 2016.

[17] M. Calder, R. Gao, M. Schröder, R. Stewart, J. Padhye, R. Mahajan, G. Ananthanarayanan, and E. Katz-Bassett. Odin: Microsoft's scalable fault-tolerant CDN measurement system. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 501–517, Renton, WA, apr 2018. USENIX Association.

[18] Cisco. Cisco IOS IP routing: BGP command reference. https://www.cisco.com/c/en/us/td/docs/ios/iproute_bgp/command/reference/irg_book/irg_bgp4.html, Oct. 2013.

[19] D. Clark. The design philosophy of the DARPA internet protocols. In *Symposium Proceedings on Communications Architectures and Protocols*, SIGCOMM '88, pages 106–114, New York, NY, USA, 1988. Association for Computing Machinery.

[20] A. Clauset, M. E. J. Newman, and C. Moore. Finding community structure in very large networks. *Phys. Rev. E*, 70:066111, Dec 2004.

[21] A. Dhamdhere and C. Dovrolis. ISP and egress path selection for multihomed networks. In *Proceedings IEEE*

INFOCOM 2006. 25th IEEE International Conference on Computer Communications, pages 1–12, 2006.

[22] Z. Duliński, R. Stankiewicz, G. Rzym, and P. Wydrych. Dynamic traffic management for SD-WAN inter-cloud communication. *IEEE Journal on Selected Areas in Communications*, 38(7):1335–1351, 2020.

[23] O. Filip, P. Machek, M. Mares, M. Matejka, and O. Zajicek. BIRD 2.0 user's guide. `https://bird.network.cz/?get_doc&v=20&f=bird-6.html`, Oct. 2019.

[24] A. Ford, C. Raiciu, M. J. Handley, O. Bonaventure, and C. Paasch. TCP extensions for multipath operation with multiple addresses. RFC 8684, RFC Editor, March 2020.

[25] V. Giotsas, C. Dietzel, G. Smaragdakis, A. Feldmann, A. Berger, and E. Aben. Detecting peering infrastructure outages in the wild. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 446–459, New York, NY, USA, 2017. ACM.

[26] D. Goodin. BGP event sends European mobile traffic through China Telecom for 2 hour. `https://arstechnica.com/information-technology/2019/06/bgp-mishap-sends-european-mobile-traffic-through-china-telecom-for-2-hours/`, June 2019.

[27] J. He and J. Rexford. Toward internet-wide multipath routing. *Netwrk. Mag. of Global Internetwkg.*, 22(2):16–21, Mar. 2008.

[28] T. Holterbach, E. C. Molero, M. Apostolaki, A. Dainotti, S. Vissicchio, and L. Vanbever. Blink: fast connectivity recovery entirely in the data plane. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 161–176, Boston, MA, Feb. 2019. USENIX Association.

[29] T. Holterbach, S. Vissicchio, A. Dainotti, and L. Vanbever. SWIFT:predictive fast reroute. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 460–473, New York, NY, USA, 2017. ACM.

[30] J. Hopcroft and R. Tarjan. Algorithm 447: Efficient algorithms for graph manipulation. *Commun. ACM*, 16(6):372–378, June 1973.

[31] J. Iyengar and M. Thomson. QUIC: a UDP-based multiplexed and secure transport. Internet-Draft draft-ietf-quic-transport-34, IETF Secretariat, January 2021.

[32] Y. Jin, S. Renganathan, G. Ananthanarayanan, J. Jiang, V. N. Padmanabhan, M. Schroder, M. Calder, and A. Krishnamurthy. Zooming in on wide-area latencies to a global cloud provider. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, pages 104–116, New York, NY, USA, 2019. Association for Computing Machinery.

[33] Juniper. hold-time (protocols BGP). `https://www.juniper.net/documentation/en_US/junos/topics/reference/configuration-statement/hold-time-edit-protocols-bgp.html`, Dec. 2020.

[34] A. Kabbani, B. Vamanan, J. Hasan, and F. Duchene. Flowbender: flow-level adaptive routing for improved latency and throughput in datacenter networks. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, pages 149–160, New York, NY, USA, 2014. Association for Computing Machinery.

[35] N. Kephart. Route leak causes global outage in level 3 network. `https://www.wired.com/2008/02/pakistans-accid/`, Feb. 2015.

[36] N. Keukeleire, B. Hesmans, and O. Bonaventure. Increasing broadband reach with hybrid access networks. *IEEE Communications Standards Magazine*, 4(1):43–49, 2020.

[37] C. Labovitz, A. Ahuja, A. Bose, and F. Jahanian. Delayed internet routing convergence. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '00, pages 175–187, New York, NY, USA, 2000. Association for Computing Machinery.

[38] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W.-T. Chang, and Z. Shi. The QUIC transport protocol: design and internet-scale deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 183–196, New York, NY, USA, 2017. ACM.

[39] S. Larson. Here's why you may have had internet problems today. `https://money.cnn.com/2017/11/06/technology/business/internet-outage-comcast-level-3/index.html`, June 2017.

[40] H. H. Liu, Y. Wang, Y. R. Yang, H. Wang, and C. Tian. Optimizing cost and performance for content multihoming. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 371–382, New York, NY, USA, 2012. ACM.

[41] D. Madory. Widespread impact caused by Level 3 BGP route leak. https://dyn.com/blog/widespread-impact-caused-by-level-3-bgp-route-leak/, Nov. 2017.

[42] A. Medina. CenturyLink / Level 3 outage analysis. https://blog.thousandeyes.com/centurylink-level-3-outage-analysis/, August 31st, 2020.

[43] K. P. Murphy. *Machine learning: a probabilistic perspective*. The MIT Press, Cambridge, MA, 2012.

[44] E. Nordmark and M. Bagnulo. Shim6: Level 3 multihoming shim protocol for IPv6. RFC 5533, RFC Editor, June 2009.

[45] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: machine learning in Python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[46] B. Peirens, G. Detal, S. Barre, and O. Bonaventure. Link bonding with transparent Multipath TCP. Internet-Draft draft-peirens-mptcp-transparent-00, IETF Secretariat, July 2016.

[47] Y. Pi, S. Jamin, P. Danzig, and F. Qian. Latency imbalance among internet load-balanced paths: a cloud-centric view. In *Abstracts of the 2020 SIGMETRICS/Performance Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '20, pages 65–66, New York, NY, USA, 2020. Association for Computing Machinery.

[48] Y. Rekhter, T. Li, and S. Hares. A Border Gateway Protocol 4 (BGP-4). RFC 4271, RFC Editor, January 2006.

[49] L. Saino. Hashing on broken assumptions. NANOG '70. https://www.nanog.org/sites/default/files/1_Saino_Hashing_On_Broken_Assumptions.pdf, June 2017.

[50] M. Sargent, J. Chu, D. V. Paxson, and M. Allman. Computing tcp's retransmission timer. RFC 6298, RFC Editor, June 2011.

[51] B. Schlinker, I. Cunha, Y.-C. Chiu, S. Sundaresan, and E. Katz-Bassett. Internet performance from Facebook's edge. In *Proceedings of the Internet Measurement Conference*, IMC '19, pages 179–194, New York, NY, USA, 2019. Association for Computing Machinery.

[52] B. Schlinker, H. Kim, T. Cui, E. Katz-Bassett, H. V. Madhyastha, I. Cunha, J. Quinn, S. Hasan, P. Lapukhov, and H. Zeng. Engineering egress with Edge Fabric: steering oceans of content to the world. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 418–431, New York, NY, USA, 2017. ACM.

[53] R. Singel. Pakistan's accidental youtube re-routing exposes trust flaw in net. https://www.wired.com/2008/02/pakistans-accid/, Feb. 2008.

[54] B. Stelter. Time Warner Cable comes back from nationwide Internet outage. https://money.cnn.com/2014/08/27/media/time-warner-cable-outage/index.html, Aug. 2014.

[55] R. R. Stewart. Stream Control Transmission Protocol. RFC 4960, RFC Editor, September 2007.

[56] S. Tao, K. Xu, Y. Xu, T. Fei, L. Gao, R. Guerin, J. Kurose, D. Towsley, and Z.-L. Zhang. Exploring the performance benefits of end-to-end path switching. *SIGMETRICS Performance Evaluation Review*, 32(1):418–419, June 2004.

[57] J. Taveira Araújo. Building and scaling the Fastly network, part 1: fighting the FIB. https://www.fastly.com/blog/building-and-scaling-fastly-network-part-1-fighting-fib/, May 2016.

[58] J. Taveira Araújo, R. Landa, R. G. Clegg, and G. Pavlou. Software-defined network support for transport resilience. In *2014 IEEE Network Operations and Management Symposium (NOMS)*, pages 1–8, May 2014.

[59] J. Taveira Araújo, L. Saino, L. Buytenhek, and R. Landa. Balancing on the edge: Transport affinity without network state. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 111–124, Renton, WA, 2018. USENIX Association.

[60] R. Teixeira, K. Marzullo, S. Savage, and G. M. Voelker. Characterizing and measuring path diversity of internet topologies. In *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '03, pages 304–305, New York, NY, USA, 2003. Association for Computing Machinery.

[61] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. Jarrod Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. Carey, İ. Polat, Y. Feng, E. W. Moore, J. Vand erPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro,

F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors. Scipy 1.0 – fundamental algorithms for scientific computing in Python. *arXiv e-prints*, page arXiv:1907.10121, Jul 2019.

[62] D. Walton, A. Retana, E. Chen, and J. Scudder. Advertisement of multiple paths in BGP. RFC 7911, RFC Editor, July 2016.

[63] L. Wasserman. *All of Nonparametric Statistics (Springer Texts in Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006.

[64] K.-K. Yap, M. Motiwala, J. Rahe, S. Padgett, M. Holliman, G. Baldus, M. Hines, T. Kim, A. Narayanan, A. Jain, V. Lin, C. Rice, B. Rogan, A. Singh, B. Tanaka, M. Verma, P. Sood, M. Tariq, M. Tierney, D. Trumic, V. Valancius, C. Ying, M. Kallahalla, B. Koley, and A. Vahdat. Taking the edge off with Espresso: Scale, reliability and programmability for global internet peering. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 432–445, New York, NY, USA, 2017. ACM.

[65] D. Zad Tootaghaj, F. Ahmed, P. Sharma, and M. Yannakakis. Homa: an efficient topology and route management approach in SD-WAN overlays. In *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*, pages 2351–2360. IEEE Press, 2020.

Appendix

A Path diversity

This section provides additional results on path diversity both at the AS path level (extending the analysis presented in §5.3) and at the dataplane level in the specific case of paths between pairs of POPs.

A.1 AS path diversity

This section extends the analysis of AS path diversity presented in §5.3 by reporting path diversity results disaggregated by remote AS type and IP version.

We classify destination ASes according to the CAIDA AS classification dataset [1]. This dataset provides an *AS type* for each candidate destination AS: *Transit/Access* for business providing Internet connectivity; *Content* for ASes which provide content hosting and distribution; and *Enterprise* for other entities that are mostly users, rather than providers of Internet access, transit or content. ASNs for which a classification is unavailable were tagged as *Unknown*.

Our findings are presented in fig. 8. Path diversity seems to induce a stable ordering in which *Content* destination ASes

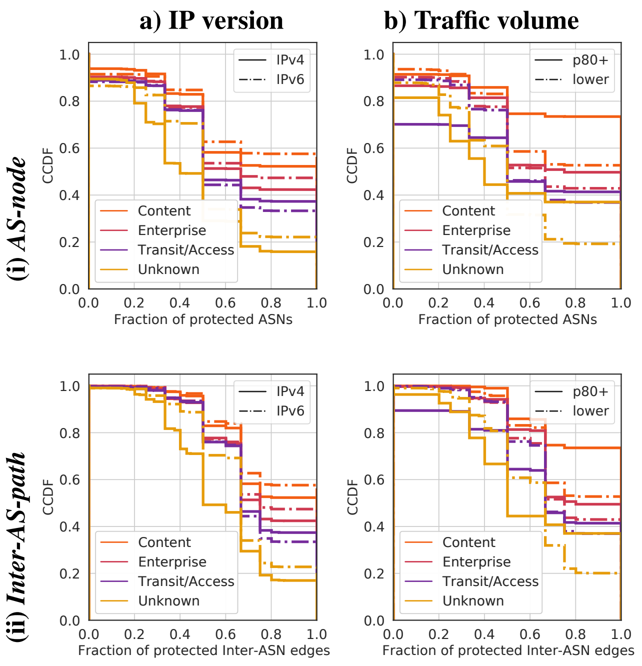
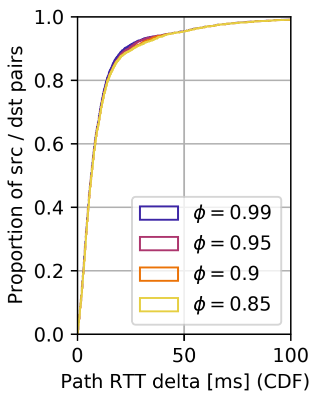


Figure 8: Relationship between both AS-node / Inter-AS path diversity and *AS type*, disaggregated by a) *IP version* of the advertised prefix and b) *Traffic volume* served.

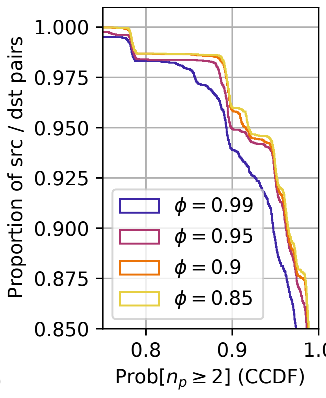
have the highest diversity, followed by *Enterprise*, *Transit/Access* and *Unknown*. While Inter-AS path diversity is understandably higher than AS-node diversity, the ordering appears for both measures. This suggests that, in general, *Content* ASes value path diversity the most; this is unsurprising given the benefits multihoming provides in content delivery [7, 40]. Although similar arguments can be made for multihoming of *Access* [8, 21] and *Enterprise* [9] networks, in practice they exhibit lower path diversity, potentially because they operate at a different cost tradeoff point. Whereas every intermediate AS is node-protected for ~50% of *Content* ASes, this only happens for ~40% of the *Enterprise* and *Transit/Access* prefixes. Overall, there is slightly improved path diversity for IPv6 compared to IPv4, except for *Transit/Access* networks (see fig. 8 i-a and ii-a).

We also consider *size* in our analysis, motivated by the intuition that large, well funded entities may plausibly enjoy better path diversity than smaller ones. We approach this issue by using traffic volume as a proxy for size, disaggregating those prefixes lying in the top 20th percentile (80th percentile overall) by traffic volume served from our edge cloud network. The result of this is presented in fig. 8-b. Although the conjecture that larger entities may enjoy better path diversity is confirmed for every *AS type*, it is emphatic for *Content* destination ASes: around ~75% of high-volume content provider prefixes are fully node- and link-protected (fig. 8 i-b and ii-b respectively). While ~30% of *Transit/Access* destination prefixes seem to not be node-protected at all, as reported in fig. 8 these correspond to directly connected peers for which path reliability is very high. On the other hand, ~40% of *Transit/Access* destination prefixes are both AS-node and Inter-AS-path protected.

A.2 Dataplane path diversity between POPs



(a) RTT delta between the shortest and longest paths between POP pairs



(b) Proportion of time that two or more paths are available between POP pairs

Figure 9: Inter-POP end-to-end path measurements

Although the BGP-based results presented above are sufficient to justify our path selection approach, as stated in §5.3 they constitute lower bounds. For some use cases it may be advantageous to directly measure dataplane multipath diversity. In this section we present data indicative of the results of such an analysis. We make use of a measurement mesh between our ~80 globally distributed POPs, which continually perform independent RTT measurements over each one of their transit providers. By exploring loss episodes in this mesh we can estimate the probability that, if a BGP-preferred path is unavailable between two POPs, CPR can find another one by random choice.

During a given 30 second measurement window, we compute the *availability* α of a POP-to-POP path as the ratio between the number of *ping* probes for which a response was received and the total number of probes sent. We then compare this to a threshold ϕ, and define a path as *unavailable* if $\alpha < \phi$. We present two measurements using this setup. First, in fig. 9a we show the CDF of the RTT difference between the shortest and longest *available* paths between two given POPs. We can see that in ~80% of cases this RTT delta is of at most ~12.5ms; for ~95% of cases this upper bound rises to 50ms. Note that, in general, *the behavior of this RTT difference is not impacted by our choice of* ϕ. This suggests that, for a large proportion of paths between POPs, alternative paths with similar RTT are available simply by choosing an alternative egress - even for relatively high availability requirements. In some cases, though, these paths *can* induce significant additional delay, and hence this effect cannot be dismissed out of hand.

We also compute the proportion of measurement windows, over a 7 day period, in which at least two POP-to-POP paths were available between any two POPs. This is presented in fig. 9b, which shows that for $\phi = 0.9$, ~96% of POP pairs have two or more *available* paths between them for at least ~90% of the time. Since data collection noise from using *ping* invariably leads to packet losses that do not correspond to path impairments, these numbers constitute a lower bound for the actual data plane path availability. Nevertheless, even this rough approximation shows that the random retry strategy presented in §4 with alternative paths being simply defined by using alternative egress transits is good enough for CPR.

B CPR state probability reconstruction

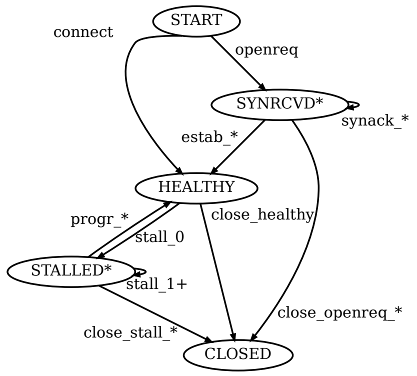


Figure 10: The CPR Finite State Machine

This section provides additional details regarding the computation of stall-event-related CPR performance measurements like those described in §5.2.

CPR state machine. A summarized representation of the CPR state machine is presented in fig. 10. Although CPR tracks the number of individual reroutes while in the `SYNRCVD*` and `STALLED*` states, these internal subdivisions and their associated transitions have been omitted (this is denoted in fig. 10 with * or +). Stall detection logic is evaluated at relevant TCP events (*e.g.,* retransmissions). Transitions between states represent either TCP events (*e.g.,* `openreq`, `estab*`), CPR events (*e.g.,* `stall*`, `progr*`) or both (*e.g.,* `synack*`, `close*`). `SYNRCVD*` and `STALLED*` correspond to points in which path reselection logic can be triggered.

For incoming connections, the `SYNRCVD` meta-state is entered when a SYN is received (`openreq`); this will elicit a SYN-ACK response from the local TCP stack. If this SYN-ACK is acknowledged, the TCP connection enters the `ESTABLISHED` state, and CPR the `HEALTHY` state. Outgoing connections enter the `HEALTHY` state directly upon termination of the three-way handshake. When connections terminate naturally (`close_healthy`), they leave both the TCP and the CPR state machines. The `SYNRCVD` meta-state subsumes one `synack*` transition for every ACK/SYN-ACK retransmission, and the `STALLED` meta-state one `stall*` transition for every time a stall is declared (see §3.2). The `progr*` transitions capture the possibility of the connection making forward progress again after a number of stalls. The `close*` transition, which corresponds to connection termination, can happen before connection establishment completes (`close_openreq*`), after suffering one or more stalls (`close_stall*`), or as part of an orderly teardown while healthy (`close_healthy`).

Probability reconstruction. Each TCP connection will increment treatment or control counters *for every transition in* fig. 10. To turn these monotonically increasing counters into meaningful probability estimates they are processed in 15 second windows that we denote *export intervals*.

We begin with $\pi_\bullet(A)$, the instantaneous occupancy of a state A, where the subscript will be used to refer to the *treatment* or *control* groups. We note that the value of $\pi_\bullet(A)$ at a given time can be obtained by keeping track of the total number of connections that have entered A and subtracting the total number of connections that have left it; this can be directly computed from the exported counters.

We continue by defining $p_\bullet^\bullet$, the probability estimates for transitions in fig. 10. As before, the subscript will denote either the *treatment* or *control* groups; however, the superscript will be used to denote either *outflow* or *inflow* probabilities. Borrowing from the usual conditional probability notation, we denote an outflow probability $p_\bullet^o(B|A)$ as the probability that, when a connection transitions *out* from state A, it will move on to state B; likewise, we denote an inflow probability $p_\bullet^i(B|A)$ as the probability that a connection transitioning *in* to state B originated from state A. An estimate of $p_\bullet^o(B|A)$ at the end of an export interval can be obtained by computing the ratio of the rate at which connections transitioned from A to B and the *total* rate at which connections transition out from A during that export interval; a similar (but opposite) procedure can be used to compute $p_\bullet^i(B|A)$ (rates can be trivially computed by discrete differentiation). Using $p_\bullet^\bullet$ we can compute specialized measures that can be helpful when estimating the strength of the association between re-routing and stall-related connection properties, such as *e.g.,* the *risk* ratios

$$\rho(A) = \frac{\pi_t(A)}{\pi_c(A)}$$

and

$$\gamma^\bullet(B|A) = \frac{p_t^\bullet(B|A)}{p_c^\bullet(B|A)}.$$

C Stall event extraction

This section provides additional details regarding the extraction of CPR stall events, like those described in §5.2.

To extract *stall events* from raw performance counters we begin by estimating their *span*, defined as the set of timestamps over which they are considered to be *active* (having a correlated effect over the stall-related performance measures $\pi_\bullet/p_\bullet^\bullet/\rho/\gamma^\bullet$ of multiple hosts in the same POP). This is achieved by first identifying candidate per-host events, which can then be clustered at the POP level.

The identification of candidate per-host events begins with standard multidimensional anomaly detection (see *e.g.,* [45,61] and references therein) aimed to identify performance degradation event *candidates*; this yields a series of points in time when treatment/control performance differences over multiple time series may be suggestive of an event. In our case, we use standard peak-finding techniques (see *e.g.,* [45, 61] and references therein) to find "potentially interesting" timestamps for each stall-related performance measure time series, and use the intersection of all these subsets of active timestamps as our set of candidates. These are further refined by applying clustering and de-noising based on the inter-event durations between successive per-host event candidates. This can be achieved by identifying continuous sequences of candidate timestamps. If the separation between any two of these sequences is smaller than a given clustering threshold, they are aggregated as part of the same event candidate; conversely, if after this process their total time span is smaller than a given threshold, the whole candidate set is discarded. This will yield a number of per-host event *spans*, each subsuming multiple candidates.

To model the state of the world both before the event started and after it ended we assign to each *span* a *context*: two time periods, one preceding it and one following it. This can be used as an experimental framework of sorts: if the candidate event finding logic described above correctly identified a set of event candidates, we would expect to find a definite statistical difference between values of at least some $\pi_\bullet/p_\bullet^\bullet/\rho/\gamma^\bullet$ for timestamps within the candidate *span* and those within the *context*. For every performance time series $\rho/\gamma^\bullet$, we compute the average *Mahalanobis distance* [43] between its *context* distribution and its actual values within the candidate span. The resulting set of distances can then be used to reject the candidate if it does not provide enough evidence that the recovered span encompasses a genuine impairment. These distances will also be used to define an event *severity*, following the assumption that a larger context/span difference indicates a more impactful event.

The next stage of the computation is to bring together sets of events at the POP level that happen at approximately the same time and which may hence share an underlying cause. To achieve this, we create a graph where the vertices are the per-host events, and links denote that the spans of the two event nodes *overlap*. This will generate dense cliques when many host-level events overlap, which would happen if multiple edge hosts in a POP experienced correlated stalls as a result of the same event. Although any graph clustering algorithm could be used (*e.g.,* modularity optimization [15, 20]), trivial detection using connected components [30] has proven to be good enough. Once per-POP events have been identified, event properties from their constituent per-host events (*e.g.*, duration, severity) are aggregated into comparable per-POP event properties.

The logic described can be reused to identify spikes in BGP updates or withdrawals for all the peers and transit providers in a given POP, yielding a number of *BGP events*. These can then be used to enrich the events found from $\pi_\bullet / p^\bullet_\bullet / \rho / \gamma^\bullet$ by using a *bookending* co-occurrence heuristic: if a BGP event happens immediately before the CPR event *starts*, it bolsters the conjecture that the CPR event was *caused* by the BGP event; if it happens immediately before the CPR event *ends*, it bolsters the conjecture that it *mitigated* it. By imposing time thresholds on the maximum time difference between the CPR and the BGP event spans and measuring the deviation between the center-of-mass of the BGP event and the start and end timestamps of the CPR event it becomes possible not only to assign suspected peers or transit providers to a given per-POP event, but also to provide a measure of the association strength between a BGP event and a CPR event. We use this to define a *BGP affinity* score to each per-POP event. Finally, the resulting per-POP events are discarded if they violate any one of a number of data quality policies, such as any of the hosts involved being put into maintenance during the per-POP event span; the total number of connections in an edge node being too low, which could induce numerical instability when calculating probabilities; the total number of nodes which observed the per-POP event being too low, which could point towards a false positive; etc.

Debugging Transient Faults in Data Center Networks using Synchronized Network-wide Packet Histories

Pravein Govindan Kannan[1*] *Nishant Budhdev*[2†] *Raj Joshi*[2†] *Mun Choon Chan*[2]
[1]*IBM Research - India* [2]*National University of Singapore*

Abstract

Data center network faults are hard to debug due to their scale and complexity. With the prevalence of hard-to-reproduce transient faults, root-cause analysis of network faults is extremely difficult due to unavailability of historical data, and inability to correlate the distributed data across the network. Often, it is not possible to find the root cause using only switch-local information. To find the root cause of such transient faults, we need: 1) *Visibility*: fine-grained, packet-level and network-wide observability, 2) *Retrospection*: ability to get historical information before the fault occurs, and 3) *Correlation*: ability to correlate the information across the network.

In this work, we present the design and implementation of *S*yNDB, a tool with the aforementioned capabilities to enable root cause analysis of network faults. We implement and evaluate *S*yNDB with realistic topologies using large scale simulation and programmable switches. Our evaluations show that *S*yNDB can capture and correlate packet records over sufficiently large time windows (∼4 ms), thus facilitating the root cause analysis of various network faults.

1 Introduction

Large cloud providers need to quickly resolve network faults to meet their high SLA (service level agreement) requirements [35, 66]. However, a data center is a distributed system that is prone to bugs caused by non-deterministic timing of distributed events [43]. Therefore, debugging network failures occurring in modern data centers is extremely challenging due to the scale and complexity of interactions in a dynamic environment. Network faults in modern data center networks are often transient and non-reproducible. A recent study [66] reports that some network faults could not be reproduced even with techniques such as EverFlow [67] or Pingmesh [30]. Further, for a given network fault, the root cause can come in many forms. For example, a packet drop due to a table miss can happen either due to a parity error [66] or due to temporal inconsistency during a network update [36, 54]. Due to the complex nature of network faults, the key bottleneck in quickly resolving them lies in finding the root cause. For example, in AliBaba's production network, 90% of the total time required to resolve a network fault is spent in finding the root cause [66]. Another study from Facebook's network [47] notes that 29% of network failures go without establishing the root cause.

[*]Work done at National University of Singapore
[†]Equal Contribution

In this paper, we focus on the problem of finding the root cause of transient and hard-to-reproduce network faults with many possible root causes. To better understand the difficulty in finding the root cause, let's consider an example scenario of a microburst. When a microburst occurs at a switch port, there is a uniform distribution of packets from different sending hosts responsible for the microburst (more details in §7.2.1). This indicates a fan-in traffic pattern with no single offending flow. Now, there are two possible root causes for such a fan-in to occur. First, the sending hosts themselves are sending the data in a synchronized fashion. In this case, the issue needs to be resolved at the sending hosts by using techniques such as application-level jitter [53]. The other possible cause could be that the sending hosts are already introducing application-level jitter, but still result in a microburst due to non-deterministic interaction with other network traffic (see §7.2.1 for details).

One way to differentiate the root causes is to look at the packets involved in the microburst before it happens. If the packet arrival times at the first-hop switches connected directly to the sending hosts are synchronized, then the root cause is synchronized traffic. Otherwise, the microburst is due to non-deterministic interactions of flows in the network and even more details are required to identify the root cause. Therefore, in order to find the root causes of complex network faults, we believe that a network monitoring system needs to have the following capabilities:

- ***Visibility:*** Observability in space, the ability to observe network-wide metrics at a packet-level resolution (e.g. packet arrivals and departures at all ports for all switches). Aggregate states such as flow level statistics will not be able to provide sufficient visibility to the underlying sequence of events.
- ***Retrospection:*** Observability in time, the ability to be "always on" and look back on past network-wide states *before* the fault has occurred. When the problem is detected, the events leading to the problem would have already occurred and information related to these past events is lost unless additional effort is made to preserve the history. Such a capability is especially necessary to deal with faults that are transient, hard to reproduce and caused by non-deterministic interactions.
- ***Correlation:*** The ability to correlate network-wide events at small timescales. This is required if faults occur due to the interaction of traffic flows across multiple switches. Without this capability, it would not be possible to correlate events from different parts of the network.

Going back to the aforementioned microburst scenario, *retrospection* allows us to look at network metrics before the microburst at the different first-hop switches with *visibility* at the granularity of packet arrivals. *Correlation* allows us to compare these arrival times across the different first-hop switches. If the root cause is not synchronous source traffic, we can again make use of these three capabilities to see what other non-deterministic interactions transpired in the network that led to the fan-in microburst. We elaborate more on this latter root cause in §7.2.1.

Several of the existing approaches [7, 32, 34, 37, 62, 63, 66] provide visibility to a good extent. But they either do not provide retrospection and correlation capabilities or provide them only partially (see Table 1). Since network faults occur unpredictably, providing retrospection typically requires the system to be "always on" in terms of collecting telemetry information. In this sense, NetSight [32], an "always on" version of postcard mode INT [7] comes closest to providing all the three properties with its packet-level visibility and retrospection capabilities. However, it does not provide a strong correlation property since it assumes that the postcards are received in order and out-of-order postcards can be corrected using topology information. Therefore, it can only correlate packets within a single flow at best. A strawman approach to achieving the three properties would be to augment a NetSight like approach with a precise data-plane time synchronization mechanism such as DPTP [38] so that it can now provide strong correlation with synchronized timestamps across the switches. However, such a solution does not scale to today's large data center networks because of the "always on" approach of NetSight in recording telemetry data. While efficient recording of telemetry data is achieved by trigger-based approaches such as INT-MX [7], PathDump [59], BurstRadar [37] and NetSeer [66], they compromise on retrospection since the packet history is not recorded, especially for switches and flows not involved in the trigger.

In this paper, we present, *S*yNDB, a packet-level, synchronized network-wide monitoring and debugging framework that provides all the 3 desired capabilities of *visibility*, *retrospection* and *correlation*. For *visibility*, *S*yNDB leverages programmable data-plane switches to capture packet-level telemetry information at nanosecond time resolution. A common issue with dataplane-based telemetry systems is that the metrics to be captured need to be specified at compile time [51]. To address this issue, *S*yNDB provides an interface to the network operator to specify and change the metrics at runtime *without* having to re-program the switch data-plane. For achieving retrospection, the key trade-off is that "always on" approaches are too expensive, while cheaper trigger-based approaches do not provide strong network-wide retrospection. We find a middle ground with *S*yNDB. Our key idea is to leverage the switch data-plane as a fast temporal storage to perform continuous recording of packet-level telemetry information (*packet records*) over a moving time window (*recording window*). When no network fault is detected, the recording window moves ahead and the older data beyond the record time-length is discarded. When a network fault is detected on any switch, the switch broadcasts a priority message to other switches in the network. On receiving this message, these switches send the packet records from the recent recording window to a monitoring server (collector) for permanent storage. At the monitoring server, the synchronized, network-wide packet-level data enables root cause analysis. In this way, *S*yNDB provides retrospection efficiently by exporting network-wide historical telemetry information only when a fault occurs. To correlate the telemetry information from different switches, *S*yNDB uses DPTP [38] to synchronize the switch data-planes. DPTP is a recently proposed time synchronization protocol for the network data-plane. *S*yNDB is thus able to provide visibility, retrospection and correlation capabilities all under the same framework.
In summary, we make the following contributions:

1. We present the design of *S*yNDB, the first network monitoring and debugging framework that provides all the three capabilities of visibility, retrospection and correlation for finding the root cause of transient and hard-to-reproduce network faults (§3).
2. For flexible visibility, we develop an abstract interface and a run-time support for the operator to configure and dynamically change the operating parameters of *S*yNDB such as fault detection conditions and the recorded metrics, without needing to re-program the data-plane (§5).
3. In order to achieve efficient retrospection capability, we design packet-level telemetry caching mechanism in the data-plane (§3.2). In doing so, we address the challenges of limited data-plane storage by developing compression techniques to minimize memory requirement while still retaining packet-level statistics (§3.2.1). We also develop techniques to further reduce the telemetry information exported for each fault trigger (§3.2.1).
4. We demonstrate the effectiveness of *S*yNDB by showing how it can be used to identify the root cause for transient and hard-to-reproduce network faults (§7). In particular, we demonstrate how *S*yNDB can identify different root causes for the same network fault using two different scenarios involving a microburst.

We have implemented *S*yNDB on Intel Tofino [8] switches using P4. The packet records at the collector are stored in a relational DBMS facilitating debugging of network faults using SQL queries (§4 and §6).

While *S*yNDB is designed to deal with transient and hard-to-reproduce network faults, it can be used as a tool to debug common network faults as well (Appendix B). In addition, *S*yNDB can be considered complimentary to existing frameworks such as INT [7] and NetSeer [66] by providing the capability to perform network-wide event correlation and retrospection.

Table 1: Comparison of *S*yNDB with existing solutions

Solution	Visibility	Retrospection	Correlation
Per-packet Postcards [32]	Packet-based	Yes, Always On	Partial
INT [7]	Packet-based	No, network trigger	Partial
SwitchPointer [60]	Packet-based (Flow-level locality)	Yes, host trigger	Partial
Sketch Frameworks [46, 62]	Flow-based	Past Aggregated Counts, N/A	No
BurstRadar [37]	Packet-based	No, fixed network trigger	No
Speedlight [63]	Switch Metrics	No, Scheduled	Causal Consistency(μs)
NetSeer [66]	Flow-based	No, fixed network triggers	No
*S*yNDB	Packet-based	Recent History, Programmable network triggers	$<100ns$ (DPTP [38])

2 Related Work

Network monitoring literature is wide and extensive, but none of the existing works provide all the three capabilities of visibility, retrospection and correlation required to find the root causes of transient and non-reproducible network faults. Here we mention some of the most relevant works.

Network Based. Query-based streaming telemetry systems like Marple [51], Sonata [31], etc as well as sketch-based frameworks [34, 46, 62] can provide network-wide visibility and retrospection but only with aggregated metrics and without any network-wide correlation capability. *S*yNDB is complementary to these streaming telemetry systems in that it additionally correlates information collected across the network and supports complex fault triggers based on input from the streaming telemetry system. Systems such as BurstRadar [37], Mozart [45], and INT-MX [7] that perform trigger-based data collection cannot provide retrospection. Mozart [45] involves coordination between network devices to start collection of telemetry data while coordination in *S*yNDB is to export already collected network-wide telemetry data. Through switch-local timestamps, INT-MX can only provide partial correlation and no network-wide correlation. Both PathDump [59] and SwitchPointer [60] leverage end-host storage to collect packet-level telemetry information providing retrospection. However, the fault triggers for both of them are only host-based and SwitchPointer provides only partial correlation with its millisecond-level epochs. Speedlight [63] uses synchronized network snapshots to provide microsecond-level casual consistency which is insufficient for correlation required in the example scenario in §1. Further, since it requires advance scheduling of snapshots every few milliseconds, it cannot provide retrospection. NetSeer [66] captures the flows that were affected by certain events like packet drops, path changes, and congestion, and mainly helps in fault localization - where the fault occurred and affected which flow (5-tuple). It only provides flow-level visibility and no retrospection or correlation. NetSight [32] which is equivalent of an "always on" version of postcard mode INT [7] (INT-XD) provides network-wide packet-level visibility and retrospection. However, it does not provide strong correlation capability as it generally assumes in order arrival of postcards. Even if strong correlation can be achieved with mechanisms such as DPTP [38], NetSight's "always on" recording is very expensive and does not scale to multi-petabit data center networks [58]. As we show in §7.3, NetSight can require 5TB or more storage per switch for every hour. The main bottleneck is the limited network bandwidth available for exporting the recorded data from the switches and the slow write speeds of data storage devices where the data would eventually end up. Other than scalability, this approach is also wasteful since network faults do not occur all the time [48] and, more than 60% of data center bugs occur due to the untimely delivery of a single message [43].

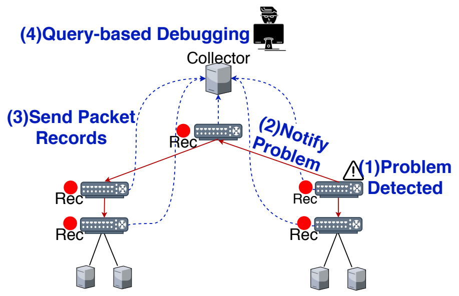


Figure 1: *S*yNDB Overview : Switches continuously maintain packet records, but send them to collector for debugging only upon detecting a problem

End-Host Based. Trumpet [50] performs end-host based monitoring of packets based on specific triggers to perform coordinated monitoring of network events. DETER [44] performs TCP replay for diagnosis of fine-grained TCP events. It constructs switch queues using simulation of the recorded TCP Packets in the end-hosts. Although it can help in debugging obscure TCP performance issues, it cannot diagnose problems inside the network at small time scales. Confluo [40] provides an end-host stack to diagnose network-wide events. However, it cannot provide packet level event correlation inside the network. SIMON [26] applies network tomography technique to reconstruct the switch queuing behaviour based on NIC packet timestamps. *S*yNDB is complementary to these techniques as it provides visibility, retrospection and correlation capabilities inside the network.

Table 1 summarizes recent related works based on their capabilities of visibility, retrospection and correlation.

3 Design

*S*yNDB provides fine-grained (packet-level, nanosecond resolution) and network-wide telemetry information, in a synchronized manner to ensure *visibility*, *correlation* and *retrospection*. *Visibility* and *correlation* are ensured by collecting packet-level telemetry and leveraging data-plane time synchronization. For Retrospection, the key idea is to leverage the switch data-plane as a fast temporal storage for recording packet telemetry information over a moving time window. One of the advantages of recording telemetry information in the switch data-plane is that *S*yNDB can record information about *all* packets within a time window at line rate.

When no network fault is detected, the recording window moves ahead and older telemetry data beyond the window size is discarded. Hence, the recording window always maintains the recent history. When a trigger condition (e.g. packet loss or high latency) is observed at any switch (Step 1), high priority trigger packets are broadcast to all switches (Step 2), as shown in Figure 1. Upon reception of trigger packets, a switch collects all the packet records from the recording window and forwards them to the collector to be stored in a database (Step 3). Once data collection is completed, the operator can debug the fault using packet records collected both before and after the fault (Step 4). Debugging can be performed by operators using SQL queries. Details of the various components of *S*yNDB are explained in the following sections, namely *Visibility* (§3.1), *Retrospection* (§3.2) and *Correlation* (§3.3).

3.1 Visibility

Packet Records. To generate packet-level telemetry data, we record information for each packet that enters a switch. We call this information a *p*-record. Each *p*-record contains 3 basic fields: $[pID, pTime_{in}, pTime_{out}]$. pID is the packet ID which is comprised of a combination of the hash value of the packet headers (5-tuple flow key) and TCP/UDP checksum. The hash value helps in associating packets from the same flow, whereas the checksum helps in uniquely tracking each packet within the flow. Although hash collisions are possible, we can resolve them using topology and timing information. $pTime_{in}$ captures the time when the packet enters the switch. $pTime_{out}$ is the time when the packet leaves the switch. Additional fields are appended by the network operator to a *p*-record to capture statistics, such as queue depth, link utilization, forwarding table version, port counters, etc. An operator can specify such additional fields via *S*yNDB configuration (§5). To identify a *p*-record with a particular flow, the flow hash to 5-tuple flow key mapping could be temporarily stored in NICs (or hosts) and retrieved on demand.

3.2 Retrospection

After a *p*-record is generated in the data-plane for each packet, we store them in a ring buffer array in the switch data-plane. This ring buffer array maintains only the recent *p*-records and we call this the *history* buffer.

Trigger Initiation. While *S*yNDB collects *p*-records for each packet in the data-plane, it requires a trigger to initiate data collection. These triggers can be events such as congestion at a link, packet drops or packet reordering. The trigger conditions are monitored in the data-plane. Once a trigger is hit, the *p*-records can be transmitted to the collector.

To initiate network-wide *p*-record collection, we create a trigger packet to be broadcast (with priority) to other switches through the data-plane. In *SyNDB*, when a trigger condition is hit, a new trigger packet is created. The trigger packet is an Ethernet frame with a trigger header consisting of: 1) Trigger ID, 2) Trigger Type: unique type to classify trigger, and 3) Trigger Time: time when the trigger was hit.

The switches receiving the trigger packets further broadcast it to their neighboring switches and so on. Due to redundancies in trigger packet broadcast (multiple paths in data center topology), unless the network is partitioned, trigger packets reach the entire network. On receiving a trigger packet in the data-plane, the switch stops storing *p*-records in the *history* buffer and instead uses a fixed buffer (we call it *future* buffer) for subsequent storage of *p*-records. If a switch had previously received a trigger packet with the same ID, then it is dropped. The *history* buffer and *future* buffer contains *p*-records of packets before and after the trigger condition respectively.

Conceptually, the size of the *history* buffer that is needed to be stored for debugging purpose depends on the round-trip-time (RTT). In a data center context, recent measurements show that VM-to-VM RTTs vary between 5μs to 100μs [24]. For a packet rate of 1 Bpps, 1 million *p*-record entries could maintain at least 1 millisecond duration of history if the switch pipeline is fully utilized. This translates to packets corresponding to at least 10's of RTTs available for debugging. In practice, the packet rate is usually much lower than 1 Bpps and this translates to a much longer time window. We present our evaluation on the time window in §7.1 and discuss the sizing of *future* buffer to enable continuous recording in §7.3.

***p*-record Collection**. Upon receiving a trigger packet with a new trigger ID in the data-plane, collection of *p*-records is performed. The control-plane initiates the data-plane packet generator which generates collection packets to read the *p*-records from the history buffer. A collection packet can read only one *p*-record each time it traverses through the switch [16], before being forwarded to the collector via a mirror-port. Consequently, we recirculate the collection packet multiple times in the data-pane to coalesce multiple *p*-records into a single packet. This reduces the large serialization overhead, if each packet contained exactly one *p*-record.

Once the number of *p*-records in a packet has reached a threshold (configured by switch control-plane), the collection packet is forwarded to the collector. A collection cycle ends when all the *p*-records stored in the data-plane have been transmitted or sufficient time has elapsed since the trigger. The collection cycle repeats upon a new trigger hit. It is important to note that regular traffic forwarding is not disrupted during the trigger and collection process. For cases when an additional trigger is hit during collection process of the previous trigger, a new trigger packet is generated and collection period is extended (discussion in §7.3). Techniques such as bulk DMA read could also be employed to collect p-records. However, such techniques require additional packetization in the control-plane to forward the whole set of p-records to a controller. The pseudo-code for recording, trigger and collection is shown in Appendix A.

Aggregating Triggers. *S*yNDB supports operator to specify collection to be performed when a set of trigger conditions occur within the historical time window. To support this, upon receiving a trigger packet, *S*yNDB maps the trigger type to a bit-index in a temporal trigger bit-array. It also sends the trigger packet to the control-plane where a timer for each trigger type is maintained, and the bit corresponding to the trigger type is cleared upon expiration. Hence, the temporal trigger bit-array represents the lists of triggers that occurred in the network for the past historical time window. Based on this trigger bit-array value, collection could be configured to be performed. The triggers are customizable by the network programmer, and is presented later (§5).

3.2.1 Reducing Collection Overhead

As *S*yNDB collects data only on event triggers, the amount of data collected is expected to be much smaller compared to continuous monitoring. To further reduce the data collected for each trigger, *S*yNDB implements two mechanisms, a compression scheme on the p-record and a scope reduction on the network level.

p-Record Compression: *S*yNDB performs the following p-record compression while ensuring packet ordering:
1. Compress pID : Consecutive packets from the same flow do not need their pID stored. Instead, a counter corresponding to the last pID is incremented.
2. Compress $pTime_{in}$: Similarly, incoming packets within a time window (e.g. 64 ns) do not need the $pTime_{in}$ stored individually. Instead, a counter for the number of packets received in the past time window is incremented. The packets within the time window are assumed to have uniform inter-arrival times. The same approach is applied to $pTime_{out}$. In the best case, a single (pID, $pTime_{in}$, $pTime_{out}$) tuple plus the corresponding (n-bit) packet counters are sufficient to represent (2^n) packets in the same time window.

Reduction on the Network Level: On detecting a fault, *S*yNDB performs collection of p-records from all switches in the network. This may burden the collector with unnecessary data if the network is huge and the root cause is localized. We mitigate this by a simple technique. Each switch maintains a list of links from which it received the packets for the historical time window. Upon fault trigger, instead of broadcasting trigger packets, they are selectively multicast to only the links from where the packets were received in the recent recording window. The intuition behind this is that we are interested in where the current set of packets came from. This solution provides the ability to trace every packet which appeared in the trigger switch to its source, while reducing the number of switches involved in the collection.

3.3 Correlation

Time-synchronization. *S*yNDB uses global timing information to correlate packet records from multiple switches to help construct an accurate network-wide ordering of events. Hence, the data-plane clocks (used for $pTime_{in}$ and $pTime_{out}$) across switches are synchronized to a fine granularity to avoid timing inconsistencies. To rightly correlate established events in distributed systems using "happened before" relation, causal consistency [41] is essential. In *SyNDB*, since a trigger event is the captured reference point, causal consistency is the right model to correlate events happened before the trigger event. We derive the necessary condition to ensure causal consistency below.

Let's consider two directly connected switches X and Y, with internal clocks C_X and C_Y respectively. We denote the synchronization error $|C_X - C_Y|$ between the internal clocks by T_{err}. Packet A is transmitted from switch X to switch Y. Packet A leaves switch X at $TimeOut_X$, and enters switch Y at $TimeIn_Y$, after a propagation delay D. $TimeOut_X$ corresponds to the time packet A enters the egress pipeline in switch X after queuing. This is the latest available time in the data-plane for a packet [16]. Similarly, $TimeIn_Y$ corresponds to the time packet A enters the ingress pipeline at switch Y. Consequently, propagation delay now becomes the sum of egress pipeline delay, packet deparsing delay, MAC processing delay, and wire delay.

$$D = EgressDelay + DeparserDelay + MACDelay + WireDelay \tag{1}$$

To ensure *causal consistency* between packet records, we should see packet A leave switch X before reaching switch Y. In short, $TimeOut_X$ should be less than $TimeIn_Y$. This will be true if the synchronization error between the internal clocks is less than the propagation delay.

$$T_{err} < D \tag{2}$$

If the condition stated in this equation can be met, we can ensure consistency between any set of packets transmitted between two adjacent switches.

Previous works on network time synchronization have shown that the T_{err} between neighbouring switches is in the order of tens of ns [38, 42]. Additionally, real world data shows that D between two adjacent switches ranges between 360 ns to 1900 ns under varying traffic conditions [38]. Thus it is possible to achieve causal consistency between adjacent switches, using current time synchronization techniques. The same principle can also be extended between switches separated by several hops in the network. In such cases, we observe that the increase in propagation delay is higher than the increase in T_{err}, thus ensuring consistency between switches across multiple hops.

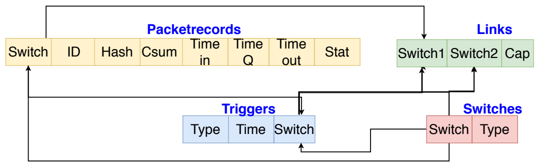


Figure 2: *S*yNDB Debugger Database Schema

4 *S*yNDB Debugger

The collector composes of multiple servers which store and analyze the *p*-records. The *p*-records are stored in a relational database (RDBMS) which allows the *p*-records to be queried using SQL. The collector also stores information regarding the trigger events, network topology, and position of switches within the topology. Before storing *p*-records in database, *S*yNDB performs hash collision removal using topology and timing information. The hash collision removal is performed using a simple heuristic based on the ground-truth of the queuing time of a *p*-record and the identity of the switch. Duplicate hashes found are then re-assigned with other *p*-record ids. The *p*-records are organized using tables in a relational database as shown in Figure 2.

1. Packetrecords: This table stores basic and custom fields within each *p*-record. Each *p*-record stores: 1) Switch ID, 2) Packet ID, 3) Packet Hash, 4) TCP/UDP Checksum 5) Time In, 6) Time Queued, 7) Time Out and, 8) Operator-specified statistics. Note that Packet ID is just a combination of packet hash and the checksum. They are stored separately to facilitate flow-level queries as well as packet-level queries.

2. Triggers: This table stores information regarding each trigger event. Each trigger event stores: 1) Trigger Type, 2) Trigger Time, and 3) Trigger Origin Switch. This enables *S*yNDB to classify network faults based on the trigger type.

3. Links: This table stores the topology of the data center, as specified by the network operator. We do not infer the topology from the packetrecords table because it is possible for some links to have zero utilization. Each link stores the endpoints and the link capacity.

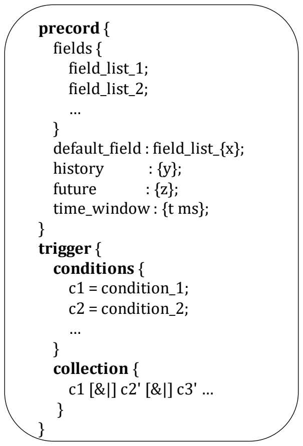
4. Switches: This table stores the position of a switch in the topology, e.g. ToR, Aggregation, Core, etc.

To determine the root cause of a network fault, we use SQL queries on the above tables. For example, in the case of an incast, culprit packets and their routes can be obtained by combining information from packetrecords, triggers, and links tables. The output of these queries can also be used to replay or build dashboards using tools [6, 12, 17], which are beyond the scope of this work.

We list some example queries below (scenarios in §7.2):

1) List the events in the trigger switch using:
```
Select * FROM packetrecords JOIN triggers
         ON packetrecords.switch = triggers.switch;
```

2) List the packets in the trigger switch and the routes taken:
```
Select * FROM (packetrecords as P) WHERE id
         IN (select id from packetrecords JOIN triggers
         ON p.switch = triggers.switch
         AND p.time_in<triggers.time) ORDERBY time_in ASC;
```



```
precord {
  fields {
    field_list_1;
    field_list_2;
    ...
  }
  default_field : field_list_{x};
  history       : {y};
  future        : {z};
  time_window : {t ms};
}
trigger {
  conditions {
    c1 = condition_1;
    c2 = condition_2;
    ...
  }
  collection {
    c1 [&|] c2' [&|] c3' ...
  }
}
```

Figure 3: *S*yNDB Configuration Syntax

5 *S*yNDB Configuration

*S*yNDB provides an interface for defining *p*-records and triggers for programmable switches. The programmer configures the following parameters in *S*yNDB: 1) the network statistics (fields) to be collected in *p*-records, 2) the number of *p*-record entries to be collected, and 3) the trigger (fault) conditions to initiate a collection of *p*-records. The fields specified in the configuration could be: 1) switch-provided metadata (queue depth, ingress port, egress port), 2) packet header data (flow-id), and 3) data that is computed and stored in user metadata by the programmer (link_utilization, counters, EWMA). The *S*yNDB configuration is compiled, then translated to P4 and finally embedded with the original switch P4 program. Figure 3 shows the interfaces to define *p*-records and trigger conditions.

A *p*-record defines a list of field_lists. Each field_list contains one or more (metadata) fields [13] from the Packet Header Vector (PHV) [16] supported by the switch architecture and defined in the user's P4 program. A "default_field" list is specified by the programmer which is the active field_list to be included in each *p*-record. The current *active* field_list can be changed during runtime. The "history" refers to the total size of the *history* buffer, while "future" refers to the size of the *future* buffer. The "time_window" is the target historical window (in milliseconds), and this is used to maintain the trigger and broadcast window. The user declares a list of trigger conditions, which are predicates operating on header/metadata fields. For example, meta.link_utilization > 90. Finally, based on the triggers declared, the collection can be configured to be performed using individual triggers or a combination (AND(&), OR(|) of multiple triggers defined). For example, let c be the local trigger condition and c' be a trigger condition happening elsewhere in the network. A representation like $c1\&c2'$ would trigger a coordinated collection by a switch A only if condition $c1$ occurs at A, and $c2$ has occurred in another switch in the network. Defining triggers conditions and collection could be based on several

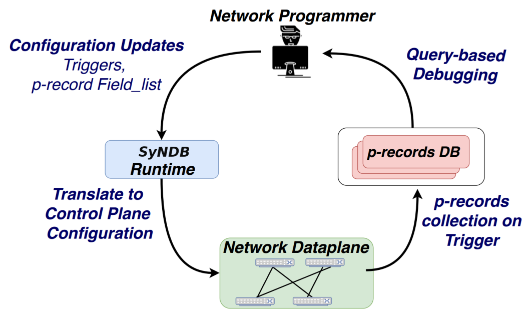


Figure 4: *S*yNDB from Programmer's perspective

network metrics in the network like packet drop, high packet queuing, loops, etc. Additionally, it could be based on well documented symptoms and alarms observed by network operators [28, 43, 47].

***SyNDB*-Runtime**. In practice, there is a need to make changes to the *p*-record structure and trigger configurations while *S*yNDB is running without the need to recompile and load a new P4 program. *SyNDB*-Runtime facilitates changing the configuration in the following ways: 1) Adding a new field_list or editing the active field_list, and 2) Adding/removing trigger conditions. Note that these changes are restricted to the available PHV contents in the data-plane as there is no modification to the parser of the underlying P4 program.

When the *S*yNDB configuration is compiled, the compiler enumerates all the PHV contents (packet headers, switch and user-defined metadata) of the P4 program. It then creates template tables with actions for each PHV container to be stored in the *p*-record. This facilitates the runtime to dynamically add/remove the fields to be recorded in each *p*-record. The fields could be TCP sequence number, TCP flags which are part of packet headers, or ingress_port, queue_depth, etc. which are part of the switch meta-data. The field to be added cannot be a metric (e.g. EWMA) that is not defined or a packet header that is not parsed by the already compiled P4 program. Since PHV contents are limited, enumerating and storing them in actions do not significantly increase data-plane resource consumption. The maximum bytes in a *p*-record and the number of *p*-record entries (*recording window*) is fixed at compile-time based on the available hardware resources (stateful ALUs and SRAM). To facilitate addition/removal of trigger conditions at runtime, *S*yNDB configuration compiler uses similar enumeration technique and generates range-based match-action tables. Since collection is performed based on the trigger bit-array value, this value is added/modified based on the collection condition changes. Additionally, *SyNDB*-Runtime updates the collector each time the *S*yNDB configuration is changed, to ensure that *p*-records are stored correctly.

Figure 4 summarizes the *S*yNDB workflow. A network programmer configures the statistics to be recorded and the fault triggers. The configuration can be continuously tweaked to suit the statistics that the programmer wants to keep an eye on using *S*yNDB-Runtime.

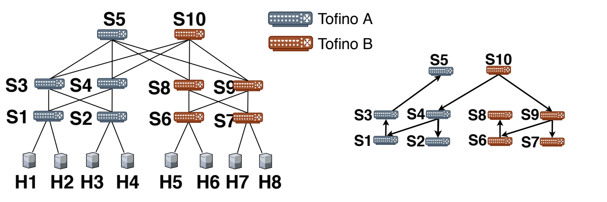
6 Implementation

***S*yNDB Dataplane.** We have implemented *S*yNDB on Intel Tofino [8] switches using P4 ($\sim$1900 LoC). We use DPTP[1] for time synchronization between switches. We use DPTP since it is implemented on PSA [16] and provides a global timestamp in the data-plane. We store the baseline contents of *p*-record in both ingress and egress pipeline. Ingress pipeline maintains the write_index of the *history* ring buffer array upon a packet arrival and stores the *pID* and $pTime_{in}$. Egress pipeline stores $pTime_{out}$ and custom field_list to be captured. *pID* is a combination of 16-bit flow hash, and 16-bit TCP/UDP checksum. $pTime_{in}$ is a 32-bit global timestamp (at nanosecond granularity) of the packet when it enters ingress pipeline of the switch. On the other hand, $pTime_{out}$ is a 24-bit field which captures time when the packet enters egress pipeline. A 24-bit value allows to calculate upto 16 ms of queuing. The basic uncompressed *p*-record is 11-bytes in size. To implement compression, we maintain separate 8-bit counter array associated with *pID*, $pTime_{in}$ and $pTime_{out}$. Trigger conditions are implemented using TCAM tables which create a trigger packet upon match. A trigger packet is created by cloning the current packet and inserting a new header type for trigger packet after stripping the payload and other headers. By using a TCAM match for the trigger table, different aggregate conditions of individual triggers can be supported using wildcards. We implement *S*yNDB control-plane, which performs : 1) Time-keeping of temporal trigger bit-array, 2) Updating multicast port-group, 3) Set up packet generators for *p*-record collection.

***S*yNDB Runtime.** We have implemented the compiler for *S*yNDB configuration using Rust ($\sim$4000 LoC). It takes as input the configuration and the switch P4 program, and generates a P4 code that implements *p*-record storage, trigger conditions and collection logic. *p*-record storage and trigger conditions are executed using stateful ALUs. Additionally, the runtime environment (implemented in Python) accepts commands to modify configuration such as: 1) changing the active field_list, 2) adding new field_list, and 3) adding/removing trigger conditions. These configurations translate to control-plane configuration updates of the composed switch P4 program. The runtime supports addition of new fields to *p*-records, by varying the recording parameters of the set of pre-enumerated PHV contents from the control-plane. A similar approach is also used for modifying trigger conditions. The maximum set of trigger conditions and contents in *p*-records are specified during the generation of the P4 program.

Finally, we implement the collector using `n2disk` utility (with *PF_RING* [15]) to store collection packets as PCAP files in the local disk. Additionally, we implement a Python program to parse the PCAP files, decompress and store individual *p*-records in a MySQL database. The collector also

[1] https://github.com/praveingk/DPTP



a. Testbed Topology

b. Network-level DAG used for DPTP synchronization

Figure 5: Evaluation Topology Testbed

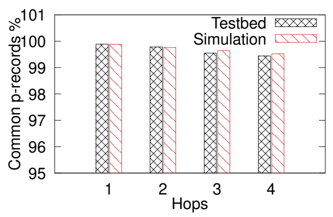


Figure 6: Comparison of testbed and simulator for common *p*-records

takes as input the *S*yNDB configuration for initializing the database. Each time the active field_list is modified through the *S*yNDB runtime, the update is passed on to the collector.

7 Evaluation

We evaluate *S*yNDB using a hardware testbed as well as large-scale simulations. Our hardware testbed consists of 4 physical servers, and 2 Intel Tofino Wedge100BF-32X switches [18]. The servers and the switches are virtualized to create a fat-tree topology [19] (see Figure 5(a)). Switches S1-S5 are virtualized on the first physical switch ("Tofino A") using 10G loopback links while switches S6-S10 are virtualized on the other physical switch ("Tofino B"). Each virtual switch is configured to have 10K and 5K *p*-records in the history and future buffer respectively. Each *p*-record entry is 16 bytes in size. Figure 7 shows the *S*yNDB configuration that we use in the testbed-based evaluation. Additionally, we synchronize each virtual switch's data-plane to S10 using DPTP (see Figure 5(b)).

For large-scale simulations, we have built a packet-level simulator[2] in C++ (~6K LoC) that implements low-level packet transmission and forwarding behaviors for hosts, links and switches. To validate our simulator, we compare its results with those from the testbed for the following experiment. For the topology in Figure 5(a), we send 10 Mpps CBR traffic along the path S1-S4-S10-S9-S7 with each switch storing 10K *p*-record entries. Switch S7 generates a trigger after receiving 10,000 packets which is then broadcast to other switches to initiate *p*-record collection. Based on the *p*-records available at the collector, in Figure 6, we plot the percentage of common *p*-records seen by other switches compared to those seen by S7. We observe that the testbed and simulation results match each other closely – due to hop delays experienced by the trigger packet, the percentage of common *p*-records reduces slightly with increasing number of hops from S7.

[2]https://github.com/rajkiranjoshi/syndb-sim

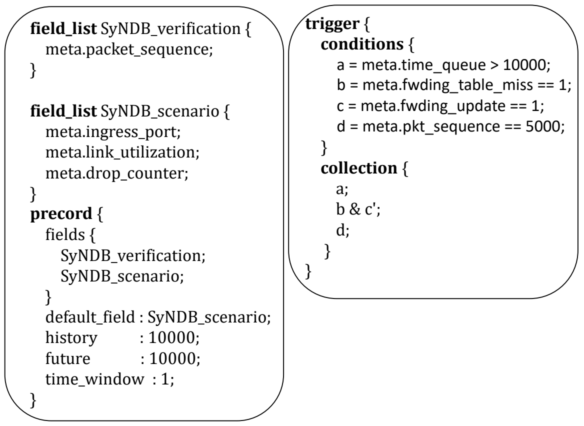

```
field_list SyNDB_verification {
    meta.packet_sequence;
}

field_list SyNDB_scenario {
    meta.ingress_port;
    meta.link_utilization;
    meta.drop_counter;
}
precord {
    fields {
        SyNDB_verification;
        SyNDB_scenario;
    }
    default_field : SyNDB_scenario;
    history       : 10000;
    future        : 10000;
    time_window   : 1;
}
```
```
trigger {
    conditions {
        a = meta.time_queue > 10000;
        b = meta.fwding_table_miss == 1;
        c = meta.fwding_update == 1;
        d = meta.pkt_sequence == 5000;
    }
    collection {
        a;
        b & c';
        d;
    }
}
```

Figure 7: *S*yNDB Configuration for Evaluation

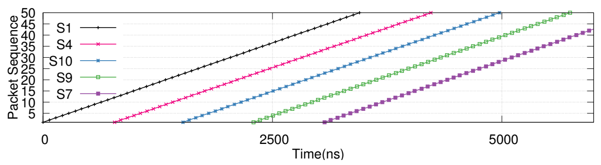


Figure 8: Sequence of packet arrival for 10Mpps CBR traffic

The evaluation is divided into three parts. In §7.1, we evaluate *S*yNDB for consistency in *p*-records and large scale operation. In §7.2, we present different network debugging scenarios with one in full details. Finally, we evaluate the overhead of *S*yNDB in §7.3.

7.1 Design Validation

In this section, we evaluate the *S*yNDB for its ability to provide consistent *p*-records at packet-level granularity and to provide retrospection and correlation at scale.

7.1.1 Consistency of *p*-records

To ensure that the *p*-records captured by *S*yNDB are consistent, the time synchronization error should be less than the propagation delay between adjacent switches (equation (2)). In our hardware testbed (Figure 5a), we measure DPTP synchronization error as well as the propagation delay between adjacent switches. We observe that the worst case synchronization error is less than 50 ns while the propagation delay varies between 400-450 ns. Thus, synchronization error is much lower than the propagation delay between two switches and hence captured *p*-records should be consistent with the ground truth. To validate this further, we send a CBR traffic of 10 Mpps (limited due to 10G host links) along the path S1-S4-S10-S9-S7, with each packet annotated with a sequence number along. The switches record the sequence number of each packet in the corresponding *p*-record. After receiving 5000 packets, switch S1 generates a trigger packet (trigger *d* in Figure 7) which when received, each switch sends the *p*-records to the collector. In Figure 8, we plot the packet sequence number against time for a sequence of 50 packets. We observe that every packet is recorded in the next switch

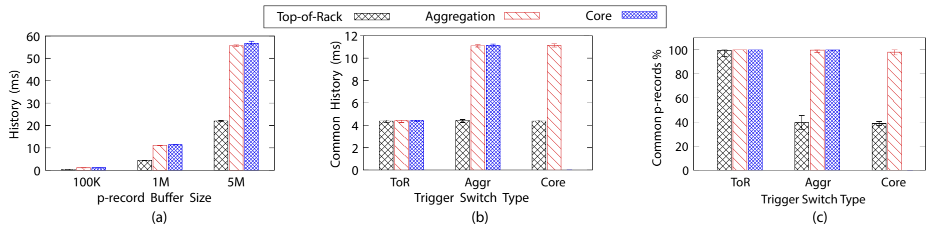


Figure 9: Simulation results for history captured at the trigger switch, correlation history and percentage of common p-records for different trigger switch types

only *after* it has left the previous switch. Furthermore, the timestamps increase linearly. This behavior confirms that the *p*-records captured by *S*yNDB across all the switches in the network are consistent and at expected intervals.

7.1.2 Retrospection and Correlation at Scale.

We perform large scale simulations to evaluate how much retrospection and correlation *S*yNDB provides in realistic traffic scenarios.

Simulation Setup. All simulation runs were done on a k=24 fat-tree topology (720 switches, 3456 hosts) with 100G links. The network has a total bisection bandwidth of 172.8 Tbps. For generating packets from the hosts, we use distributions of packet size and inter-packet gap as measured by Benson et al. [20] from a data center hosting web applications. We configure the traffic pattern such that 75% of total traffic is intra-rack as observed in cloud datacenters [20]. On top of these basic traffic characteristics, we add additional incast traffic such that 30% of the host links experience 100% utilization for 10% of total simulation time. The inter-packet gap distribution from Benson et al. is originally for 1 Gbps switch links. We scale it to adjust the load on 100 Gbps host links such that the average utilization (over 100 ms interval) is about 34%, with the busiest 5% of links experiencing about 42% utilization. These utilization characteristics are similar to those reported by Facebook [55]. All switches are configured with a *p*-record history buffer of 1M entries. We also configure a hop delay of 1 μs per switch such that the maximum RTT across the network (inter-pod) is $\sim$11.5 μs [21]. Each simulation run simulates 100 ms of network run time and delivers about 5.2B packets. Within each run, we generate 50 triggers on randomly chosen switches across the three switch types – top-of-rack (ToR), aggregation (Aggr) and Core. The following results are based on the aggregate data from 10 independent simulation runs.

We use two metrics to compare *p*-record buffers between the triggering switch and the upstream switches from which it receives packets: (i) Common *p*-records (Figure 9(c)): the percentage of common *p*-records between the trigger switch and the upstream switches. (ii) Common History (Figure 9(b)): the time difference between the latest and oldest common *p*-record. While the first metric quantifies the correlation using the similarity of *p*-records between the switches, the latter reflects the ability of *S*yNDB to perform retrospection.

For triggers originating at the ToR switch, the maximum common history that can be captured at other upstream switches is limited by the history at the ToR switch ($\sim$4 ms). Note that $\sim$4 ms history is worth $\sim$350 RTTs since maximum RTT in our setup is $\sim$11.5 μs. For triggers originating at the Aggr/Core switches, history of $\sim$11 ms is recorded. This is expected since the ToR switches experience higher packet rates (due to 75% intra-rack traffic) and hence provide smaller history relative to Aggr and ToR switches.

As for percentage of common *p*-record, we can capture $\sim$100% of common *p*-records in the upstream switches in many cases. The exceptions are for cases where the trigger switch is the Aggr/Core. The percentage of common *p*-records with other ToR switches is $\sim$40% since the *p*-records in upstream ToR switches are quickly overwritten by newer intra-rack packets. Note that in a fat-tree topology a packet passes through exactly one Core switch. Hence, if the trigger switch is a Core switch, there is no upstream Core switch.

Figure 9(a) shows the time window history that can be recorded for different *p*-record buffer sizes.

Takeaway: The simulation results show that the amount of history that can be captured depends on the incoming packet rate and the buffer size. For the traffic load and distribution used in the evaluation, *S*yNDB is able to consistently capture common *p*-record across different upstream switches. The time history available for retrospection varies from 4ms to 11ms using a buffer size of 1M *p*-record.

7.2 Network Debugging Scenario

In this section, we show how *S*yNDB can be used to debug one of the most common transient network faults, namely microburst [37, 47]. The evaluation uses the same configuration setup as defined in Figure 7 on the hardware testbed in Figure 5. Each *p*-record is configured to contain the custom "field_list: SyNDB_scenario". It contains three metrics : 1) Ingress Port 2) Link Utilization and 3) Drop Counter.

Ingress port of a packet is provided by the switch meta-data. Link utilization is calculated over a window of 10 μs in the data-plane using a low-pass-filter. Drop counter is the number of packets which missed the forwarding table. Additionally, we configure *S*yNDB to perform collection of *p*-records based on three triggers : (1) High Queuing Delay (trigger *a*), (2) Table Lookup Miss (trigger *b*) and (3) Network Configuration Update (trigger *c*). Data collection is initiated when a switch receives trigger *a* or a switch receives both triggers *b* and *c*. In each of the following case studies, we generate data and control traffic to emulate the corresponding network faults. The host data traffic is generated using MoonGen [23].

7.2.1 Microbursts

Microburst is a common problem in data centers where congestion is caused by a short burst of packets lasting for at most a few hundred microseconds [56, 65]. Traffic bursts occur due to various reasons like application traffic patterns (e.g. DFS, MapReduce), TCP behavior and also NIC-offloads (segmentation, receive) [39]. The complex interactions and traffic patterns make microbursts debugging extremely complicated. It is necessary to find the root cause to determine how the issue should be resolved.

In this experiment, we demonstrate how two microburst events that are detected by the same trigger can be attributed to different root causes using *S*yNDB. In one scenario, the microburst is due to incast of synchronized application traffic. In the other scenario, the microburst is caused by the interaction of uncorrelated flows with different source-destinations.

Synchronized Application Traffic. We consider the commonly known fan-in traffic pattern of data center networks exhibited by applications such as MapReduce and Distributed File System (DFS). This is an incast traffic pattern where many sources transmit to a small number of destinations within a short time window. These short bursts of traffic increase the queuing delay at microsecond time-scales. The challenge in identifying the root cause of such microburst is that many sources contribute to the total traffic and the burst occurs only for a very short time.

We setup the experiment with hosts H1 to H6 sending data to H8 as shown in Figure 10. Each host sends a burst of 10 1500-byte packets at an average rate of 1 Gbps to H8 via ToR switch S7. All links have capacity of 10 Gbps. In the experiment, the sources started in an asynchronized fashion, but over time transmissions from different hosts can synchronize their transmissions causing sudden spikes in queuing delays on switch S7, triggering the trigger *a*. Such synchronization of periodic messages over time has been known to occur in routing message updates [25].

With *S*yNDB, to determine if the issue of microburst is caused by synchronized fan-in traffic, a query of the queuing delay at S7 together with the packet arrival information at the ToR switches before the microburst detection can be performed at the collector as shown below:

```
SELECT switch, ingress_port, time_in  FROM packetrecords
    WHERE id IN (SELECT id FROM packetrecords AS A
    JOIN triggers as T  ON (A.time_in < T.time
    AND A.switch = T.switch)) AND switch
    IN (SELECT switch FROM switches WHERE type = "tor");
SELECT time_queue FROM packetrecords where switch=7;
```

Listing 1: Query to list the packet arrival times at ToR switch ports and queuing delay at S7

The answer to the query is shown in Figure 10. The top

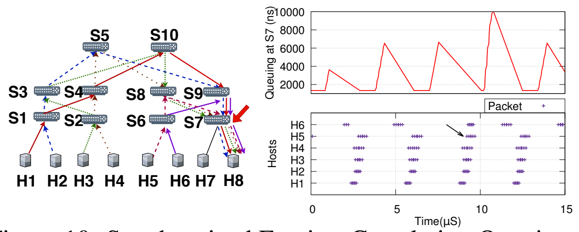


Figure 10: Synchronized Fan-in : Correlating Queuing at S7 and Packets arrival sequence at ToR Switches

right plot in the figure illustrates the queue buildup over time. When we correlate the packet arrivals from different hosts before the bursts occurred, we see that the packets that make up the bursts are transmitted by hosts H1 to H6 synchronously and reach S7 at about the same time. The root cause of this microburst from H1 to H6 can thus be determined as host-based synchronized traffic.

Non-Synchronized Application Traffic. Synchronized incast is just one possible cause for microburst. As discussed by Shan et al. [57], there are many other scenarios for microbursts. In this experiment, we generate microburst events through the interaction of traffic from multiple hosts that are not synchronized at host. However the individual flows, due to different queuing behaviour across hops (due to cross-traffic), arrive synchronously at the bottleneck link. In this scenario, hosts H1 to H6 send bursts of 10 packets at an average rate of 1Gbps to H8. A randomized delay of upto 5μs is added before sending a burst to minimize traffic synchronization. In addition, another flow sends a burst of 10 packets every 1ms of packets from H9 to H6 (through S1-S4-S5-S8-S6) at an average rate of 2Gbps. Note that this flow (H9 - H6) runs asynchronously and does not travel through the bottleneck switch (S7) where the microburst occurred. Nevertheless, we observe microbursts on the link from S7 to H8.

A query of the queuing delay at S7 together with the packet arrival information is shown in Figure 11. The shaded portion in the bottom right plot shows the duration in which the flow from H9 to H6 can occur. The information provided by *S*yNDB shows that the microburst is likely due to a combination of factors, namely (1) the synchronization of the bursts among the pair of flows from H1 & H2, H3 & H4 and H5 & H6; (2) the burst from H9 to H6 arriving just before the bursts from H1 & H2 in S1 and the bursts from H3 & H4 in S5. This causes a queue buildup resulting in packets from H1 to H6 arriving at S7 at about the same time. The root-cause is thus due to interaction of network queuing effect caused by cross traffic.

Additional Use Cases: Table 2 presents a list of additional use-case scenarios for *S*yNDB. We have experimentally evaluated (on the hardware testbed) the use-cases for debugging network faults related to network configuration updates and transient load imbalance whereby the use of multiple triggers is demonstrated. The details are provided in Appendix B.

Takeaway: The scenarios we presented show that in or-

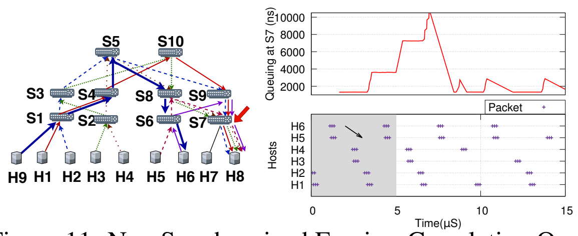


Figure 11: Non Synchronized Fan-in : Correlating Queuing at S7 and Packets arrival sequence at ToR Switches

Table 2: Use-cases of *SyNDB*

Fault and Description
Routing Bugs. Bugs in the routing protocols, for example, synchronization between LDP IGP protocols [10] could be due to timing issues such as race conditions [49]. Since *S*yNDB provides causal consistency, it helps in correlating different protocol packets and to narrow down the root-cause.
App Timing Bugs. *S*yNDB can be used to debug timing bugs in distributed systems (Hadoop [1], ZooKeeper [2]) where more than 60% of the bugs are due to a single packet [43]. In these timing bugs, a dead-lock is caused by a missed or delayed message. *S*yNDB can help to identify and track message lost or delayed by raising trigger conditions when it observes reordering or drops of certain packets.
Traffic Pattern Analysis. *S*yNDB collection could be triggered at regular intervals to study and profile traffic patterns [64] and to optimize cloud applications. In this case, *p*-records could contain the flow-id (5 tuple) to understand the interactions on a flow-level granularity.
Routing Loops Routing Loops can be detected by observing duplicate *p*-record ids at the switches.
Network Configuration Updates. Refer Appendix B
Transient Load imbalance. Refer Appendix B

der to identify the root cause of complex network faults, it is often necessary to have the *visibility* into packet statistics, the ability to look at past events (*retrospection*) and the timing information to correlate observations across switches (*correlation*). While NetSight [32] and INT-MX [7] can detect routing loops and bugs, NetSight has higher collection overhead due to its "always on" nature (§7.3). Also, while NetSight cannot perform *correlation* across the network, INT-MX cannot perform *retrospection* to identify whether the root-cause of such issues is due to configuration, race condition, etc. While Marple [51], BurstRadar [37] can detect microbursts, they do not provide *correlation* and packet-level *visibility* to inspect the root-cause of microbursts due to timing related issues like synchronized traffic. While it is possible to analyze traffic patterns in a coarse manner using systems like Speedlight [63] with tpprof [64], *S*yNDB can be used to understand microsecond-level changes in traffic.

7.2.2 Partial Deployability

*S*yNDB-enabled switches can be deployed incrementally with each new switch providing additional visibility into the network. To maximize effectiveness, deployment can start from ToR switches where most congestion events occur [65]. For DPTP synchronization, links can be added between adjacent ToR switches, which is not a complex undertaking [61]. With just *SyNDB*-enabled ToR switches, issues like microbursts(§7.2.1), application timing bugs can be debugged fully with just the ToR switches' *p*-records, while issues like routing loops, bugs, load imbalance and configuration updates can be partially debugged if the ToR switch is involved in the fault. In such cases, to infer the core network's states, network tomography techniques [26] can be employed.

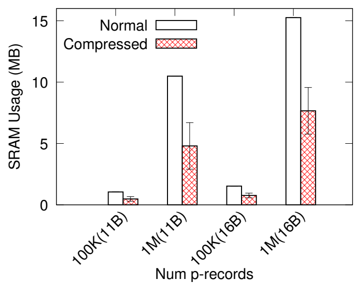


Figure 12: SRAM consumption for different packet rates and *p*-record size

7.3 *S*yNDB Overhead

SRAM Overhead. We estimate the total amount of SRAM consumption used by the *history* buffer based on a compressed *p*-record size of: 11 bytes (baseline compressed *p*-record), 16 bytes (evaluation configuration in Figure 7 + baseline compressed *p*-record) in Figure 12. We plot the SRAM consumption for different profiles in Figure 7. For example, "100K(11B)" represents 100K precords with 11-byte baseline *p*-record.*S*yNDB consumes an average of ∼5 MB while consuming ∼10 MB of SRAM to record 1 Million uncompressed baseline *p*-records respectively. For 16-byte *p*-records, we observe the SRAM overhead to be about ∼7 MB on average.

Compression saves 50% of SRAM memory on average, and can save upto 80% depending on the traffic pattern. The SRAM consumption can be easily accommodated by latest switching ASICs [3, 4, 11] which contain SRAM greater than 100 MB. Recent studies [65] have observed high utilization only across a few switch ports during congestion events. Thus the pipeline utilization is usually much lower than its capacity. To support lower packet rates like <500 Mpps, *S*yNDB uses about 2 MB of SRAM. The programmer can trade-off between the total capture duration and the memory budget.

Collection Overhead. We measure the overhead incurred at the switch to collect the *p*-records. To perform collection, the switch control-plane sets up packet generator in the data-plane packets to inject collection packets at 100 Mpps. The collection packets typically coalesce *p*-records (64 *p*-records per packet) by recirculation. Collection of 10000 compressed *p*-records requires 104 collection packets on average. We observe that it takes a total time of 245μs to evict the *p*-records. Also, it takes only 45μs to collect these packets in the data-plane using packet generator and recirculation, with the majority of the time being signalling from the control-plane to start the packet generator. We believe this timing overhead would be reduced drastically in upcoming architectures [9] which support triggering packet generation from data-plane events.

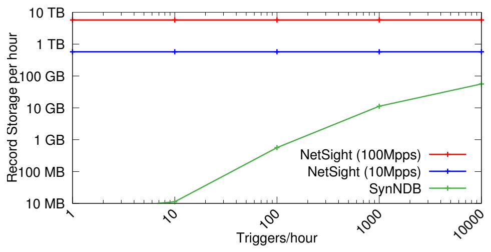


Figure 13: Storage overhead comparison with NetSight

The overall pipeline overhead incurred is about 100 Mpps and bandwidth consumption is limited to re-circulation port and the collection forwarding port (e.g. mirror port), thus not affecting regular data-plane traffic. In order to collect 1M compressed *p*-records (1ms history at 1 Bpps), it takes only about 323μs on an average. Out of this, it takes 123μs to recirculate 7800 packets to collect the compressed *p*-records, and 200μs to trigger the packet generation. *S*yNDB can resume recording (in *history* buffer) after half the *p*-records are collected in about 260μs. This means *S*yNDB can ideally support upto $\approx$ 6000 triggers/sec. Note that, *S*yNDB has a *future* buffer to store *p*-records once trigger condition is met. To support continuous recording of all future events, the minimum duration the *future* buffer needs to capture is 260μs.

With a 1ms *history* buffer, the ability to support 1000 triggers per second without any break in recording is sufficient to enable continuous monitoring. Hence, *S*yNDB can capture microbursts occurring every few milliseconds [65] as well as network incidents separated by hours [47].

We observe that the latency to receive, decompress and store the *p*-records in the collector takes few hundreds of milliseconds per switch on a single collector server. Complex queries with several join operations take several seconds or more. Query optimizations are beyond the scope of this work.

Comparison with Other Debugging Tools. Next, we compare the total storage overhead of *S*yNDB to that of NetSight [32]. NetSight creates a post-card by stripping the packet payload, and attaching switch ID and ingress port to the post-card. We compare the storage overhead incurred at the collector from a single switch for *S*yNDB compared to NetSight for a period of 1 hour. *S*yNDB performs collection only upon fault triggers while NetSight performs collection throughout the network operation. In Figure 13, we plot the overall storage incurred for an hour of network operation with increasing number of triggers/hr. We assume both NetSight and *S*yNDB store 16-byte post-cards/*p*-records per packet. Irrespective of the frequency of faults, NetSight collects about 500 GB and 5 TB of data per hour from a single switch at 10 Mpps and 100 Mpps packet rates, respectively. *S*yNDB on the other hand collects only 56 GB per hour for 10000 triggers/hr and 1 Bpps data-plane traffic. This means, when *S*yNDB monitors packets at the maximum rate (e.g. 1.6 Tbps), the total fraction of data exported for debugging is 0.01%.

Switch Resource Overhead. We evaluate the total hardware resource consumption of *SyNDB* (with configuration shown in Figure 7) compared to the baseline switch.p4 [14]. switch.p4[3] is a baseline P4 program that implements various common networking features applicable to a typical data center switch. As we implement *S*yNDB along with DPTP, we show the total resources consumed by all the components (switch.p4, DPTP and *SyNDB*) in Table 3. The majority of resources required for *S*yNDB arise from the need to store *p*-records in the data-plane. We observe that *S*yNDB consumes 33% of the stateful ALUs and 15% of the SRAM to store *p*-records and trigger conditions in the evaluation configuration. Thus, *S*yNDB can be implemented on top of switch.p4 in programmable switch ASICs available today.

Table 3: Hardware resource consumption of *S*yNDB compared to the baseline switch.p4

Resource	switch.p4	DPTP [5]	*S*yNDB	Combined
SRAM	29.58%	2.29%	15.31%	47.18%
Stateful ALU	14.58%	8.83%	33.33%	56.74%
VLIW Actions	36.72%	4.43%	6.25%	47.4%
TCAM	32.29%	0%	1.04%	33.33%
Hash Bits	34.74%	3.99%	14.14%	52.87%
Ternary Xbar	43.18%	0%	0.63%	43.81%
Exact Xbar	29.36%	2.34%	12.5%	44.2%

8 Conclusion and Discussion

In this paper, we design and implement *SyNDB*, which to the best of our knowledge, the first system providing packet-level *visibility*, *retrospection* and *correlation* to tackle transient faults. *S*yNDB leverages data-plane time synchronization and data-plane storage (SRAM) to temporally store packet records which can be exported to aid in debugging upon network faults. It provides the unique ability of looking back at the trace of events before the occurrence of a network fault. Additionally, since it performs collection only upon occurrence of programmable event triggers, it exports only a small fraction of the data-plane traffic for targeted debugging. We study case-studies which uncover *S*yNDB's capabilities in finding the root cause of transient faults.

We believe that *S*yNDB's capability goes beyond debugging. A network device's configuration can be used along with network traces to create a "replay" of the network fault. This in turn, can be used to form regression test suites. Finally, it will also be interesting to develop tools that provide dashboards, query suggestions and an assistant (Similar to Dogga et al. [22]) to network operators using AI techniques to facilitate faster debugging.

Acknowledgements. We are grateful to our shepherd Ravi Netravali, the anonymous NSDI reviewers, Anirudh Sivaraman and Facebook research for their valuable feedback on the previous drafts of this paper. This work is supported by the Singapore Ministry of Education Academic Research Fund Tier 2 (MOE2019-T2-2-134) and the Facebook Networking Systems Award (2019).

[3]DC_BASIC_PROFILE in Intel P4 Studio 8.9.2

References

[1] Apache Hadoop. http://hadoop.apache.org.

[2] Apache ZooKeeper. http://zookeeper.apache.org.

[3] Broadcom StrataXGS BCM56970 Tomahawk II. https://www.broadcom.com/news/product-releases/broadcom-first-to-deliver-64-ports-of-100ge-with-tomahawk-ii-ethernet-switch.

[4] Broadcom StrataXGS BCM56980 Tomahawk III. https://www.broadcom.com/blog/at-a-glance-tomahawk-3-is-the-first-12-8-tb-s-chip-to-achieve-mass-production.

[5] DPTP Source Code. https://github.com/praveingk/DPTP.

[6] Graphana. https://grafana.com/.

[7] In-band Network Telemetry. https://github.com/p4lang/p4-applications/blob/master/docs/INT_v2_1.pdf.

[8] Intel tofino. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html.

[9] Intel tofino 2. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html.

[10] LDP IGP Synchronization. https://tools.ietf.org/html/rfc5443.

[11] Mellanox Spectrum 2. https://www.mellanox.com/page/products_dyn?product_family=277&mtag=spectrum2_ic.

[12] NetworkX Library. https://networkx.github.io/.

[13] P4-16 Specification. https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html.

[14] P4 Language Consortium. 2018. Baseline switch.p4. https://github.com/p4lang/switch/blob/master/p4src/switch.p4.

[15] PF_RING. https://www.ntop.org/products/packet-capture/pf_ring/.

[16] Portable Switch Architecture. https://p4.org/p4-spec/docs/PSA-v1.0.0.pdf.

[17] Pyplot – Matplotlib. https://matplotlib.org/api/pyplot_api.html.

[18] WEDGE 100BF-32X. https://www.edge-core.com/productsInfo.php?cls=1&cls2=180&cls3=181&id=335.

[19] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM*, 2008.

[20] T. Benson, A. Akella, and D. A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *IMC*, 2010.

[21] D.Firestone et al. Azure accelerated networking: SmartNICs in the public cloud. In *NSDI*, 2018.

[22] P. Dogga, K. Narasimhan, A. Sivaraman, and R. Netravali. A System-Wide Debugging Assistant Powered by Natural Language Processing. In *SoCC*, 2019.

[23] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle. Moongen: A scriptable high-speed packet generator. In *IMC*, 2015.

[24] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *NSDI*, 2018.

[25] S. Floyd and V. Jacobson. The synchronization of periodic routing messages. *IEEE/ACM transactions on networking*, 2(2):122–136, 1994.

[26] Y. Geng, S. Liu, Z. Yin, A. Naik, B. Prabhakar, M. Rosenblum, and A. Vahdat. SIMON: A simple and scalable method for sensing, inference and measurement in data center networks. In *NSDI*, 2019.

[27] S. Ghorbani, Z. Yang, P. B. Godfrey, Y. Ganjali, and A. Firoozshahian. DRILL: Micro Load Balancing for Low-latency Data Center Networks. In *SIGCOMM*, 2017.

[28] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. In *SIGCOMM*, 2011.

[29] A. Guidara, S. E. P. Hetnändez, L. M. X. R. Henríquez, H. H. Kacem, and A. H. Kacem. A Study of the Forwarding Blackhole phenomenon during Software-Defined Network Updates. In *Software Defined Systems (SDS)*, 2019.

[30] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *SIGCOMM*, 2015.

[31] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger. Sonata: Query-driven streaming network telemetry. In *SIGCOMM*, 2018.

[32] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks. In *NSDI*, 2014.

[33] K. He, E. Rozner, K. Agarwal, W. Felter, J. Carter, and A. Akella. Presto: Edge-based Load Balancing for Fast Datacenter Networks. In *SIGCOMM '15*, 2015.

[34] Q. Huang, P. P. Lee, and Y. Bao. Sketchlearn: relieving user burdens in approximate measurement with automated statistical inference. In *SIGCOMM*, 2018.

[35] K. Jang, J. Sherry, H. Ballani, and T. Moncaster. Silo: Predictable message latency in the cloud. In *SIGCOMM*, 2015.

[36] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer. Dynamic Scheduling of Network Updates. In *SIGCOMM*, 2014.

[37] R. Joshi, T. Qu, M. C. Chan, B. Leong, and B. T. Loo. Burstradar: Practical real-time microburst monitoring for datacenter networks. In *APSys*, 2018.

[38] P. G. Kannan, R. Joshi, and M. C. Chan. Precise time-synchronization in the data-plane using programmable switching asics. In *SOSR*, 2019.

[39] R. Kapoor, A. C. Snoeren, G. M. Voelker, and G. Porter. Bullet Trains: A Study of NIC Burst Behavior at Microsecond Timescales. In *CoNEXT*, 2013.

[40] A. Khandelwal, R. Agarwal, and I. Stoica. Confluo: Distributed Monitoring and Diagnosis Stack for High-speed Networks. In *NSDI*, 2019.

[41] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 1978.

[42] K. S. Lee, H. Wang, V. Shrivastav, and H. Weatherspoon. Globally Synchronized Time via Datacenter Networks. In *SIGCOMM*, 2016.

[43] T. Leesatapornwongsa, J. F. Lukman, S. Lu, and H. S. Gunawi. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. In *ASPLOS*, 2016.

[44] Y. Li, R. Miao, M. Alizadeh, and M. Yu. DETER: Deterministic TCP replay for performance diagnosis. In *NSDI*, 2019.

[45] X. Liu, M. Shirazipour, M. Yu, and Y. Zhang. MOZART: Temporal Coordination of Measurement. In *SOSR*, 2016.

[46] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *SIGCOMM*, 2016.

[47] J. Meza, T. Xu, K. Veeraraghavan, and O. Mutlu. A Large Scale Study of Data Center Network Reliability. In *IMC*, 2018.

[48] J. Meza, T. Xu, K. Veeraraghavan, and O. Mutlu. A Large Scale Study of Data Center Network Reliability. In *IMC*, 2018.

[49] I. Minei and J. Lucek. *MPLS-Enabled Applications: Emerging Developments and New Technologies*. Wiley, 2008.

[50] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. Trumpet: Timely and Precise Triggers in Data Centers. In *SIGCOMM*, 2016.

[51] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim. Language-Directed Hardware Design for Network Performance Monitoring. In *SIGCOMM*, 2017.

[52] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for Network Update. In *SIGCOMM*, 2012.

[53] J. Rothshild. High Performance At Massive Scale - Lessons Learned At Facebook, 2009.

[54] A. Roy, D. Bansal, D. Brumley, H. K. Chandrappa, P. Sharma, R. Tewari, B. Arzani, and A. C. Sneoren. Cloud Datacenter SDN Monitoring: Experiences and Challenges. In *IMC*, 2018.

[55] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *Proceedings of NSDI*, 2015.

[56] D. Shan, F. Ren, P. Cheng, R. Shu, and C. Guo. Microburst in data centers: Observations, analysis, and mitigations. In *ICNP*, 2018.

[57] D. Shan, F. Ren, P. Cheng, R. Shu, and C. Guo. Microburst in data centers: Observations, analysis, and mitigations. In *Proceedings of ICNP*, 2018.

[58] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *SIGCOMM*, 2015.

[59] P. Tammana, R. Agarwal, and M. Lee. Simplifying Datacenter Network Debugging with PathDump. In *OSDI*, 2016.

[60] P. Tammana, R. Agarwal, and M. Lee. Distributed Network Monitoring and Debugging with SwitchPointer. In *NSDI*, 2018.

[61] V.Liu et al. Subways: A case for redundant, inexpensive data center edge links. In *CoNEXT*, 2015.

[62] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *SIGCOMM*, 2018.

[63] N. Yaseen, J. Sonchack, and V. Liu. Synchronized network snapshots. In *SIGCOMM*, 2018.

[64] N. Yaseen, J. Sonchack, and V. Liu. tpprof: A network traffic pattern profiler. In *NSDI*, 2020.

[65] Q. Zhang, V. Liu, H. Zeng, and A. Krishnamurthy. High-resolution measurement of data center microbursts. In *IMC*, 2017.

[66] Y. Zhou, C. Sun, H. H. Liu, R. Miao, S. Bai, B. Li, Z. Zheng, L. Zhu, Z. Shen, Y. Xi, P. Zhang, D. Cai, M. Zhang, and M. Xu. Flow Event Telemetry on Programmable Data Plane. In *SIGCOMM*, 2020.

[67] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, et al. Packet-level telemetry in large datacenter networks. In *Proceedings of SIGCOMM*, 2015.

A *S*yNDB Pseudocode

```
precordArray   : Register Buffer Array
writeIndex     : Current index to write
N              : Size of the ring buffer
POST_TRIG_SIZE : Size of buffer for post trigger
pwriteIndex    : Current index to write post trigger
triggerArray   : Temporal Trigger bit-array
triggerConditions : Bitmask configuration of
                    TriggerArray for collection
Time_Now       : Current Global Time
-------------------------------------------------------
```

Packet Record Logic

```
if packet is normalPacket:
    if collectInProgress == False:
        Store Hash, Time_now, Time_queue,
            CustomStats in precordArray[writeIndex]
        writeIndex = (writeIndex + 1) % N
        add_to_port_group(ingress_port)
    else :
        if pwriteIndex < POST_TRIG_SIZE:
            Store Hash, Time_now, Time_queue,
            CustomStats in precordArray[pwriteIndex]
            pwriteIndex = (pwriteIndex + 1)
    if triggerHit is True:
         clone(packet)

if packet is clonedPacket:
    add_header(trigger)
    remove_header(ipv4/tcp/udp)
    trigger.time = Time_now
    trigger.id   = triggerId
    trigger.type = triggerType
    recirculate()
```

Trigger Packet Logic

```
if packet is triggerPacket:
    if trigger.id != lastSeenId[trigger.source]:
        triggerArray |= 1 << (trigger.type - 1);
        lastseenId[trigger.source] = trigger.id;
    else:
        drop()
    if triggerArray in triggerConditions:
        collectInProgress = True
        Multicast(port_group)
```

Collection Packet Logic

```
if packet is collectPacket:
    if collectPacket.entries < MAX_ENTRIES_PKT:
        p-record =  precordArray[readIndex]
        readIndex = (readIndex + 1) % N
        add_header(p-record)
        collectPacket.entries++
        recirculate()
    else:
        l2fwd_to_collector()
```

B More Network Debugging scenarios

B.0.1 Network Configuration Updates

Networks operate in a dynamic environment where operators frequently modify forwarding rules and link weights to perform tasks from fault management, traffic engineering, to planned maintenance [52]. However, dynamic network configurations are complex and error prone especially if they

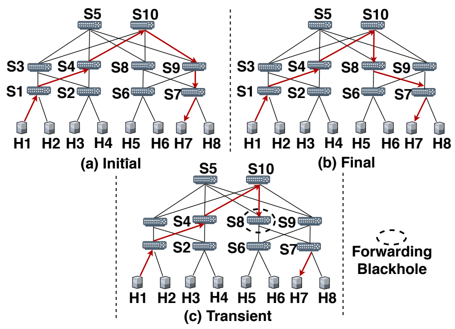


Figure 14: Network Update Scenario causing a Forwarding Blackhole at S8

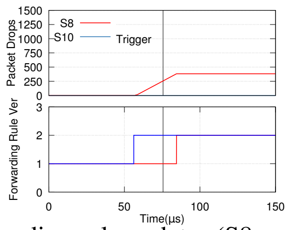


Figure 15: Forwarding rule updates (S8 and S10) leading to drops at S8

involve several devices. For example, updating the route for a flow(s) can lead to unexpected packet drops if the updates are not applied consistently or efficiently [36, 54]. In this case study, we use *S*yNDB to identify whether a transient error is due to a network update or localized hardware fault.

For the experiment, we add forward_rule_version to the *field_list SyNDB_scenario*. We assume that each forwarding rule indicates a version number and route based on destination MAC address as shown below.

```
table_add forward
    send_to_port ethernet_dstAddr <dstMac> =>
    output_port <num> entry_ver <num>
```

A transient forwarding blackhole occurs when an out-of-order execution of a network update gives rise to non-deterministic network behavior leading to temporary packet loss [29]. We emulate the transient blackhole using a setup shown in Figure 14. Figures 14(a) and (b) depict the initial and final state of the network after the updated route. The routing of a flow from H1 to H7 is updated by rerouting traffic from S10 to S8. However, transitioning from configuration (a) to (b) requires updates to both S10 and S8.

In this network update, a new rule to route the flow needs to be added to S8 first and then S10 needs to update the policy to route the flows from S9 to S8. If the update at S8 occurs later than the reroute at S10, a temporary forwarding blackhole will form, resulting in packet drops.

However, the packet drop at S8 due to table lookup miss could also be flagged as a parity error [66], when the context of the table miss is unknown. To check if a delayed network update is a possible cause, with *S*yNDB, we can query (Listing 2) the forwarding rule versions observed by each packets at S10 and S8 along with the number of drops observed in

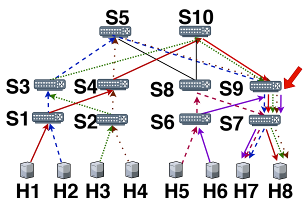


Figure 16: Congestion at S9 due to Link Load Balancing problem

Figure 15. From Figure 15, by correlating the rule version number and packet drops in time, it is clear that the packet dropped can be attributed to a transient inconsistency in rules between switches S8 and S10.

```
SELECT forwarding_rule_ver , drop_counter
    FROM packetrecords WHERE switch=8 OR switch=10;
```

Listing 2: Query for correlating network update with drops

Note that in this experiment, data collection is triggered based on an aggregating trigger defined over multiple switches. Switch S8 broadcasts the trigger *b* to other switches on detection of forwarding table miss and S10 broadcasts the trigger *c* on policy update. Trigger *b* or *c* by itself does not trigger data collection. When a switch receives both triggers (within a time window), then data collection is triggered. Such multi-switch trigger reduces both false-positives and collection overhead.

B.0.2 Transient Load Balancing Issues

Modern data center topologies such as fat-tree provide redundant paths between a source-destination pair. ECMP [27, 33, 58] is a common load balancing policy for handling multi-path routes. However, it has a lot of inefficiencies in distributing the load evenly [27, 33]. As a result, it has been observed that a subset of core-links regularly experience congestion while there is spare capacity on other links [20].

In this scenario, we setup ECMP based load balancing. Each switch calculates the hash of the 5-tuple and redirects the flow via one out of the two links. We experiment with a variety of combinations of 5-tuple flows, and use a set of combinations which can lead to load imbalance in the network. In one such combination, S9-S7 is congested, even though spare capacity is available at S8-S7. We create multiple flows in the network originating from H1 to H6 with the destination as H7 and H8 (Figure 16). The traffic (containing faulty combination) is sent at short bursts, with an overall throughput of 1 Gbps per flow. The load imbalance happens when both the core switches (S5 and S10) direct too many flows to S9,

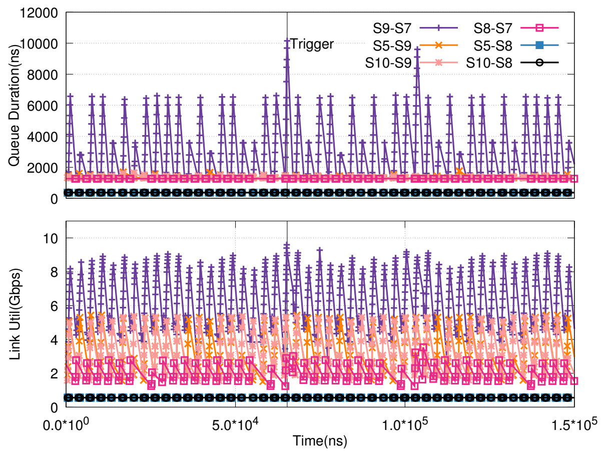


Figure 17: Link Utilization and Queuing delays observed at the core links points to load imbalance

resulting in congestion on the S9-S7 link.

With only the congestion indication, it is be difficult to determine the root cause. To determine if load imbalance is the root cause, one would have to observe the queuing duration and link utilization of various links at the same time. These network metrics are not available with both NetSight and INT. SpeedLight [63] can measure only coarse-grained link utilization (several μs). With *S*yNDB, we can plot the utilization of the links measured at the same time at packet-level granularity using the query shown in Listing 2.

```
SELECT switch1 , switch2 , link_utilization*8, time_queue
    FROM (SELECT switch1 , switch2 FROM links
    WHERE (switch1 IN (select switch FROM switches
    WHERE type !="tor") AND switch2 IN (SELECT switch
    FROM switches WHERE type !="tor"))) AS L
    JOIN (SELECT * FROM packetrecords) AS A
    JOIN (SELECT * FROM packetrecords) AS B
    ON (A.hash = B.hash AND A.switch = L.switch1
    AND B.switch = L.switch2);
SELECT forwarding_rule_ver
    FROM packetrecords WHERE switch=10;
```

Listing 3: Query for link utilization and queue depths

The result is shown in Figure 17. We can observe that there is high link utilization at S9-S7 while link S8-S7 sees no significant utilization. Furthermore, the congestion trigger at the link S9-S7 is preceded by higher than normal link utilization in links S5-S9 and S10-S9. Thus, the load distribution from the core switches (S5 and S10) to S8 and S9 is heavily skewed, with most flows being routed via S9 during some time intervals. Based on this observation, one can infer that the root cause for the congestion at the link S9-S7 is the load imbalance cause by the load balancing scheme.

Alohamora: Reviving HTTP/2 Push and Preload by Adapting Policies On the Fly

Nikhil Kansal[*], Murali Ramanujam[*], Ravi Netravali
UCLA

Abstract

Despite their promise, HTTP/2's server push and preload features have seen minimal adoption. The reason is that the efficacy of a push/preload policy depends on subtle relationships between page content, browser state, device resources, and network conditions—static policies that generalize across environments remain elusive. We present Alohamora, a system that uses Reinforcement Learning to learn (and apply) the appropriate push/preload policy for a given page load based on inputs characterizing the page structure and execution environment. To ensure practical training despite the large number of pages served by a site and the massive space of potential policies to consider for a given page, Alohamora introduces several key innovations: a page clustering strategy that favorably balances push/preload insight extraction with the number of pages required for training, and a faithful page load simulator that can evaluate a policy in several milliseconds (compared to 10s of seconds with a real browser). Experiments across a wide range of pages and mobile environments (emulation and real-world) reveal that Alohamora accelerates page loads by 19-61%, provides 3.6-4× more benefits than recent push/preload systems, and properly adapts to *never* degrade performance.

1 Introduction

Mobile web browsing has rapidly risen in popularity [15, 17, 51]. Given the importance of mobile web speeds for both user satisfaction [6, 7, 19] and content provider revenue [18], a vast array of optimizations have been developed [8, 30, 39, 40, 43, 54, 57, 61]. Yet page loads remain too slow for users in practice, taking over 10 seconds to load even with state-of-the-art mobile devices and LTE cellular networks [4, 54].

Recent studies have identified that a key culprit to slow mobile page loads is the blocking network delays that arise from the dependencies between a page's objects [39, 54]. For example, a browser may learn that it needs an image only after fetching and executing a JavaScript file, which is discovered only after downloading and parsing the page's top-level HTML. Such dependency chains essentially serialize object fetches, leading to high load times, particularly in mobile settings where access link latencies tend to be high [23, 66].

The latest HTTP/2 standard [5] anticipated the negative impact of network delays on web performance, and in response, includes several relevant features. Most notable are HTTP/2 *push* and *preload*. With push, servers can proactively send objects to clients in anticipation of future requests; requests for already-pushed objects can be satisfied locally at the client, avoiding blocking network fetches. In contrast, with preload, servers can notify clients of objects that they will soon require (potentially from other domains) by listing those URLs in HTTP headers. Clients issue requests for those objects immediately after parsing HTTP headers, and without evaluating response bodies, thereby parallelizing network and computation tasks [54].

Unfortunately, despite their promise, developing performant push/preload policies has proven to be challenging, leading to low adoption rates. For example, we find that only 5% of the Alexa top 500 pages [3] include a domain that uses push or preload; this drops to 0.9% for the Alexa top 10,000 pages. A major reason is that the performance of a given push/preload policy depends on the subtle, low-level interactions between page content, browser (cache) state and execution dependencies, client device and network resources, and QoE goals [53, 60, 69, 70]. Consequently, even for a given page, we find that using a policy outside of the execution environment for which it was designed can either forego significant (18-31%) performance benefits or degrade performance by up to 20% compared to a default page load (§2).

These results preclude the use of the static policies and guidelines promoted by prior push/preload systems [53, 54, 70], and instead highlight the need for *dynamic, adaptive* policies that explicitly target the environments in which they are deployed. For example, the aggressive push/preload policies that effectively utilize resources in high-bandwidth settings must be shrunk or dispersed across a page load as link rates drop to avoid potential network contention that slows the downloads of blocking resources. Similarly, as device CPU speeds decrease, policies should grow in size to take advantage of the (increased) blocking compute delays that leave the network idle.

We present **Alohamora**, a system that learns and applies the appropriate push/preload policies for different pages and execution environments (Figure 1). Alohamora represents its policy generation logic as an expressive neural network that is trained offline using Reinforcement Learning (RL); we list the benefits of using RL in §3. During a client page load, Alohamora's model takes as input a set of features that summarize the client's execution environment (network, CPU, cache contents), and structural information about the page at hand, and outputs a push/preload policy intended to optimize QoE for the current load. Importantly, Alohamora *does not require new browser features*, and involves minimal server changes: servers provide structural information about their pages (which content management systems commonly track [14, 30, 67]), and Alohamora's policy generation runs transparently on a co-located frontend server.

[*] These authors contributed equally to this work.

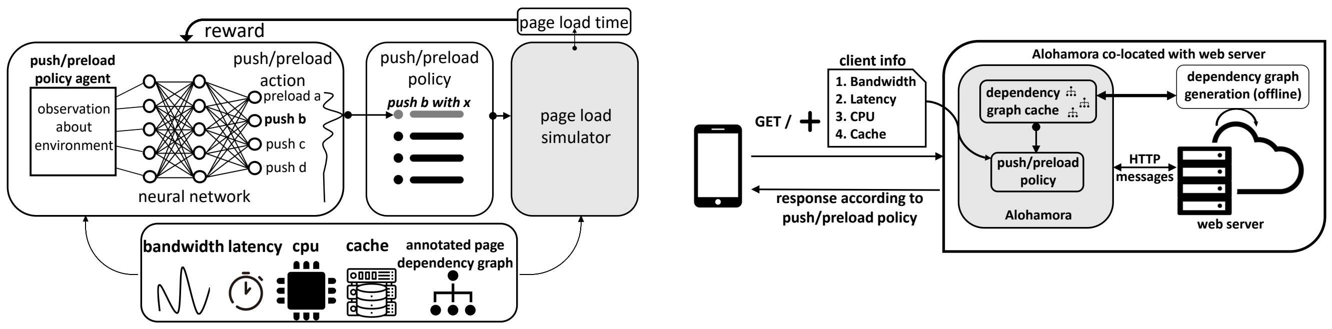


Figure 1: **Alohamora trains (left) its push/preload policy generation model using Reinforcement Learning, exploring a large search space of environments and policies, and learning from the resulting (simulated) performance. During client page loads (right), for each origin, Alohamora collects the required inference inputs from client browsers (using existing features) and servers which track changes to their page dependency graphs; the generated policies are applied transparently for the remainder of the page load.**

Realizing Alohamora's data-driven approach to HTTP/2 push/preload policy generation requires overcoming two key practical challenges with respect to training efficiency:

- **Generalizing across pages (§4)**: websites commonly serve thousands of pages, and it is impractical to incorporate each into the training process. However, failing to incorporate different pages during training may hide push/preload insights, and result in poorly generalizable models. To overcome this, Alohamora leverages our observation that even though sites serve thousands of URLs, their pages typically cover a far smaller number of page structures. The key idea is that these *shared structural properties typically dictate the efficacy of different push/preload strategies*. Thus, Alohamora needs not train on multiple pages with the same structural properties, as those would contribute similar push/preload insights. More specifically, push/preload benefits are dictated by resource utilization during the load process, which in turn can be characterized by 1) browser and inter-object dependencies, and 2) the overheads imposed by tasks involving the network and CPU. By extracting this structural information from a site's pages and clustering pages accordingly, we find that Alohamora is able to strike a desirable balance between the number of pages required for training and model generalizability.
- **Simulating page loads (§5)**: Alohamora's training involves testing different push/preload policies in diverse environments. However, the large number of potential environments and push/preload policies per page (exponential in terms of object count), coupled with the high mobile load times described above, make this approach far too slow. For example, even for a single environment, exploring the thousands of potential policies for `nytimes.com` would require 30 days on a powerful desktop machine. To handle this, Alohamora introduces a novel page load simulator which faithfully (errors of 0.4-2.2%) predicts the performance of a policy 3-4 orders of magnitude faster than running a real browser; for context, this cuts training time to 20 minutes for `nytimes.com`. To the best of our knowledge, Alohamora's simulator is the first to faithfully predict page load performance *across metrics and environmental conditions*, without requiring costly profiles [68] or emulation [60] for each environment. The key insight is in *judiciously extracting invariants about the page load process and superimposing variable resource constraints* by simulating browser-environment interactions; invariants (e.g., page/browser dependencies) are collected via a single profiling run with a real browser, while variable properties about the target environment and push/preload policy are taken as input. The simulator is general enough to support other optimizations that modulate network/compute delays [2, 40, 43, 57] or scheduling policies [8, 39].

We evaluated Alohamora using more than 500 web pages, and a wide range of mobile networks, client devices, and cache conditions. Our experiments, both emulation and real-world, reveal that Alohamora reduces page load time and Speed Index by 19-61% and 15-48%, respectively, compared to a default page load (i.e., no push/preload) and standard push/preload-all policy. In addition, Alohamora marginally (0.9-1.7×) outperforms WatchTower [43], a recent proxy-based accelerator, and delivers 3.6-4× more benefits than Vroom [54], a state-of-the-art push/preload system. Importantly, whereas Vroom slows down 24-34% of page loads, Alohamora properly adapts to *never degrade performance*. Source code and experimental data for Alohamora are available at https://github.com/nkansal96/alohamora.

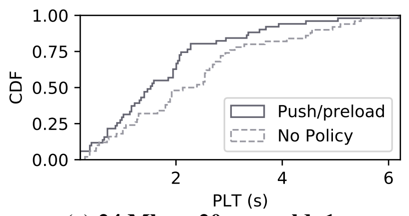
2 Background and Motivation

We begin with an overview of HTTP/2 (§2.1), and then present measurements that illustrate the potential benefits and challenges with HTTP/2 push and preload (§2.2).

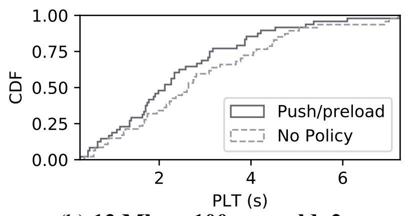
2.1 HTTP/2 Overview

HTTP/2 [5] alters the traditional HTTP/1.1 page load process primarily by adding the following new features:

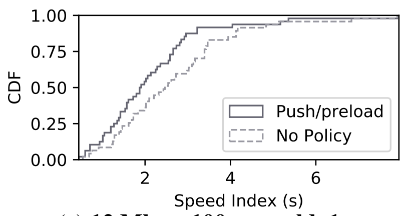
- **Request multiplexing:** With HTTP/1.1, browsers can open and reuse multiple concurrent TCP connections per origin. In contrast, HTTP/2 permits only a single con-



(a) **24 Mbps, 20 ms, cold, 1×**



(b) **12 Mbps, 100 ms, cold, 2×**



(c) **12 Mbps, 100 ms, cold, 1×**

Figure 2: **Push/preload benefits when policies are explicitly tuned to the deployment environment's available resources. Environments are listed as {bandwidth, latency, cache setting, CPU slowdown}. Results are shown for either Page Load Time (PLT) or Speed Index.**

nection per origin, and allows browsers to multiplex requests onto that connection as parallel *streams*. Unlike HTTP/1.1 pipelining, HTTP/2's multiplexing permits out-of-request-order delivery to avoid head of line blocking. HTTP/2 also mandates the use of TLS (and thus, HTTPS).

- **Server push:** Unlike HTTP/1.1 servers which only serve objects in response to explicit client requests, HTTP/2 servers can *push* objects that they own in anticipation of future requests. Servers have flexibility in defining a *push policy*, which specifies the mapping between objects that are explicitly requested and the set of files pushed along with them. Pushed objects are usable for the duration of the current page load, regardless of the associated HTTP caching headers. Note that pushed objects that are already in the browser's cache imply wasted network bandwidth.
- **Preload:** HTTP/2 also carried over HTTP/1.1's preload feature, which enables servers to list URLs to fetch directly in HTTP Link headers. Upon parsing such Link headers (i.e., before parsing the response body), browsers will immediately issue requests for the listed URLs; responses are not evaluated until the objects are referenced by the page. Thus, like push, preload enables servers to help browsers pre-warm their caches rather than relying on object execution to discover downstream objects. However, preload differs from push in that: 1) requests are client-driven and still involve network delays *to* origin servers, 2) the risk of re-downloading cached objects is eliminated since preload requests pass through the browser cache, and 3) servers can specify to preload third-party objects, not just objects that they own.
- **Stream prioritization:** HTTP/2 offers a mechanism with which both clients and servers can explicitly specify how parallel request streams on a single TCP connection share network and server-side processing resources. In particular, endpoints can annotate each request with a single integer that denotes its target share of the resources.

In this paper, we focus on the HTTP/2 push and preload features because they are configured by servers, i.e., Alohamora's target deployment location. In contrast, stream priorities are usually specified by browsers [11, 64], and yield limited benefits [31]. We note that push/preload policies exhibit a notion of prioritization that we do consider: "push A+B with C" and "push B+A with C" are different policies.

2.2 Limitations of Static Push/Preload Policies

Push/preload policies have been widely studied, yielding mixed performance results [16, 53, 54, 60, 70]. The key reason is that the performance of a policy depends on numerous page and environmental properties. To better understand the relationships between these properties and push/preload policies, we performed a study involving 50 random pages from the Alexa top 500 US sites [3]. Our results use the same methodology and environmental parameters (network, device CPU, cache, QoE metrics) described in §6.1.

For each environment and page combination, we selected the best policy using a brute force search. Since the number of potential policies for a page scales exponentially with the number of objects (which regularly exceeds 100), a complete brute force search across environments is impractical. Instead, to ensure practicality and sufficient coverage, we weighted object types based on their potential for blocking the client-side page load (i.e., JS $=$ CSS $>$ image $>$ font) [39, 59]. To generate a policy, we randomly selected the number N of objects to push/preload, and then sampled the object types N times according to their relative priorities (picking randomly within each type). Finally, we randomly selected the fraction of objects to mark as push vs. preload, and for each object, we randomly selected an earlier object in the load to push/preload from. Using this approach, for each page, we generated 200 policies and picked the one that delivered the largest improvements.

Takeaway 1: Push/preload has potential. For each environment and page pair, we compared the best push/preload policy (selected explicitly for that pair) to a default page load (i.e., no push/preload). Figure 2 shows representative results for several settings. As shown, when selected explicitly based on the environment, push/preload policies are able to provide significant speedups. For instance, in the {24 Mbps, 20 ms RTT, cold cache, 1x CPU slowdown} setting, median (95th percentile) page load time benefits are 18% (44%).

Takeaway 2: Push/preload policies do not generalize well. Despite the potential benefits, our results also highlight that push/preload policies quickly degrade in performance when run outside of the precise environments for which they were tuned. To evaluate this, we performed multiple experiments in which we started with a fixed environment, and selectively modulated each environmental factor while keeping the oth-

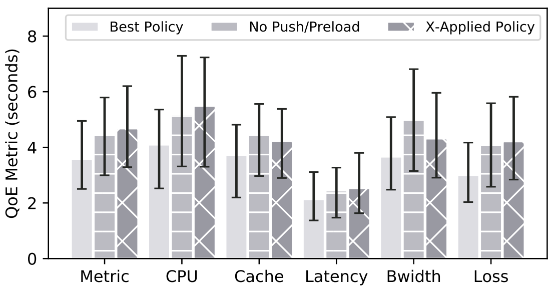


Figure 3: **Push/preload performance degrades as the environment changes. The base configuration was {12 Mbps, 100 ms, cold cache, 1x CPU, PLT}; each cluster modulates only one factor. "Best Policy" was tuned to each setting, and "X-Applied" applies the base configuration's best policy to each setting. Bars show medians, with error bars spanning 25-75th percentiles.**

ers fixed. In each resulting environment, we compared the performance of 1) the best policy for the fixed environment, 2) the best policy for the resulting environment, and 3) no push/preload. Figure 3 depicts our results for one fixed environment; we omit results for others due to space constraints, but note that the trends persist. These results illustrate two significant drawbacks to using push/preload policies across environments. First, they leave significant (18.4-30.7%) performance gains on the table compared to policies designed explicitly for the deployment setting. Second, and worse, they can degrade performance compared to a default page load. For instance, performance degrades by 6% (20%) at the median (95th percentile) when device CPU speeds change. These slowdowns are even more pronounced when multiple environmental factors are modified in parallel.

Summary: Collectively, our results suggest that, to realize the significant performance potential of push/preload, policies must be designed to explicitly consider page properties and characteristics of the target deployment environment.

3 Design Overview

Figure 1 shows the high-level design of Alohamora's offline training and online (i.e., during client page loads) inference phases. In this section, we will describe the workflow for each task in the context of a single web page. We present extensions to ensure practical training via cross-page generalization (§4) and page simulation (§5) in subsequent sections.

3.1 Offline Training

Why RL? Alohamora represents its push/preload policy generator as a neural network that is trained using Reinforcement Learning (RL) [32]. RL offers several advantages in this setting compared to more standard, supervised learning approaches. Most notably, the search space of push/preload policies is massive (exponential in terms of the number of objects in a page, which regularly exceeds 100), and it is impractical to generate a labeled training dataset that incorporates all of the fruitful push/preload policies for a page. RL overcomes this by using an efficient exploration strategy, whereby experience of prior tested policies is used to dynamically guide the traversal through the large search space.

Training with Reinforcement Learning involves learning from a large number of experiments and generally operates as follows. A *learning agent* interacts with an *environment*, and at each step, the agent *observes some state* in the environment, performs an *action*, and receives a *reward* from the environment. The overall goal of the learning agent is to maximize the cumulative (discounted) reward that it receives from the environment. In our case, the environment is a mobile page load setting, i.e., a combination of a device, network, and browser cache. The training process is structured as a series of *episodes*, each of which considers a single page and environment, and evaluates a running push/preload policy (starting with an empty one) that is incrementally modified to include an additional action. An action is a push/preload decision for a single object. We describe each component in more detail below.

Action/Action space: The action space lists the set of possible push/preload decisions for all objects in a page. Each action is represented as a six-tuple ($type$, $domain$, $push\_object$, $push\_ancestor$, $preload\_object$, $preload\_ancestor$). $type$ lists the action to perform (push, preload, nothing); $domain$ represents the domain whose objects to consider if the action is "push" ("preload" can consider objects from any domain); $push\_object/preload\_object$ and $push\_ancestor/preload\_ancestor$ list the object to push/preload and the object to do so with, respectively. Note that objects are identified by ID numbers here, not precise URLs, because URLs can vary on short time scales [8]; IDs are converted into URLs during inference (§3.2).

Episode: At the start of an episode, Alohamora first selects a random operating environment by picking values for the average network bandwidth, latency, and loss rate, as well as the mobile device CPU speed, and browser cache settings (time since the last load of the page, which in turn dictates cache contents [41]). Because the space of each value is continuous and thus infinitely large, we discretize each into bins that are sized according to prior work [43] and our own empirical analysis of the impact that changes to each factor have on push/preload performance. In particular, we group network bandwidth/latency/loss rate to the nearest 5Mbps/10ms/0.5%, and CPU speed to the nearest 1× slowdown relative to a baseline. This lets us consider far fewer environments without sacrificing model generalizability.

In addition to the environment specification, the agent is given access to an *annotated dependency graph* for the page (Figure 5). Traditional web dependency graphs [8, 39, 43, 59] are directed acyclic graphs with a node per page object, and edges that represent initiator relationships (i.e., a parent's computation triggered the fetch of a child). We add annotations (§5.1 details how annotations are made) which list, for each object, information about its 1) size, 2) computation delays, 3) content type (e.g., HTML), 4) ordering (timing) relative to both all other page objects and only those objects belonging to the same domain, 5) cache status, and 6) candidacy for push/preload. Candidacy reflects the fact that only recurring objects in a page should be considered for push/preload, in order to reduce the potential for wasting bandwidth; we determine candidacy in the same way as prior work [54], by loading the page several times and extracting a stable list of URLs. Collectively, the operating environment and annotated dependency graph represent the *observable state* that the agent can glean from the environment.

Throughout an episode, the agent selects actions according to a probability distribution over the potential space of 6-tuples (i.e., the action space described above). The probability distribution function starts as uniform, but is dynamically updated based on the agent's experiences. More formally, the agent uses a policy gradient method [33] in which it estimates the gradient of the expected total reward for each possible new action—the agent selects the action predicted to deliver the highest reward. For each new action that is added to the running push/preload policy, the updated policy is evaluated in the environment to obtain a reward that is fed back to the agent along with the *observable state*.

Each episode ends when the agent either chooses an action of *type* "nothing," repeats an action to push/preload an object that is already represented in the running policy, or selects an invalid (i.e., disallowed) action, e.g., pushing across domains or preloading an ancestor object. Regardless of which reason ends an episode, upon completion, Alohamora automatically assigns a reward of 0 to signal to the agent that the terminal policy is not one to consider. The policies learned for each episode are ultimately aggregated to generate Alohamora's overall push/preload policy generation model. We note that training can be configured to adhere to a preset bandwidth cap for pushed objects, i.e., we can end an episode if a policy that pushes an undesirably high number of bytes arises.

Reward function: Structuring the reward function requires careful thought because each action in an episode is not entirely independent. Thus, rather than simply using the page load metric of choice, we structured our reward function to take into account the relative improvement or degradation (on the metric of choice) per action, giving a boost in reward as the agent discovers a set of actions that leads to a new global (i.e., within an episode) minimum. More formally, we define the reward for the i^{th} action in an episode as:

$$R_i(P_i, P_{i-1}, P_{\text{best}}) = \begin{cases} \frac{k_1}{P_i} & P_i < P_{\text{best}} \\ \frac{k_2 P_{i-1}}{P_i} & P_i < P_{i-1} \\ \frac{-k_2 P_i}{P_{i-1}} & P_i > P_{i-1} \end{cases}$$

where P_i, P_{i-1}, and P_{best} are the raw values for the target QoE metric for the current, previous, and best-so-far policies in the episode, respectively. k_1 and k_2 are constants, where $k_1 >> k_2$. The idea is to give a positive (negative) reward proportional to an improvement (regression) in QoE. We note that the reward function is compatible with any QoE metric that denotes improved performance with lower values. We consider different reward structures in §6.5.

Implementation: Alohamora trains its models with Ray [36], using the RLLib [26] and Tune [27] libraries. Each model is a recurrent neural network that consists of 2 densely-connected layers with 256 units and the `tanh` activation function, followed by an LSTM with cell size 256. As shown in §6.5, LSTMs are helpful given the sequential nature of each episode: they prevent the agent from infinitely deferring its reward and always choosing longer policies over shorter ones. Training stops after 150 iterations, or if the standard deviation in the past 50 rewards is less than 5% of the last one (whichever comes first).

Our implementation uses the latest A3C [33] algorithm, but is compatible with others [34, 65]. As reported in prior work, A3C may incur high convergence times when network conditions or reward signals exhibit high variance during training [29]. Alohamora's training process sidesteps this in two ways. First, in addition to reducing training times, Alohamora's page load simulator eliminates noise in the observed reward signal. Second, Alohamora trains on deterministic emulated networks (including time-varying links) using Mahimahi [44], so network characteristics are unchanged within each training episode.

3.2 Online Inference

At runtime, Alohamora introduces a frontend server (or reverse proxy) that is colocated with the existing server for a domain (Figure 1); colocating ensures that end-to-end HTTPS security is preserved. All client requests first hit the Alohamora server, whose goals are to 1) collect the information required to query its policy generation model, 2) query the model, and 3) apply the suggested policy to the current load. Each origin in a page independently runs Alohamora to generate its own policy; training explores a sufficient number of policies to enable an origin to *hedge* against the set of policies that other origins may employ. We present results for partial deployment scenarios in §6.

Data collection for inference: The information required to query the policy generation model matches the *observable state* from training, i.e., network bandwidth, latency, loss rate, CPU speed, cache status, and annotated dependency graph. Alohamora collects the required network, device, and cache information through its interactions with clients, and the annotated dependency graphs directly from origin servers. Importantly, all data collection involving clients leverages existing interfaces that modern browsers already expose. In other words, *Alohamora does not require any new browser features, and instead only needs certain pre-existing features to be enabled.*

To extract network latencies, Alohamora's server analyzes the SYN/SYN-ACK time during the client's initial connection setup. Further, summaries of the client's cache are collected using either the latest cache manifest standards [46], or a server-based cookie which logs the time since the user's last load of the page [12]. We note that caching information is collected on a per-domain basis in order to preserve existing web privacy guarantees, i.e., a domain only learns about the cached objects that it owns. CPU speeds are set based on the HTTP User-Agent header that denotes the client device [37]. Lastly, average network bandwidth and loss information are collected using browser user experience reports [9].

Alohamora also requires an up-to-date dependency graph to determine the precise URLs to push/preload according to the generated policy. Alohamora relies on origin web servers to collect and share updated dependency graphs offline [30], as those servers are the first to be aware of page changes. In particular, content management systems [14, 67] support hooks that fire any time a page-altering change is pushed, e.g., for A/B testing. Alohamora adds a transparent hook to collect up-to-date dependency graphs, which requires only a lightweight (headless) load of the largely local page [40, 43].

Applying push/preload policies: Upon receiving a client request for a page, Alohamora's server queries its model to generate a push/preload policy that directly targets the current load. The resulting policy is a listing of object IDs to push/preload, and the corresponding ancestors. Alohamora then uses the latest dependency graph to translate IDs to precise URLs (according to positions in the dependency graph). Finally, to enforce the policy, the Alohamora server issues local HTTP(S) requests (mimicking client HTTP headers) to the colocated origin server, which responds with the up-to-date objects. Alohamora then applies the policy to the returned object headers throughout the rest of the page load.

4 Generalizing Across Pages

In practice, sites commonly serve thousands of different pages. Unfortunately, incorporating each page into the training process would be far too slow and resource intensive. Consequently, Alohamora faces a tricky tradeoff: train on only a few of a site's pages and achieve efficient training at the risk of omitting pages that warrant unique push/preload strategies, or train on many of a site's pages to develop generalizable policies at the expense of high training overheads.

Alohamora addresses this tradeoff by leveraging the observation that, even though sites serve thousands of different pages, those pages typically cover a *small number of page structures*, e.g., because they are automatically generated using a fixed set of templates, and thus share styles, JavaScript libraries, etc. [30, 48]. For example, news sites intuitively comprise a main home page, category home pages, and several classes of article pages. The key idea here (validated below) is that these shared structural properties typically dictate the efficacy of different push/preload strategies, and thus, we need not train on multiple pages that are structurally similar.

The primary challenge with leveraging this insight is in determining precisely which pages in a site are necessary to consider during training. Answering this implicitly requires an understanding of what pages have sufficient structural similarity from the perspective of the push/preload policies that they warrant. In other words, how should we represent and compare pages to determine structural similarity? Our goal is for representations to be coarse enough to avoid deeming all pages as structurally different (which would eliminate savings in training efficiency), but also detailed enough to capture structural differences that affect policies.

4.1 Clustering by Page Structure

We observe that the efficacy of a push/preload policy depends on the utilization of network and client device resources throughout the page load process. Building off of this, the primary determinants of resource utilization are 1) browser- and page-imposed dependencies [38, 39, 59], e.g., JavaScript execution blocking HTML parsing, and 2) the duration and overhead of different page load tasks involving the network or CPU. To capture all of these factors and identify page structures that warrant similar policies, Alohamora uses the annotated dependency graphs described in §3. Recall that the structure of these dependency graphs captures inter-object dependencies and constraints on request scheduling, while the per-object annotations characterize network and CPU overheads of fetch and execution tasks.

Given these dependency graphs (or trees) for each page that we hope to accelerate for a site, Alohamora defines the distance between two page's trees T_i and T_j as the tree edit distance between them; we use the state-of-the-art APTED algorithm [50]. The cost of inserting/deleting a node is set to 1, and the cost of each change to any part of a node's label (content type, size, execution time, etc.) is set to 0.25, i.e, label alterations are equally weighted such that changes to all labels are equivalent to a node insertion/deletion. To avoid incorporating label edits that minimally impact push/preload strategies, Alohamora deems objects that have sizes or execution times within $\delta\%$ of each other as equivalent; we use $\delta = 25\%$ but find the precise value to have little impact as node insertions/deletions dominate difference values.

After computing the distances between each pair of trees, we construct a *distance matrix D* where $D_{i,j} = \text{distance}(T_i, T_j)$. With this, Alohamora can run any clustering algorithm that operates on non-Euclidean distance functions—we use agglomerative clustering [63]—to group pages that are structurally similar from a push/preload perspective. During clustering, we minimize the average distance between the pages in a cluster while permitting islands (a cluster of size 1); we sweep a range for the target number of clusters, and choose the lowest one which, if increased, does not result in a new island. From there, Alohamora only considers a single (random) page per cluster for training.

Handling page changes: Recall that origin servers track changes to their page dependency graphs and share those graphs with Alohamora's runtime server (§3). A natural question is how to determine when a change to a page's dependency graph is substantial enough to deem Alohamora's model suboptimal (for that page) and prompt a retrain? To answer this, upon receiving a graph from an origin server, Alohamora re-clusters by computing the pair-wise distances between the new graph and graphs for all pages used in training. If the new clustering results remain stable such that the new graph falls into an existing cluster, then Alohamora needs not retrain. On the other hand, if the new page forms an island, then Alohamora will automatically trigger a re-train. During re-training, Alohamora will still use its model to service pages whose graphs have not substantially changed.

Prior work has shown that page dependency graphs remain structurally similar over long time scales (e.g., weeks), with only the precise URLs changing over short periods [8, 39, 43]. Thus, we expect retraining with Alohamora to be infrequent in practice. For example, we verified that the clustering results from Figure 4 are unchanged across 2 weeks.

4.2 Evaluations

We performed case studies on 100 randomly selected sites in the Alexa top 500 US list [3]. For each site, we ran a monkey crawler [1] that generated a list of 300 URLs by performing random interactions (e.g., clicks) starting from the site's landing page. From this set of URLs, we selected 30 pages that covered the logical clusters that we perceived for the page, e.g., articles vs. home page vs. user profile pages. For each of the 30 pages, we generated the corresponding annotated dependency graph, computed the pair-wise tree edit distances to all other pages, and performed the clustering described above. The generated clusters largely matched our high-level clustering intuition, e.g., for `The Atlantic`'s website, there exists a cluster for the home page (1), articles (21), category pages (5), and user profile pages (3).

We evaluated our clustering strategy for each site as follows. We first ran a brute force search (§2.2) to find the best push/preload policy for each of the site's considered pages. We then applied each page's best policy to all of the other pages for the site, including those in the same cluster, and those in other clusters. In each case, we measured the fraction of potential push/preload benefits that a page x's best policy achieved for another page y (as compared to the improvements delivered by y's best policy). In the event that a referenced object was missing for a page, we removed the corresponding action from the policy; this was rare as policies are based on fetch order IDs, not precise URLs.

Figure 4 lists our results for one environment; we note that the trends held for the other environments in §6.1. As shown, we find that push/preload policies are able to generalize well within a cluster, but not across clusters for a given page. In particular, at the median, policies that are generated and applied to the pages within a cluster achieve 89.6% of the potential push/preload benefits; this number drops to 36.3% for

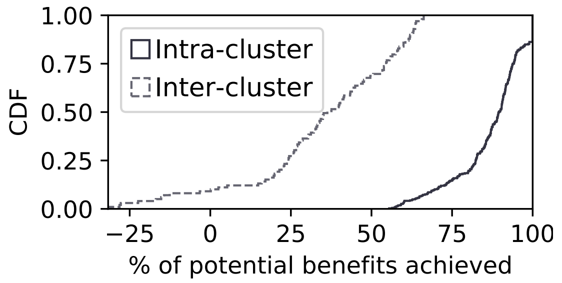


Figure 4: **Policies generalize well within (but not across) Alohamora's clusters. Results are for the {24 Mbps, 20 ms, 2× CPU slowdown, PLT} setting, and consider 100 sites, with 30 pages each. For each site, we applied each page's best policy to all other pages, and measured the % of potential benefits achieved.**

policies that are applied across clusters. §A.2 shows results for two representative pages, and also presents end-to-end results for Alohamora's policies given this clustering strategy.

5 Page Load Simulator

Even for a single page, training is impractical due to the large number of policies and environments, and the slowness of mobile page loads. To accelerate training (§3), Alohamora uses a novel page load simulator that, given an annotated dependency graph for a page, a target execution environment, and a push/preload policy as input, outputs an estimated QoE (e.g., PLT, SI) value. Unlike prior simulators (§A.1), Alohamora's is able to faithfully predict load performance (with any policy) across metrics and environments, without requiring costly profiles [68] or emulation [60] per environment—this is critical for Alohamora's training as loading pages in each environment would forego most simulation speedups. We will start by describing our simulator's operation in the context of cold cache loads, no push/preload, and PLT, and then relax those assumptions in §5.4 and §A.1. We note that Alohamora's simulator focuses on HTTP/2 page loads.

5.1 Collecting Simulator Inputs

The first step in the simulation process is to profile a load of the target page to extract information characterizing properties dictated by page composition [39] or browser dependencies [38, 59]. These properties do not describe the operating environment (which we will simulate), but instead dictate how page load tasks should share the simulated resources.

To extract such information, Alohamora records the target page with a record-and-replay tool [44], and replays the page over an unshaped local network with desktop-level CPU resources. During replay, Alohamora extracts an annotated dependency graph (Figure 5) that matches the ones used in §3 and §4.1. In particular, the graph structure captures the inter-object ordering and dependency constraints, and we add additional annotations that characterize each object's size, execution time, content type, etc. To aid simulation, we further break down an object's network and compute delays into:

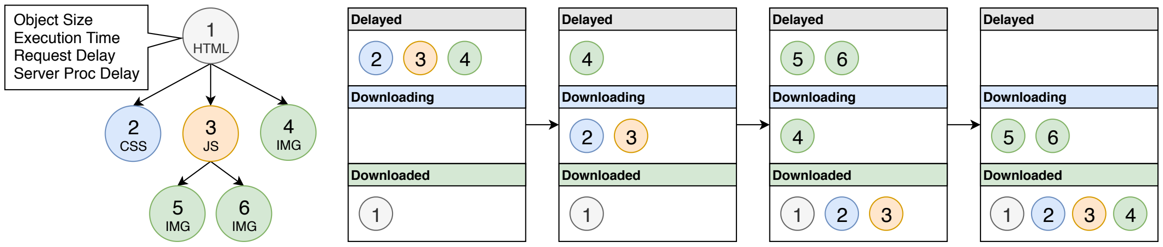


Figure 5: **Operation of Alohamora's page load simulator. The simulator operates in steps, as objects flow through these three queues, incurring blocking (e.g., connection setup, inter-object dependencies), network, and compute delays, respectively. Once an object is executed, its children are added to the top (as *delayed*) to simulate the browser discovering those dependencies.**

- **execution time:** time spent parsing, executing, or rendering the object with the well-provisioned CPU; this does not include the time to execute any referenced objects.
- **request delay:** the amount of time between when the object's parent has finished downloading, and when the object's request is issued; this embeds the parsing/execution delays of the parent, as well as any synchronous processing delays for objects referenced earlier in the parent's execution, e.g., a blocking external `<script>`.
- **server-processing delay:** server-side delay in generating and serving the response; we extract this information directly from web record-and-replay frameworks [30, 44].

In addition to this dependency graph, Alohamora's simulator also takes as input an environmental specification, listing the average network bandwidth, latency, and loss rate (Mbps, ms, %), device CPU speed (slowdown compared to profiling CPU speed), and browser cache contents.

5.2 Simulating the Execution Environment

In order to enforce the specified network and CPU values on all page load tasks, Alohamora's simulator uses a new **Request Queue** abstraction. Here, we describe how the Request Queue operates on objects passed into it; we will then describe how objects get added to the Request Queue.

At any time, the Request Queue keeps track of three types of objects using three subqueues: *delayed*, *downloading*, and *downloaded*. *Delayed* objects are those that have been discovered by the browser, but whose downloads are currently blocked, e.g., due to connection setup delays or the object's *request delay*; *downloading* objects are currently being fetched over the network; *downloaded* objects have been fetched and are currently being evaluated (or awaiting evaluation). At a high level, the Request Queue operates in steps, whereby objects flow through these queues, and once executed, children are added to the top (as *delayed*) to simulate the browser discovering those dependencies. In order to determine how long an object stays in each queue, the Request Queue models the interaction between the browser and environment, with respect to network and CPU usage.

Enforcing latency/loss overheads: In order to compute the number of round trips required to download an object, the Request Queue considers two factors. First, if the object is the first to be downloaded from a given domain, the Request Queue adds 2 RTTs to account for the TLS handshake that HTTP/2 mandates. Second, the Request Queue estimates the number of round trips required for the TCP-level data transfer by (approximately) keeping track of TCP window state for each connection (assuming cubic) and assuming that concurrent objects fairly share the window. More specifically, it assumes an initial congestion window of 10 [22], additively increases the window as bytes are downloaded, and halves the window on each idle RTO (200 ms) or probabilistic packet loss. Note that these round trip counts are computed when an object is added to the Request Queue, and are thus approximate since currently downloading objects may complete prior to the new object moving to *downloading*.

Enforcing bandwidth overheads: Across all concurrently *downloading* objects, the Request Queue must enforce an appropriate split of the specified network bandwidth. The simulator treats the bandwidth specification (either average bandwidth or a packet delivery trace [44]) as characterizing the access link, which is commonly the bottleneck in wireless networks [66] and is shared by all origins' connections. By default, the Request Queue assumes that outstanding requests fairly share the available bandwidth, thereby disregarding discrepancies in cross-connection window state.

Enforcing CPU overheads: The Request Queue modulates the *execution delay* and *request delay* for each object by multiplying by the magnitude of the CPU slowdown factor. The simulator ignores CPU core counts, and instead focuses on clock speeds, which have been shown to be the main factor affecting browser performance [10]. To support parallel iframe execution, the Request Queue subtracts out execution times from concurrently *delayed* objects across frames.

Request Queue operation: The Request Queue proceeds in discrete "steps". In each, the Request Queue inspects the lists of *downloaded*, *downloading*, and *delayed* objects, and finds the object(s) that are scheduled to either finish execution first, finish downloading first (fewest bytes remaining), or transition to *downloading* soonest, respectively. Each step is clocked by the duration t until those object events complete. After computing t, the Request Queue will subtract t from the *execution delays* of all *downloaded* objects, subtract the number of bytes that can be downloaded in t from all currently *downloading* objects, and subtract *t* from the blocking delays for all currently *delayed* objects. It will then move all *delayed* objects whose blocking delays have expired to the *downloading* queue, and mark all objects that complete *downloading* as *downloaded*. We discuss how *downloaded* objects affect subsequent resource discovery next.

5.3 Simulating Page Loads

Starting from the root node in the dependency graph (i.e., the top-level HTML), each time an object is marked as *downloaded* by the Request Queue, the simulator immediately adds all of that object's direct children as *delayed* to the Request Queue, simulating the browser's discovery of those objects. In other words, each child of the completed object is scheduled in a one-step look-ahead process, resulting in a dependency graph traversal that is breadth-first across each object's children, but not necessarily across siblings with different parents (Figure 5). Note that, after its children are added to the Request Queue, the parent object remains in the *downloaded* queue until its *execution delay* expires; in parallel, each child incurs its own *request delay* which characterizes the offset in the parent's execution until it is discovered.

This simple approach closely mimics the browser graph traversal strategy [39, 59], but with one issue: execution dependencies between an object's children. For instance, consider a simple scenario in which the top-level HTML includes two adjacent HTML `<script>` tags that reference files $S1$ and $S2$, both of which have children. Because browsers are unaware of the potential state dependencies between these two JavaScript files, upon discovering the first `<script>` tag, HTML parsing would halt and trigger a synchronous (i.e., blocking) fetch and execution of $S1$ [39]. This has several implications on dependency graph traversal, which Alohamora's simulator must account for:

- during a real load, the children of a parent may not be scheduled in a single burst. The simulator accounts for this by including an object's *request delay* (which accounts for inter-children blocking delays) in the duration that the Request Queue marks it as *delayed* (§5.2).
- even with the enforced *request delay*, it is possible for the Request Queue to mark $S2$ as *downloaded* before $S1$, e.g., if $S2$ is far smaller and the simulated network is bandwidth-constrained. This could result in cascading discrepancies in graph traversal since $S1$'s children should be handled before $S2$'s. To handle this, the simulator also exposes the Request Queue to IDs listing the object fetch orders logged in the profiled load. These IDs inherently follow the order in which browsers require, or are blocked on, specific objects. With this information, the Request Queue treats *downloaded* objects as a priority queue, signaling object completion to the graph traversal component only once the next required object (i.e., the lowest incomplete ID) is complete. Asynchronously-fetched objects are returned after their closest synchronous neighbors.

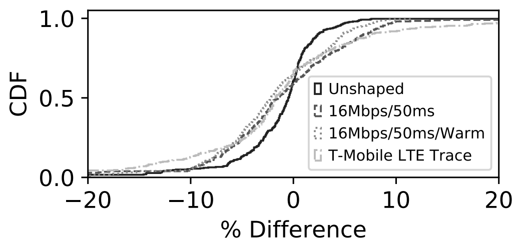


Figure 6: **Faithfulness of the Alohamora simulator's predicted PLT compared to measurements from a real browser.**

We note that, despite these strategies, the simulator's dependency graph traversal still faces potential inaccuracy in the fact that objects involving a blocking dependency, such as $S1$ and $S2$ in the above example, may download concurrently and share network resources. However, the simulator bounds the cascading effects of these inaccuracies on the page load process by ensuring that the ordering of downstream object discovery faithfully mimics that of a real browser.

Measuring PLT: As *downloaded* objects complete execution in the Request Queue, they are marked with a completion time relative to the start of the page load. PLT is the maximum object completion time [42]. In §A.1, we discuss how other metrics such as Speed Index are measured.

5.4 Simulating Push/Preload Policies

To support push/preload, when an object is being added to the Request Queue, the simulator also schedules the corresponding objects to push and preload along with that object (as per the input policy). The objects added for push/preload largely share the blocking delays of the ancestor since push/preload objects cannot begin downloading until the ancestor does. In particular, the Request Queue imposes the ancestor's *request delay*, but alters the remaining delays in two ways: 1) their server-side processing delays are preserved (and not adopted from the ancestor), and 2) preload objects incur an additional network RTT to account for the download of the ancestor's response HTTP headers (0.5 RTT) and transmission of the preloaded object's request (0.5 RTT).

Once scheduled, the key challenge is in determining how pushed/preloaded objects affect the delays from the profiled load; this delta could be positive or negative due to, e.g., bandwidth contention. To understand this, once the simulator hits a pushed/preloaded object, it determines how the object's download progress compares (or will compare) to the case when the object was not pushed/preloaded. This is done by simulating the load without that object being pushed/preloaded, and comparing the resulting delays. Note that, if the pushed/preloaded object is blocking, delays for downstream siblings are edited to reflect the observed deltas.

5.5 Evaluations

We evaluated our simulator by comparing to a real browser on two metrics: fidelity in predicted performance and overall runtime. We follow the same setup as described in §6.1.

	Median	95th Percentile
Alohamora's simulator	4.7	22
Unshaped	1347	3815
24Mbps/20ms/2x CPU	5936	16683
12Mbps/60ms/4x CPU	9631	27765

Table 1: **Per-page runtimes (ms) of Alohamora's simulator (top row) and a real browser in different execution environments.**

Fidelity: Figure 6 shows that Alohamora's simulator reports highly faithful load times compared to a real browser. For example, in an environment with no network or CPU shaping and a cold browser cache, the simulator's reported load times were within 0.4% and 4.3% of the real browser, at the median and 95th percentile, respectively. Median discrepancies marginally increase to 1.4%, 1.7%, and 2.2% as fixed-rate (16 Mbps, 50 ms link) and time-varying (T-Mobile LTE) network shaping, and caching are incrementally added; we note that the errors for all other tested environments are within 4% of these numbers. Further, the low error rates persist when evaluating push/preload policies (§A.1).

Runtime: As shown in Table 1, Alohamora's simulator evaluates page loads 3-4 orders of magnitude faster than real browsers, with the discrepancies growing as the target environment becomes more resource-constrained. §A.1 describes how simulation times vary with policy length. For context, these runtime savings enable Alohamora to *reduce the training time for a page from 10s of days to just 10s of minutes*.

6 Evaluation

6.1 Methodology

To create a reproducible test environment and cover a wide range of environments, our evaluation mainly involves emulation using the Mahimahi record-and-replay tool [44]; we present real-world experiments in §6.4. Our main corpus comprised the Alexa top 500 US landing pages [3], but we also used non-landing and less popular pages in §A.2. We recorded versions of each page at multiple times to mimic different warm cache scenarios: back to back loads, and loads separated by 4, 12, and 24 hours. Mobile-optimized (including AMP [21]) pages were used when available. Experiments used Google Chrome for Android (v72).

Our emulation evaluation considered a broad range of network bandwidths (6-48 Mbps, as well as Verizon and AT&T LTE traces [44]), latencies (0-100 ms), loss rates (0.5-5%), and client device conditions (CPU slowdowns of 1-4×, relative to a desktop with an Intel Xeon Gold 5220 CPU @ 2.20GHz). Network emulation was performed using Mahimahi [44], and CPU constraints were enforced using Chrome's Devtools Protocol [13]. Unless otherwise noted, Alohamora generated a single policy generation model per page that covered the aforementioned conditions; results for Alohamora's cross-page models based on clustering (§4) are shown in §6.6. Further, in accordance with §3, dependency graphs for inference were made just prior to the experiments.

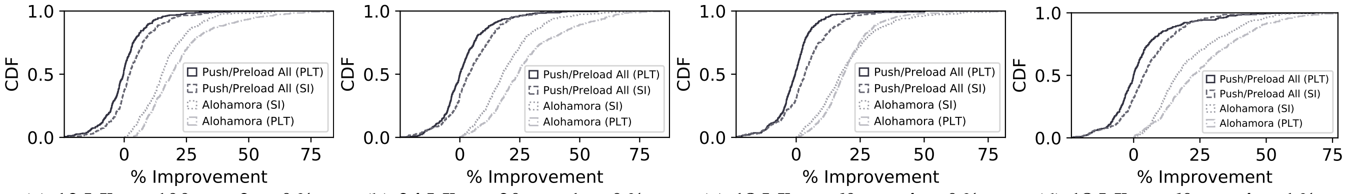
We compared Alohamora to default page loads (i.e., no push/preload) and two standard push/preload strategies: 1) the *push/preload all* strategy where, on the first incoming request, each origin pushes all static resources that it owns, and preloads all referenced static third-party resources, and 2) the *push/preload all JavaScript* strategy which operates in the same manner but only considers JavaScript objects that (unlike images) may trigger subsequent object fetches. With both strategies, push/preload order matches the order in which objects are referenced by a page. Our analysis, described in §A.2, revealed that *push/preload all* consistently delivers larger speedups than *push/preload all JavaScript*. Thus, for brevity, we only present results comparing Alohamora with *push/preload all*.

Our evaluation considers two performance metrics: page load time (PLT) measured as the time between the `navigationStart` and `onload` events, and Speed Index (SI) (measured with pwmetrics [24]) which captures the time needed to fully render the initial viewport. Due to space constraints, we present results for select settings, but note that presented trends persist in all tested scenarios. For all results, domains make push/preload decisions *independently* with Alohamora, and objects can only be pushed within a given domain, e.g., `google.com` cannot push an image belonging to `g.static` (which Google owns).

6.2 Page Load Speedups

Cold cache: Figure 7 illustrates Alohamora's ability to accelerate cold cache page loads across four representative settings. For example, in a {24 Mbps, 20 ms, 1× CPU slowdown, 0% loss} environment, median (95th percentile) PLT improvements with Alohamora were 24% (61%); the push/preload all strategy achieved only -0.2% (22%) improvements. Alohamora's benefits persist as network and CPU conditions change, although the generated policies vary: benefits are 19% (45%) when conditions degrade to {18 Mbps, 60 ms, 4× CPU slowdown, 0% loss}, 22% (57%) when 1% loss is introduced, and 14% (60%) over a time-varying Verizon LTE trace (not shown). Figure 7 also shows that Alohamora provides substantial SI benefits, ranging from 15-19% and 36-48% at the median and 95th percentile. Importantly, across all settings, Alohamora's push/preload policies *never degraded performance* compared to a default page load. This is in stark contrast to the static push/preload all policy, which slowed down 40% of pages by up to 22%.

Warm cache: Figure 8 shows that, across a wide range of warm cache browsing scenarios, Alohamora accelerates page loads compared to both a default page load and a static push/preload all strategy. For instance, for back-to-back (i.e., perfectly warm-cache) page loads, median PLT improvements are 0.9 s and 0.4 s for Alohamora and the push/preload all strategy, respectively. These relative improvements persist (12-18%) as the time between page loads increases.



(a) **12 Mbps, 100 ms, 2×, 0%** (b) **24 Mbps, 20 ms, 1×, 0%** (c) **18 Mbps, 60 ms, 4×, 0%** (d) **18 Mbps, 60 ms, 4×, 1%**

Figure 7: **Load time (PLT and SI) improvements over a default page load for a static push/preload all strategy, and Alohamora. Environments are listed as {bandwidth, latency, CPU slowdown, loss rate}. Results used cold browser caches.**

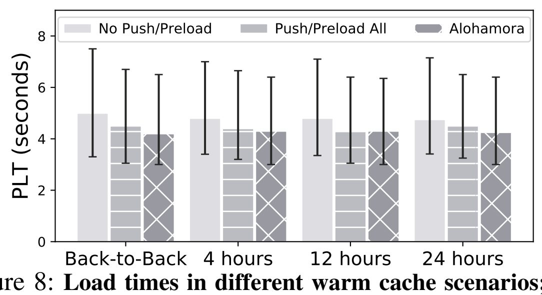
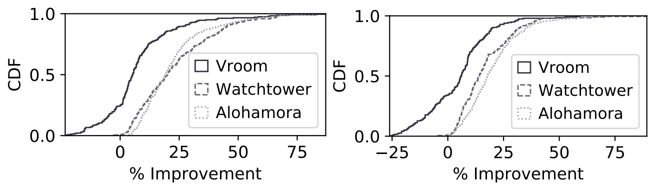


Figure 8: **Load times in different warm cache scenarios; "No push/preload" is a default page load. Bars represent medians, with errors bars spanning 25-75th percentiles. Results are for the {12 Mbps, 100 ms, 2×, 0%} setting.**



(a) 12Mbps, 100ms, 2×, 0%, PLT (b) 24 Mbps, 20 ms, 1×, 0%, SI

Figure 9: **Comparison with Vroom [54] and WatchTower [43].**

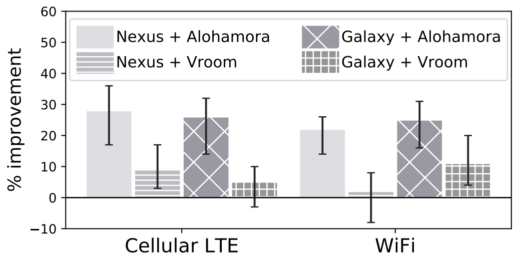


Figure 10: **PLT improvements over a default page load with Alohamora and Vroom, in the wild. Results used cold caches.**

6.3 Comparison to State-of-the-Art

We compared Alohamora with two recent mobile web accelerators, Vroom [54] and WatchTower [43]. Vroom improves upon the push/preload all policy by using a client-side scheduler to integrate priorities into the ordering of pushed/preloaded objects. In contrast, WatchTower selectively uses proxies (per origin) that fetch objects on behalf of clients using fast wired networks. Client-origin-proxy latencies were set as if proxies were run on Amazon EC2 in California, and WatchTower ran in HTTPS-sharding mode.

As shown in Figure 9, Alohamora outperforms Vroom on both PLT and SI. For example, in a {12 Mbps, 100 ms, 2× CPU slowdown, 0%, PLT} environment, benefits with Alohamora are 3.6× and 1.4× higher than Vroom's at the median and 95th percentile, respectively. The main reason for this discrepancy is that, even though Vroom adds dynamism to push/preload in the form of priority-based scheduling, Vroom remains too constrained to adapt to diverse execution environments. In particular, the set of objects to push/preload are static and match the push/preload all approach. This is partly evidenced by the fact that Vroom still harms a large fraction of page loads, e.g., 34% in the {24 Mbps, 20 ms, 1× CPU slowdown, SI} setting. In contrast, Alohamora can vary all aspects of the push/preload policy (objects, orderings, etc.) to best cater to the target setting, and *never harm performance*. Figure 9 also shows that Alohamora marginally outperforms WatchTower (0.9-1.7× more median benefits) *without* requiring per-origin proxy servers.

6.4 Real-World Experiments

We also evaluated Alohamora in the wild, using the same 500-page corpus from §6.1, live Verizon LTE and residential WiFi networks, and 2 mobile phones: a Nexus 6 (Android Nougat; 2.7 GHz quad core processor; 3 GB RAM) and a Galaxy Note 8 (Android Oreo, 2.4 GHz octa core; 6 GB RAM). To apply Alohamora's policies without relying on origin web server modifications, our setup uses an NGINX reverse proxy server [45]. The proxy was run on a c4.large Amazon EC2 instance in California, which had a median latency of 11 ms to the origin web servers in our corpus.[1]

Immediately prior to the experiment, the proxy downloaded the dependency graph for each page, and all of the static objects in the graph that are candidates for pushing. The proxy also housed Alohamora's learned model for each page. At runtime, all requests from the mobile device were forwarded to the proxy via DNS rules; even with a single proxy, browsers still opened a separate connection per origin server since connection setup decisions are based on domain name (not IP address). Upon receiving the first request per origin in a page, the proxy generated and applied a policy (for that origin), pushing cached resources and rewriting HTTP headers to reflect preload decisions. The proxy could also apply Vroom's policies or forward requests to origin servers.

As shown in Figure 10, median PLT improvements were 2.3-11× higher with Alohamora than Vroom; SI results followed the same trend, but were elided for space. Figure 10

[1]These proxy-to-origin latencies present a pessimistic setting, since Alohamora is designed to run directly on origin web servers.

	Reward	LSTM	BW	CPU	Latency	Loss
C1	66 (97)	61 (93)	49 (93)	48 (90)	57 (94)	59 (91)
C2	69 (97)	65 (95)	58 (95)	54 (94)	59 (92)	62 (92)

Table 2: **Impact of removing features/properties in Alohamora's model. Results are reported as median (95th percentile) percentage of potential PLT improvements (compared to Alohamora's full model). "Reward" considers the intuitive $-PLT$ reward function. C1 and C2 are the $\{$12 Mbps, 100 ms, 2$\times$, 0%$\}$ and $\{$24 Mbps, 20 ms, 1$\times$, 1%$\}$ settings, respectively.**

also illustrates Alohamora's ability to properly adapt to conditions in the wild: whereas Vroom harms performance for up to 43% of pages, Alohamora *always* sped up loads.

6.5 Understanding Alohamora's Benefits

Ablation study: To understand the relative impact of each of Alohamora's features and model properties, we performed an ablation study (Table 2). Our results reveal that bandwidth, latency, CPU, and loss information all play significant roles in Alohamora's ability to generate performant push/preload policies, with the removal of CPU inputs resulting in the largest median degradations (46-52%). Our results also highlight the importance of Alohamora's reward function and incorporation of LSTM. For instance, (intuitively) setting the reward to $-PLT$ leads to performance degradations of around 30% because it becomes easy for the agent to artificially inflate the observed reward by selecting policies with fewer actions, i.e., the earlier policies in an episode will be favored as the cumulative reward will be lower. Removing LSTM, on the other hand, led to degradations of $\sim$35%, largely due to the lack of a discount factor that guides the agent to avoid unnecessarily favoring longer policies (§3).

Alohamora's policies: To understand the learned insights behind Alohamora's benefits, we analyzed its generated push/preload policies. Admittedly, we observe that policy composition and the mix between push/preload varied dramatically across pages and resource settings; indeed, subtle interactions between these properties were a primary motivator for Alohamora's machine learning-based approach. However, we note the following common principles:

- In lower bandwidth settings, Alohamora either 1) reduced the policy length or cut data-intensive objects, or more commonly, 2) spread the same set of pushed/preloaded objects out across a larger set of parents in order to stagger downloads and reduce bandwidth contention.
- With slower CPUs, Alohamora's policies are careful to only push objects whose bytes could be downloaded until the next blocking JavaScript file is required; the goal is to prevent downstream CPU tasks from blocking on the network. In these cases, Alohamora's policies preloaded additional resources with the goal of having their downloads start (after the 0.5 RTT to contact the server) only after the next blocking resource was downloaded. In essence, the idea is to perfectly interleave downloads of non-blocking resources with the execution of blocking resources.
- For image-heavy sites (e.g., `pinterest.com`), Alohamora commonly excluded JavaScript/CSS files from its policies, and instead pushed/preloaded images that are rendered towards the top of the viewport, particularly in high-bandwidth settings or when SI is the target metric. The reason is that these pages have flat (not deep) dependency graphs, so blocking JavaScript files do not trigger cascaded serial network fetches; instead, image downloads have a larger blocking impact on load times.

We leave a more detailed analysis of Alohamora's generated policies, and an exploration into whether those policies could be converted into fixed general heuristics, to future work.

Unnecessary data usage: A well-documented risk with HTTP/2 push is in having servers push objects that are not needed by or already cached at the client browser [43, 54]. Alohamora avoids this issue in two ways: browser cache contents are explicitly considered during policy generation, and only resources that consistently appear in a page are considered for push/preload (§3.1). Consequently, we observe that Alohamora's policies do not waste any bandwidth, i.e., all pushed/preloaded objects are used in the targeted page load.

Training times: Training an Alohamora model for the median page in our corpus required 76 iterations and took a total of 19.4 minutes to reach convergence using an Amazon EC2 `c5.18xlarge` instance. This translates to a monetary cost of $0.62. We note that training costs are incurred offline and infrequently: Alohamora must retrain only when a new or modified page falls outside of the previous cross-page clusters, which we observe occurs on the order of weeks (§4.1).

Inference times: Alohamora's policy generation adds negligible delays to overall load times: median (95th percentile) inference times are 11 ms (40 ms), respectively.

6.6 Additional Results

We briefly summarize our remaining experiments here due to space constraints, and defer details to the §A.2.

Incremental deployment: We ran experiments to understand how benefits vary with different adoption rates. We found that benefits (unsurprisingly) increase as more domains adopt Alohamora, but simply having the top-level origin can achieve 56% of the potential median benefits.

Cross-page clustering: We performed an end-to-end evaluation of Alohamora's clustering strategy (§4) by training a model for each of the 100 sites in Figure 4, using only a single page per cluster. These models achieve 85-90% of the improvements achieved when training on pages individually.

Other pages, input errors, and energy savings: Alohamora's benefits persist (and in fact, increase) for interior pages and less popular sites, are robust to errors in network or device measurements, and yield per-page mobile device energy reductions of 16-23%.

7 Related Work

We discuss the most closely related work here, and present additional related work in §A.3.

Server push systems: Numerous studies have explored the performance of HTTP/2 (formerly SPDY), both with and without server push and preload [16, 20, 53, 54, 60, 70]. Like us, these works have found mixed performance benefits due to the subtle relationships between HTTP/2 and network characteristics, page composition, and TCP semantics. However, these prior efforts have all investigated and promoted static policies and configuration guidelines. In contrast, Alohamora leverages a data-driven approach to dynamically tune push/preload policies by explicitly factoring in both page composition and the target execution environment.

Mobile-optimized pages: Certain systems, most notably Prophecy [40], automatically rewrite web pages and return post-processed versions of objects to clients that reduce client compute and network costs. Unlike Prophecy, Alohamora does not alter page content, which has proven to be error-prone in practice [2]. Further, Alohamora can accelerate Prophecy pages which require at least one HTML file per frame, and unmodified image and style files—these are the static files which Alohamora targets for push/preload.

Proxy-based accelerators: Compression proxies [2, 47, 55, 58] compress objects in-flight between clients and servers, while remote dependency resolution proxies [43, 44, 56, 57] perform certain object fetches and computations on behalf of clients. Though performant, such acceleration proxies violate the end-to-end security guarantees of HTTPS. WatchTower [43] addresses this dilemma, but at a significant deployment cost, as each origin in a page must operate its own proxy. Alohamora avoids such security concerns by relying only on end-to-end HTTP/2 optimizations.

Dependency-aware scheduling: Klotski [8] analyzes pages offline to identify high-priority objects, and uses knowledge of network bandwidth and page structure to stream them to clients before they are needed. Klotski's dynamic prioritization hinges on global knowledge of object fetches, which proxies provide at the cost of security; in contrast, Alohamora origins operate independently and hedge against the decisions that other origins may make. Polaris [39] uses a client-side scheduler that reorders requests to minimize serial round trips without violating dependencies. However, unlike Alohamora, Polaris relies on clients to discover page resources, and thus cannot eliminate certain serial fetches.

8 Conclusion

Configuring HTTP/2 push/preload policies has proven challenging, as benefits depend on complex interactions between dynamic page, network, device, and browser properties. This paper presents Alohamora, a mobile web optimization system that dynamically generates HTTP/2 push/preload policies using Reinforcement Learning. To ensure practicality, Alohamora introduces novel techniques that drastically reduce the number of pages to consider for training, and the cost of training any one page—these benefits come without a drop in model generalizability. Across a broad range of settings, we find that Alohamora outperforms default loads and recent push systems by 19-61% and 3.6-4×, respectively.

Acknowledgements. We thank Harsha Madhyastha, Vaspol Ruamviboonsuk, and Anirudh Sivaraman for their valuable feedback on earlier drafts of this paper. We also thank our shepherd, Aruna Balasubramanian, and the anonymous NSDI reviewers for their constructive comments. This work was supported in part by NSF grant CNS-1943621.

References

[1] SeleniumHQ Browser Automation. https://selenium.dev/, 2019.

[2] V. Agababov, M. Buettner, V. Chudnovsky, M. Cogan, B. Greenstein, S. McDaniel, M. Piatek, C. Scott, M. Welsh, and B. Yin. Flywheel: Google's Data Compression Proxy for the Mobile Web. NSDI '15. USENIX, 2015.

[3] Alexa. Top Sites in the United States. http://www.alexa.com/topsites/countries/US, 2018.

[4] D. An. Find out how you stack up to new industry benchmarks for mobile page speed. https://www.thinkwithgoogle.com/marketing-resources/data-measurement/mobile-page-speed-new-industry-benchmarks/, 2018.

[5] M. Belshe, M. Thomson, and R. Peon. Hypertext transfer protocol version 2 (HTTP/2). 2015.

[6] N. Bhatti, A. Bouch, and A. Kuchinsky. Integrating User-perceived Quality into Web Server Design. World Wide Web Conference on Computer Networks : The International Journal of Computer and Telecommunications Networking, 2000.

[7] A. Bouch, A. Kuchinsky, and N. Bhatti. Quality is in the Eye of the Beholder: Meeting Users' Requirements for Internet Quality of Service. CHI, The Hague, The Netherlands, 2000. ACM.

[8] M. Butkiewicz, D. Wang, Z. Wu, H. Madhyastha, and V. Sekar. Klotski: Reprioritizing Web Content to Improve User Experience on Mobile Devices. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI. USENIX Association, 2015.

[9] G. Chrome. Chrome User Experience Report.

[10] M. Dasari, S. Vargas, A. Bhattacharya, A. Balasubramanian, S. R. Das, and M. Ferdman. Impact of Device Performance on Mobile Internet QoE. In *Proceedings of the Internet Measurement Conference 2018*, IMC '18, pages 1–7. ACM, 2018.

[11] A. Davies. HTTP/2: Discover the Performance Impacts of Effective Prioritization. https://developer.akamai.

com/blog/2019/01/31/http2-discover-performance-impacts-effective-prioritization, 2019.

[12] DeNA Co., Ltd. H2O Cache-Aware Push. https://h2o.examp1e.net/configure/http2_directives.html, 2019.

[13] G. Developers. Chrome DevTools. https://developers.google.com/web/tools/chrome-devtools/.

[14] Drupal. Drupal - Open Source CMS. https://www.drupal.org/, 2019.

[15] E. Enge. MOBILE VS. DESKTOP USAGE IN 2019. https://www.perficientdigital.com/insights/our-research/mobile-vs-desktop-usage-study, 2019.

[16] J. Erman, V. Gopalakrishnan, R. Jana, and K. K. Ramakrishnan. Towards a SPDY'Ier Mobile Web? *IEEE/ACM Trans. Netw.*, 23(6):2010–2023, Dec. 2015.

[17] D. Etherington. Mobile internet use passes desktop for the first time, study finds. https://techcrunch.com/2016/11/01/mobile-internet-use-passes-desktop-for-the-first-time-study-finds/, 2016.

[18] T. Everts and T. Kadlec. WPO stats. https://wpostats.com/, 2019.

[19] D. F. Galletta, R. Henry, S. McCoy, and P. Polak. Web Site Delays: How Tolerant are Users? Journal of the Association for Information Systems, 2004.

[20] U. Goel, M. Steiner, M. P. Wittie, M. Flack, and S. Ludin. Http/2 performance in cellular networks: Poster. In *Proceedings of the 22Nd Annual International Conference on Mobile Computing and Networking*, MobiCom '16, pages 433–434. ACM, 2016.

[21] Google. Accelerated Mobile Pages Project – AMP. https://www.ampproject.org/.

[22] S. Ha, I. Rhee, and L. Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS operating systems review*, 42(5):64–74, 2008.

[23] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. A close examination of performance and power characteristics of 4g lte networks. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, pages 225–238, New York, NY, USA, 2012. ACM.

[24] P. Irish. pwmetrics: Progressive web metrics. https://github.com/paulirish/pwmetrics, 2019.

[25] B. Jun, F. E. Bustamante, S. Y. Whang, and Z. S. Bischof. AMP up your Mobile Web Experience: Characterizing the Impact of Google's Accelerated Mobile Project. In *Proceedings of the 25th Annual International Conference on Mobile Computing and Networking*, MobiCom. ACM, 2019.

[26] E. Liang, R. Liaw, P. Moritz, R. Nishihara, R. Fox, K. Goldberg, J. E. Gonzalez, M. I. Jordan, and I. Stoica. Rllib: Abstractions for distributed reinforcement learning. *arXiv preprint arXiv:1712.09381*, 2017.

[27] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*, 2018.

[28] D. Lymberopoulos, O. Riva, K. Strauss, A. Mittal, and A. Ntoulas. PocketWeb: Instant Web Browsing for Mobile Devices. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII. ACM, 2012.

[29] H. Mao, S. B. Venkatakrishnan, M. Schwarzkopf, and M. Alizadeh. Variance reduction for reinforcement learning in input-driven environments. In *International Conference on Learning Representations*, 2019.

[30] S. Mardani, M. Singh, and R. Netravali. Fawkes: Faster Mobile Page Loads via App-Inspired Static Templating. In *Proceedings of the 17th USENIX Conference on Networked Systems Design and Implementation*, NSDI, Berkeley, CA, USA, 2020. USENIX Association.

[31] P. Meenan. HTTP/2 Prioritization. https://calendar.perfplanet.com/2018/http2-prioritization/, 2018.

[32] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Harley, T. P. Lillicrap, D. Silver, and K. Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, page 1928–1937. JMLR.org, 2016.

[33] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

[34] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[35] Monsoon Solutions Inc. Power monitor software. http://msoon.github.io/powermonitor/, 2018.

[36] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica. Ray: A Distributed Framework for Emerging AI Applications. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 561–577. USENIX Association, 2018.

[37] U. Naseer and T. Benson. Configtron: Tackling network diversity with heterogeneous configurations, 2019.

[38] J. Nejati and A. Balasubramanian. An In-depth Study of Mobile Browser Performance. In *Proceedings of the 25th International Conference on World Wide Web*, WWW '16, pages 1305–1315. International World Wide Web Conferences Steering Committee, 2016.

[39] R. Netravali, A. Goyal, J. Mickens, and H. Balakrishnan. Polaris: Faster Page Loads Using Fine-grained Dependency Tracking. In *Proceedings of the 13th*

USENIX Conference on Networked Systems Design and Implementation, NSDI, Berkeley, CA, USA, 2016. USENIX Association.

[40] R. Netravali and J. Mickens. Prophecy: Accelerating Mobile Page Loads Using Final-state Write Logs. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI, Berkeley, CA, USA, 2018. USENIX Association.

[41] R. Netravali and J. Mickens. Remote-Control Caching: Proxy-based URL Rewriting to Decrease Mobile Browsing Bandwidth. In *Proceedings of the 19th International Workshop on Mobile Computing Systems &#38; Applications*, HotMobile '18, pages 63–68. ACM, 2018.

[42] R. Netravali, V. Nathan, J. Mickens, and H. Balakrishnan. Vesper: Measuring Time-to-Interactivity for Web Pages. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI, Renton, WA, USA, 2018. USENIX Association.

[43] R. Netravali, A. Sivaraman, J. Mickens, and H. Balakrishnan. WatchTower: Fast, Secure Mobile Page Loads Using Remote Dependency Resolution. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '19, pages 430–443. ACM, 2019.

[44] R. Netravali, A. Sivaraman, K. Winstein, S. Das, A. Goyal, J. Mickens, and H. Balakrishnan. Mahimahi: Accurate Record-and-Replay for HTTP. Proceedings of ATC '15. USENIX, 2015.

[45] NGINX. NGINX Reverse Proxy. https://docs.nginx.com/nginx/admin-guide/web-server/reverse-proxy/, 2019.

[46] K. Oku and Y. Weiss. Cache Digests for HTTP/2. https://httpwg.org/http-extensions/cache-digest.html, 2019.

[47] Opera. Opera Turbo. http://www.opera.com/turbo, 2018.

[48] Optimizely. Content Management System. https://www.optimizely.com/optimization-glossary/content-management-system/, 2019.

[49] V. N. Padmanabhan and J. C. Mogul. Using Predictive Prefetching to Improve World Wide Web Latency. *SIGCOMM Comput. Commun. Rev.*, 26(3):22–36, July 1996.

[50] M. Pawlik and N. Augsten. Efficient Computation of the Tree Edit Distance. *ACM Trans. Database Syst.*, 40(1):3:1–3:40, Mar. 2015.

[51] C. Petrov. 52 Mobile vs. Desktop Usage Statistics For 2019 [Mobile's Overtaking!]. https://techjury.net/stats-about/mobile-vs-desktop-usage/, 2019.

[52] L. Ravindranath, S. Agarwal, J. Padhye, and C. Riederer. give in to procrastination and stop prefetching.

[53] S. Rosen, B. Han, S. Hao, Z. M. Mao, and F. Qian. Push or Request: An Investigation of HTTP/2 Server Push for Improving Mobile Performance. In *Proceedings of the 26th International Conference on World Wide Web*, WWW. International World Wide Web Conferences Steering Committee, 2017.

[54] V. Ruamviboonsuk, R. Netravali, M. Uluyol, and H. V. Madhyastha. Vroom: Accelerating the Mobile Web with Server-Aided Dependency Resolution. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM. ACM, 2017.

[55] S. Singh, H. V. Madhyastha, S. V. Krishnamurthy, and R. Govindan. FlexiWeb: Network-Aware Compaction for Accelerating Mobile Web Transfers. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, MobiCom. ACM, 2015.

[56] A. Sivakumar, C. Jiang, S. Nam, P. Shankaranarayanan, V. Gopalakrishnan, S. Rao, S. Sen, M. Thottethodi, and T. Vijaykumar. Scalable Whittled Proxy Execution for Low-Latency Web over Cellular Networks. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, Mobicom. ACM, 2017.

[57] A. Sivakumar, S. Puzhavakath Narayanan, V. Gopalakrishnan, S. Lee, S. Rao, and S. Sen. Parcel: Proxy assisted browsing in cellular networks for energy and latency reduction. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, pages 325–336, New York, NY, USA, 2014. ACM.

[58] J. Volpe. Nokia Xpress brings cloud-based compression to the Lumia line. Engadget. https://www.engadget.com/2012/10/03/nokia-xpress-brings-cloud-based-compression-to-the-lumia-line/, October 3, 2012.

[59] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystifying Page Load Performance with WProf. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI. USENIX Association, 2013.

[60] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. How Speedy is SPDY? In *Proceedings of NSDI*, NSDI'14, pages 387–399, Berkeley, CA, USA, 2014. USENIX Association.

[61] X. S. Wang, A. Krishnamurthy, and D. Wetherall. Speeding Up Web Page Loads with Shandian. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI. USENIX Association, 2016.

[62] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie. How Far Can Client-only Solutions Go for Mobile Browser Speed? In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12. ACM, 2012.

[63] J. H. Ward Jr. Hierarchical grouping to optimize an objective function. *Journal of the American statistical association*, 58(301):236–244, 1963.

[64] M. Wijnants, R. Marx, P. Quax, and W. Lamotte. HTTP/2 Prioritization and Its Impact on Web Performance. In *Proceedings of the 2018 World Wide Web Conference*, WWW '18, pages 1755–1764, 2018.

[65] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[66] K. Winstein, A. Sivaraman, and H. Balakrishnan. Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI. USENIX Association, 2013.

[67] WordPress. Blog Tool, Publishing Platform, and CMS – WordPress. https://wordpress.org/, 2019.

[68] K. Zarifis, M. Holland, M. Jain, E. Katz-Bassett, and R. Govindan. Modeling HTTP/2 speed from HTTP/1 traces. In *International Conference on Passive and Active Network Measurement*, pages 233–247. Springer, 2016.

[69] T. Zimmermann and O. Hohlfeld. Skip to the article Adoption, performance, and human perception of HTTP/2 Server Push. https://blog.apnic.net/2018/04/26/adoption-performance-and-human-perception-of-http-2-server-push/, 2018.

[70] T. Zimmermann, B. Wolters, O. Hohlfeld, and K. Wehrle. Is the Web ready for HTTP/2 Server Push? In *Proceedings of the 14th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT. ACM, 2018.

Length	0	1-9	10-19	20-29	30-39
Runtime	4.7 (22)	12 (73)	45 (189)	105 (348)	172 (546)

Table 3: **Median (95th percentile) simulator runtimes in milliseconds with varying push/preload policy length.**

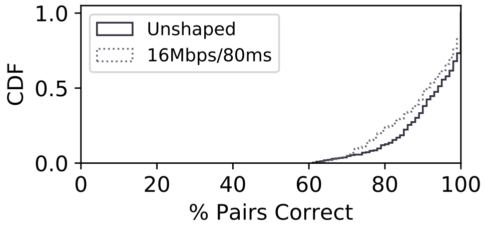


Figure 11: **Alohamora's simulator is able to correctly compare push policy pairs (in terms of relative performance).**

A Appendix

A.1 Additional Simulator Details

Additional performance metrics: We extended Alohamora's simulator to return the Speed Index and above-the-fold time [42], which is the time-instant version of Speed Index (§6.1). For this, during profiling, the simulator determines the positional information for each page component. Note that this information is dictated by page content, and can be parsed in relation to the target viewport size, i.e., we can collect positional coordinates for each page component during the profiling load, and then determine the visible content for any given viewport size [42]. With this information, the simulator identifies the set of page objects that affect the visual aspects of the target browser viewport, and characterizes performance as the time when the last node in the collected set completes its load. The simulator is also amenable to other performance metrics. For instance, to evaluate Ready Index [42], the profiling step must measure the fraction of the viewport that is visually or functionally affected by each object's execution; performance would be progressively tracked as the weighted average between time and each object's fraction.

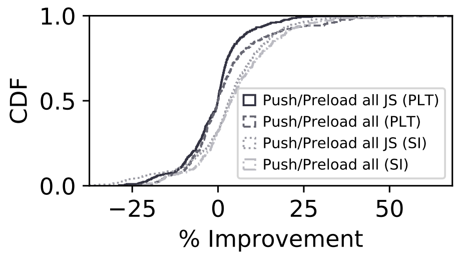
Warm cache page loads: In order to handle warm-cache browsing scenarios, the simulator takes an additional input: the list of resources that it should consider as cached, which can be computed by analyzing HTTP headers according to a desired warm cache timing, i.e., the time between the cold and warm cache page load [41]. The simulator then operates as normal, but sets the network RTTs required to fetch a cached resource, and the bytes that must be downloaded, to 0; *request delays* for downstream children of blocking resource are also updated.

Simulator speed vs. push/preload policy length: Table 3 shows that the simulator's runtime does steadily increase as the length of the push/preload policy under test grows. The reason is that Alohamora's approach to handling push/preload policies requires re-simulations of the page a number of times that is quadratic with the policy length. We note, however, that the resulting runtimes are still several orders of magnitude lower than default browsers, and Alohamora rarely requires investigation of policies longer than 20 objects (§6).

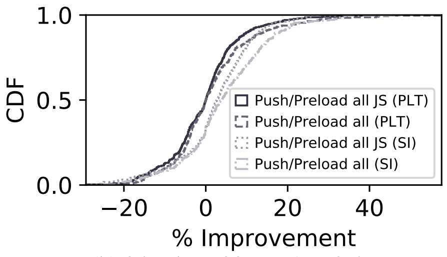
Push/preload fidelity results: §5 presented results showing the low error rates that Alohamora's simulator achieves for page loads that do not use push/preload policies. With respect to push/preload policies, the key property required for Alohamora's training is to be able to determine which of two policies results in superior performance. To evaluate the simulator's faithfulness for this, we generated 20 random push/preload policies for each page in our corpus. For each page, we counted the fraction of policy pairs for which the simulator correctly predicted the relative performance (correctness was defined by a real browser). As shown in Figure 11, the simulator correctly reported the relative comparisons across pairs 90% of the time.

Comparison to prior simulators. Several recent works have proposed web page load simulators and emulators. Here we briefly describe these prior approaches, explain their limitations for Alohamora's training scenario, and contrast them with the operation of Alohamora's simulator.

- EPLOAD [60] controls the variability in the page load process (for reproducible measurements) by profiling a page load and recording fine-grained delays between browser compute tasks (including dependencies captured by WProf [59]). EPLOAD then replays the page load process by replaying those blocking delays (via injected sleeps), but making fetches over a live (controlled) network. Thus, EPLOAD *emulates* the page load process, running live network tasks and forcing compute delays to match those from the profiled load. In contrast, Alohamora *simulates* the entire page load process, by respecting the invariant dependencies enforced by a browser and page content, as well as by modeling the interactions between the browser and underlying environment. This difference is critical to Alohamora's simulation goals: simulation enables Alohamora to evaluate push/preload policies in a few milliseconds (rather than 10s of seconds); EPLOAD's emulation approach does not shrink load times, and instead focuses on fine-grained reproducibility. Consequently, EPLOAD would not be able to accelerate training with Alohamora. Beyond this fundamental difference, Alohamora's simulator also is able to evaluate HTTP/2 push/preload policies, simulate a variety of environmental factors (CPU speeds, etc.), and evaluate performance on multiple performance metrics (e.g., Speed Index)—EPLOAD lacks these features.
- RT-H2 [68] uses profiles of HTTP/1.1 page loads to estimate the predicted performance changes for the scenario when those profiled loads were converted to using HTTP/2. The system's page load conversion model considers how HTTP/2 features (e.g., request multiplexing) affect the ordering of different page load tasks, as well as



(a) 18 Mbps, 60 ms, 4×, 1%



(b) 24 Mbps, 20 ms, 1×, 0%

Figure 12: **Performance comparisons between two existing (standard) push/preload strategies: push/preload all and push/preload all JavaScript (JS). Results are relative to default page loads (i.e., no push/preload).**

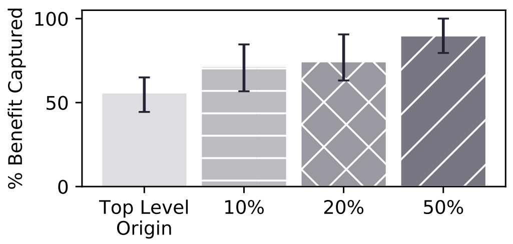


Figure 13: **Percentage of potential benefits achieved when X% of origins in each page run Alohamora. Results are for the {12 Mbps, 100 ms, 2×} setting. Bars show medians, with error bars spanning 25-75 percentiles.**

the cascading effects that those changes have on underlying network resources (e.g., TCP semantics). While similar to Alohamora's simulator with respect to approximate TCP modeling, the core limitation of RT-H2 (with respect to Alohamora's training) is that RT-H2 can only predict the performance of HTTP/2 page loads within the exact same settings as seen in the HTTP/1.1 profiles. In other words, RT-H2 cannot use its profiles to predict HTTP/2 performance outside of the profiling environments. This is problematic for Alohamora's setting, as this implies that a profile would have to be collected for every environment considering during training—this would forfeit most of the simulation benefits. Beyond this, unlike Alohamora, RT-H2 does not consider different performance metrics, compute resources, preload, or variable push policies.

A.2 Other Results

Comparison of existing push/preload strategies: In determining a competitive performance baseline to compare Alohamora with, we considered two standard push/preload (static) heuristics: *push/preload all* and *push/preload all JavaScript*. As noted in §6.1, these strategies solely differ in the set of objects that they consider for push/preload. Using the same experimental setup from Figure 7, we compared these two strategies in terms of speedups that they provide over a default page load. Figure 12 presents representative results. As shown, *push/preload all* provides roughly the same (0-1% more) speedups as *push/preload all JavaScript* at the median, and 5-17% larger speedups at the 95th percentile.

Incremental deployment: Since origins make independent push/preload decisions with Alohamora, we ran experiments to understand how Alohamora's benefits vary with different adoption rates. For each page in our corpus, we ordered the domains in the page according to the fraction of objects that they contribute. We then ran experiments where only the top X% of origins used Alohamora; origins not running Alohamora did not push/preload any objects. We also specifically considered the case where only the top-level origin deployed Alohamora. As expected, Figure 13 reveals that benefits increase as more domains adopt Alohamora. However, simply having the top-level origin can achieve 56% of the potential (i.e., with 100% adoption) median benefits.

Cross-page clustering: To this point, the presented results considered Alohamora models that were trained for a single page (across environments). In order to evaluate Alohamora's ability to train generalizable models across a site's pages, we consider the 100 sites presented in §4 (Figure 4). For each site, we trained a single Alohamora model using only a single (randomly selected) page from each cluster, and evaluated across all of the site's pages. Alohamora's cross-page models are able to achieve within 85-90% of the improvements achieved when training individually on each tested page; this slight degradation comes with the significant benefit of improved training efficiency.

Robustness to input errors: To generate push/preload policies, Alohamora's models ingest a variety of observations that collectively characterize the execution environment. While cache contents require zero approximation to collect, network and CPU measurements can be noisy and hard to report accurately. We evaluated Alohamora's ability to deliver speedups in the face of noisy inputs characterizing network and CPU speeds by considering the following errors: average bandwidth, latency, and loss errors of {1, 2, 3} Mbps, {10, 20, 30} ms, and {0.5, 1, 2}%, and CPU slowdown errors of {1, 2, 4}×. We find that Alohamora's generated policies are largely robust to such errors. For instance, median PLT improvements dropped by only 3.4%, 4.6%, and 3.9% in the {24 Mbps, 20 ms, 2× CPU slowdown} environment with errors of 2 Mbps, 20 ms, and 1%, respectively. Similarly, a CPU slowdown error of 1× resulted in only a 2.6% reduction in PLT improvements.

Additional pages: In addition to the 500-page corpus that we used for our primary experiments (§6.1), we also eval-

	Cluster 1	Cluster 2	Cluster 3
Cluster 1	88% (91%)	50% (55%)	52% (59%)
Cluster 2	57% (52%)	94% (89%)	59% (55%)
Cluster 3	61% (56%)	49% (57%)	100% (93%)

Table 4: **Evaluating Alohamora's cross-page generalization approach. Results are for the {24 Mbps, 20 ms, 2× CPU slowdown} setting, and 30 pages per site. Here we show results for two representative pages that yielded 3 clusters each: NPR (clusters with 19, 8, and 3 pages) and CNN (1, 17, and 12 pages). For each cluster, we picked a random page and found its best policy (via brute force search). We then applied that same policy to the other pages in the same cluster, and to pages in different clusters. Results list the % of possible push/preload benefits for the median page in each cluster, and are presented as CNN (NPR). Takeaway: policies generalize well within clusters (blue regions), but not across clusters (white regions).**

uated Alohamora on two additional sets of sites: 1) 100 interior pages that were collected using a monkey crawler [1] that clicked links on each landing page in our primary corpus, and 2) 100 less-popular landing pages that were randomly selected from the Alexa top 10,000 list (excluding the top 500). Using the same experimental setup described in §6.1, we find that the previously reported trends persist, and in fact, Alohamora's benefits increase. For example, in the {24 Mbps, 20 ms, 1× CPU slowdown, 0%} setting, median PLT improvements with Alohamora were 26% and 27% for the interior and less-popular corpora, respectively; for comparison, the push/preload all strategy achieved benefits of only 3% and 5%. These results are consistent with the observations in prior work [40] that interior and less popular pages are typically more complex, and involve longer serial dependency chains that can be optimized.

Energy usage: To evaluate the impact that Alohamora has on mobile device energy usage, we reused our real world experimental setup (§6.4) and connected the Nexus 6 smartphone to a Monsoon power monitor [35]. Overall, we observed that Alohamora reduces median per-page energy consumption by 23% and 16% compared to a default page load, on the LTE and WiFi networks, respectively. The savings are higher in the LTE setting primarily due to the fact that LTE radios consume more energy than WiFi hardware when active [57]—the higher network latencies on LTE networks lead to more significant load time reductions, which in turn produce larger energy savings.

Clustering: Table 4 presents evaluation results for Alohamora's clustering strategy using two representative sites; these results are a subset of those in Figure 4.

A.3 Additional Related Work

Mobile-optimized pages: Certain sites cater to mobile settings by serving pages that involve less client-side computation, fewer bytes, and fewer network fetches. For example, Google AMP [21, 25] is a recent mobile web standard that requires developers to rewrite pages using restricted forms of HTML, JavaScript, and CSS. Unlike AMP, Alohamora accelerates legacy pages without developer effort. Further, Alohamora's adaptive push/preload policies can improve the performance of AMP pages because all page resources still must traverse a client's slow mobile access link.

Prefetching: Prefetching systems predict user browsing behavior and optimistically download objects prior to user page loads [28, 49, 62]. Unfortunately, such systems have witnessed minimal adoption due to challenges in predicting what pages a user will load and when; inaccurate page and timing predictions can waste device resources or result in stale page content [52]. In contrast, Alohamora generates push/preload policies only after a user navigates to a page, and considers the environmental conditions and page properties collected in situ.

Oblique: Accelerating Page Loads Using Symbolic Execution

Ronny Ko, James Mickens
Harvard University

Blake Loring
Royal Holloway, University of London

Ravi Netravali
UCLA

Abstract

Mobile devices are often stuck behind high-latency links. Unfortunately for mobile browsers, latency (not bandwidth) is often the key influence on page load time. Proxy-based web accelerators hide last-mile latency by analyzing a page's content, and informing clients about useful objects to prefetch. However, most accelerators require content providers to divulge cleartext HTTPS data to third-party analysis servers. Acceleration systems can be installed on first-party web servers, avoiding the violation of end-to-end TLS security; however, due to the administrative overhead (and additional VM costs) associated with running an accelerator, many first-party content providers would prefer to outsource the acceleration work—if outsourcing could be secure.

In this paper, we introduce Oblique, a third-party web accelerator which enables secure outsourcing of page analysis. Oblique symbolically executes the client-side of a page load, generating a prefetch list of symbolic URLs. Each symbolic URL describes a URL that a client browser should fetch, given user-specific values for cookies, the `User-Agent` string, and other sensitive variables. Those sensitive values are never revealed to Oblique's analysis server. Instead, during a real page load, the user's browser concretizes URLs by reading sensitive local state; the browser can then prefetch the associated objects. Experiments involving real sites demonstrate that Oblique preserves TLS integrity while providing faster page loads than state-of-the-art accelerators. For popular sites, Oblique is also financially cheaper in terms of VM costs.

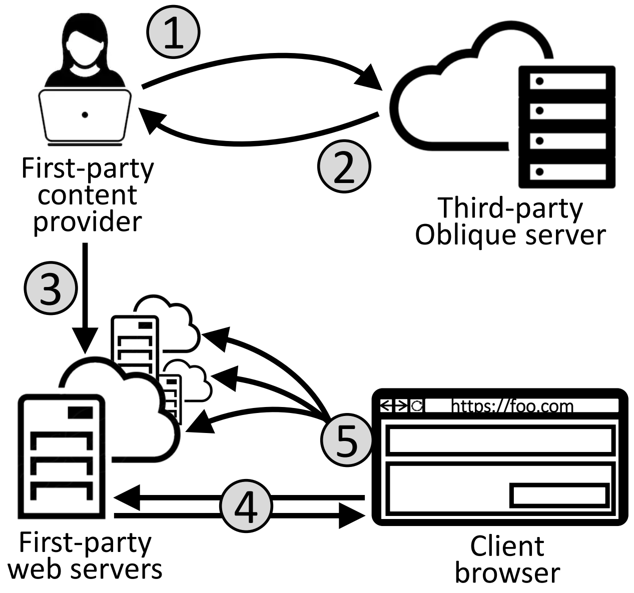


Figure 1: Overview of Oblique's design. A developer uploads page content to Oblique's analysis server (①). Oblique returns the path constraint tree for the page (②). The developer uploads the page content to web servers (③), injecting Oblique's JavaScript library into the page's HTML. Later, when a user loads the page (④), the prefetching library uses the path constraint tree to prefetch objects (⑤).

1 Introduction

Two trends are reshaping modern web services: the increasing prevalence of mobile traffic, and the continued shift from HTTP to HTTPS. 53% of all page requests now originate from smartphones [5]. 90% of those requests use HTTPS [16].

Many mobile users (particularly in emerging markets) are still stuck behind slow 3G and 4G links; even high-bandwidth 5G links often suffer from 4G latencies [11]. Unfortunately, page load times are usually determined by latency, not bandwidth [21, 24]. A variety of mobile page accelerators try to mask last-mile latency by (1) analyzing the objects (e.g., HTML and JavaScript files) that are contained by a page, and then (2) reducing the perceived fetch latencies for those objects (e.g., using server-side pushing [1,24,29,34,35] or client-side prefetching [21,29]). Unfortunately, the shift from HTTP to HTTPS has created tensions between security, performance, and the financial cost of hosting a web site. Accelerators like Silk [1] that perform remote dependency resolution [3,24,35] route client traffic through third-party proxies; these proxies are owned by browser vendors or mobile providers, and are operated for the benefit of customers. The proxies require access to cleartext HTTPS content to determine which objects to prefetch (§2). Thus, content providers that use HTTPS are faced with a dilemma: allow third-party proxies to man-in-the-middle TLS connections, or forgo the performance benefits provided by outsourced web accelerators. The former choice breaks end-to-end TLS security, and the latter option hurts page load times.

A content provider could decide to run a first-party web accelerator like Vroom [29] locally; this approach would avoid revealing cleartext HTTPS data to an untrusted middlebox. However, the content provider would incur a new financial penalty. Running a web accelerator requires extra CPU cycles and memory space beyond what is required to run a traditional web server. The content provider would have to pay for those extra VM resources.

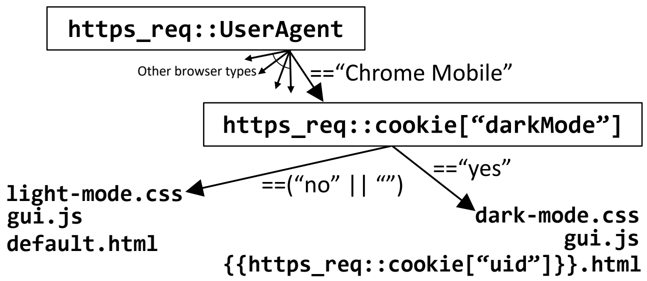


Figure 2: A simplified example of a path constraint tree. At page load time, Oblique's client-side JavaScript library traverses the tree, using load-time concrete values to trace a path to a leaf. The leaf enumerates which URLs Oblique should prefetch. Those URLs may need to be concretized with load-time values (e.g., a cookie value in this example).

In this paper, we introduce Oblique, a new system for accelerating page loads. Oblique's goal is to improve the load time reductions provided by state-of-the-art accelerators, while enabling *cheaper, more secure outsourcing* of the analyses which identify the objects that a client should prefetch. Figure 1 depicts Oblique's architecture. When a content provider creates a new page, the provider feeds the new content to a third-party Oblique server. The server performs a *symbolic page load*, exploring the possible behaviors of a web browser and a web server during the page load process. The output of the symbolic page load is a *path constraint tree*, as shown in Figure 2. Each leaf is a set of URLs that a client should prefetch, and each path from root to leaf represents symbolic constraints on the actual client-side and server-side state that is observed at the time of a real page load. During an actual page load, Oblique inspects concrete state like a client's cookie, traces a path through the constraint tree, and prefetches the relevant objects using a client-side JavaScript library.

Security: Oblique's offline analysis server does not see concrete instances of uniquely-identifying client data. For example, the server does not observe concrete values for any user's cookies or `User-Agent` string. Instead, the analysis server only sees page content that could have been fetched by any actor on the internet who can issue HTTPS requests. Later, during an actual page load, the analysis server is totally uninvolved, and receives no information about sensitive concrete values. In contrast, remote dependency resolution (RDR) forces clients to divulge cleartext HTTPS traffic, exposing cookies and other private values.

Financial cost: Oblique's offline symbolic analysis occurs when a new page version is created. The analysis cost (as measured by VM rental fees) is amortized across all client loads of the page. For popular pages, this amortized expense will be less than the aggregate per-page-load costs incurred by third-party RDR or first-party accelerators like Vroom. RDR must launch a proxy-side web browser for every client-initiated page load, whereas a Vroom-enabled web server must analyze HTML on-the-fly (§5.2).

Performance: In addition to Oblique's security and cost benefits, Oblique also loads pages up to 16% faster than RDR and Vroom (§5.2). The reason is that Oblique's symbolic analysis enables more accurate prefetching: fewer unnecessary objects are prefetched, and more objects that are truly needed are prefetched. For example, Oblique can accurately model URLs that embed random numbers; these numbers are represented symbolically in a path constraint tree, and are late-bound to concrete values at page load time (when a client generates concrete random numbers). In contrast, Vroom and RDR early-bind random numbers at analysis time, resulting in wasted prefetches, or potentially prefetchable objects being ignored (§3.3).

Summary: This paper provides three contributions:

- We describe how to symbolically evaluate client-side page load activity (§3.2), using concolic execution to model the JavaScript engine and the rendering engine. Core technical challenges involve tracking symbols that flow across the DOM interface, and preventing Oblique's third-party server from gaining insights about sensitive client-side values derived from nondeterministic functions like `Math.random()`.
- We also describe how to symbolically evaluate the *server-side* half of a page load. By analyzing both sides of a load, Oblique can generate even better prefetching hints (§3.4). The core challenges involve the complexities of HTML templating engines, and the careful orchestration needed to ensure that server-side symbols propagate to the symbolic analysis of client-side behavior. To analyze both client-side and server-side behavior, Oblique requires access to backend code and data; that state is sensitive, so Oblique should execute on first-party machines in this mode.
- We build and evaluate an Oblique prototype, and compare it to prior state-of-the-art load accelerators. When Oblique executes in third-party analysis mode, it only analyzes client-side symbols; in this mode, first-party developers can securely outsource prefetch analysis to Oblique, enjoying better security than RDR, and faster page loads than both RDR and Vroom. For popular pages, Oblique also provides lower economic costs due to better amortization of analysis overheads. If page owners are willing to run Oblique on first-party infrastructure, Oblique's client+server analysis can unlock even greater reductions in load time.

Oblique requires no changes to end-user browsers, and reduces overall page load times by up to 31%. To the best of our knowledge, Oblique is the first web accelerator that securely enables outsourced prefetch analysis for HTTPS content.

2 Background

A web page's *dependency graph* [21] captures the load-order relationships between a page's constituent objects. For example, a page's top-level HTML might contain references to a JavaScript file and an image. To load the page, a browser must fetch and evaluate both objects. Evaluating the JavaScript file might generate additional fetches, e.g., because the executed JavaScript code uses the `Fetch` API to issue new HTTP requests. Evaluating the image file causes the associated pixels to be displayed; the reception of the image data may also trigger JavaScript `onload` event handlers. Those handlers can generate more fetches. The overall page load completes when a critical subset of a page's objects have been fetched and evaluated. Different load metrics use different criteria to identify the critical subset (§5).

Web accelerators leverage knowledge of a page's dependency graph to reduce a page's load time. One popular approach is remote dependency resolution (RDR) [1,3,23,24,26,35]. An RDR system deploys a proxy server that has low-latency paths to the internet core. An end-user's browser sends each page load request to the proxy. Upon receiving such a request, the proxy launches a headless browser (i.e., a browser that lacks a GUI). The proxy-side browser loads the requested page and streams the fetched objects to the user's browser. By doing so, the proxy can partially mask the user's high last-mile latency: the page's dependency graph is resolved via the proxy's fast network links, and the bytes in each discovered object are pushed to the client as soon as the proxy receives those bytes.

RDR can reduce page load times by up to 40% [24]. Unfortunately, RDR proxies are computationally expensive to run, because web browsers (even headless ones) are complex, resource-intensive applications. A proxy can use backwards program slicing [37] to try to only execute the JavaScript code that influences calls to functions like `Fetch()`. However, slices are often inexact, and the degraded prefetching underperforms traditional RDR for 34% of pages [34].

An RDR proxy must act as a man-in-the-middle for TLS connections. Doing so allows the headless browser to parse cleartext web content and fetch the same objects that a user's browser will eventually want to fetch. However, breaking TLS's end-to-end security is obviously problematic; it allows RDR proxies to see user cookies and other sensitive HTTPS content.[1] This security violation also plagues non-RDR accelerators that perform third-party analysis of dependency graphs [6,22,40]. Cryptographic schemes that allow middlebox computation over encrypted TLS data [32] are insufficiently expressive to analyze dependency graphs; prefetch analysis requires a Turing-complete language to parse HTML and evaluate JavaScript.

Vroom [29] is a first-party web accelerator: dependency analysis runs on infrastructure belonging to the content provider. For each page, Vroom performs both offline and online analysis. The offline phase runs periodically (e.g., once an hour), using a headless browser to collect the set of URLs loaded by a page. Across multiple offline page loads, Vroom identifies a "stable set" of URLs that were fetched during each load. When a client initiates a real page load, a Vroom-modified web server parses HTML on-the-fly while streaming it to the client, extracting the embedded URLs. These embedded URLs, plus the ones found during offline analysis, comprise the set of URLs to prefetch. The web server induces the client to speculatively load these URLs via a combination of HTTP/2 push [2] and `<link>` prefetch hints [42].

Vroom's analyses run on first-party machines, so HTTPS secrets are not leaked to third parties. However, Vroom's online analysis cannot be outsourced securely: a benevolent mobile provider who wants to run Vroom on behalf of its users will have to break the HTTPS confidentiality of real user page loads. Vroom's offline phase also requires hand-tuning to deal with the heterogeneity of client browsers. For example, many sites define mobile and desktop versions of each page. A server determines which version to return by examining the `User-Agent` header in a client's HTTP request. Vroom's offline phase must be manually configured to explore the state space of all client-specific parameters like `User-Agent` and the client's screen size. Oblique's symbolic analysis allows Oblique to automatically explore this state space.

3 Design

At a high level, Oblique's offline analysis generates a *prefetch tree* for a page. The tree informs a client which HTTPS objects to prefetch in which situations. The input to the tree traversal is client-specific, potentially-sensitive information like cookie values; the output is a set of URLs. Oblique generates the tree by symbolically evaluating the client-side of a page load (§3.2). The URLs (found at the leaf nodes) are symbolic expressions that a client makes concrete by plugging in client-specific information that is never revealed to Oblique. If a page uses Node [25] (a popular server-side JavaScript framework) to generate HTML, Oblique can also symbolically evaluate server-side code (§3.4). Receiving visibility into both client and server execution allows Oblique to generate prefetch trees with more true positives and fewer false negatives: in other words, clients will fetch more useful objects and fewer unnecessary ones.

3.1 Overview of Concolic Execution

Oblique uses a particular variant of symbolic evaluation called concolic execution [14,31]. In concolic execution, a program is given a concrete set of initial inputs. The program is then executed under the observation of the concolic framework. The concolic framework assigns a "shadow" symbolic expression to each input value and to each internal program variable. An input's initial symbolic expression is only con-

[1]WatchTower [24] allows each HTTPS origin to run its own RDR proxy. This approach solves the security problem by exacerbating the computational overhead problem, since now *every* HTTPS origin must run a proxy.

strained by the limitations of the input's type. For example, a `uint32` input x might receive an initial concrete value of 2, but an initial symbolic constraint of $(0 \leq x \leq 2^{32}-1)$. During the program's execution, the assignment $y = x/2$ would result in y receiving the concrete value of 1, and the symbolic constraint $y == x/2$. When the program's execution hits a branch statement (e.g., `if(x >= 42){...}else{...}`), execution proceeds along the appropriate path, but the symbolic expressions for the branch-test variables are updated. In the running example, the `else` clause is executed because x (equal to 2) is less than 42; x's symbolic constraint is updated to become $(0 \leq x < 42)$. As the program continues execution, variables receive updated concrete values and updated shadow constraints. Eventually, the program halts or a timeout fires. The concolic framework then explores a different execution path by backtracking along the branch history and selecting a branch direction to invert. In the running example, the concolic framework might choose to explore the taken side of the branch `if(x >= 42){...}else{...}`. To do so, the framework inverts the relevant part of x's symbolic expression, generating the constraint $(42 \leq x \leq 2^{32}-1)$. The framework consults an SMT solver [7, 12] to generate a concrete value for x that satisfies the new constraints. Concolic execution then proceeds down the new branch until the program terminates or a timeout fires. This backtrack-and-explore pattern repeats until all execution paths have been discovered or (more likely) the overall time budget for concolic execution expires. For each discovered path, the framework records the *path constraints*, i.e., the symbolic constraints on all of the input variables which must be true for the path to be taken. Note that path constraints are different than the symbolic constraints on a particular variable. In our running example, the constraint on y is $y == x/2$. The path constraints for that execution path are the aggregate set of constraints placed on x and the rest of the program inputs.

3.2 Analyzing Client-side Behavior

In the context of a concolic page load, the program inputs are client-specific environmental variables. These environmental variables determine the content returned by web servers, and the execution paths taken by a page's JavaScript. For example, when a server receives the HTTP request for an HTML file, the server may examine the `User-Agent` header to determine whether to return the mobile-optimized HTML or the desktop-optimized HTML. The value for the local browser's `User-Agent` header is accessible to JavaScript via the `navigator.userAgent` variable; JavaScript code might inspect that variable to execute different code paths for different browsers. Thus, a client's user agent string is an input to the concolic page load. Table 1 enumerates the client-side inputs that Oblique considers.

Figure 3 depicts the life-cycle for a concolic page load. A distributor assigns concrete values to the inputs; the cookie value is set to an empty string, and other inputs are set to

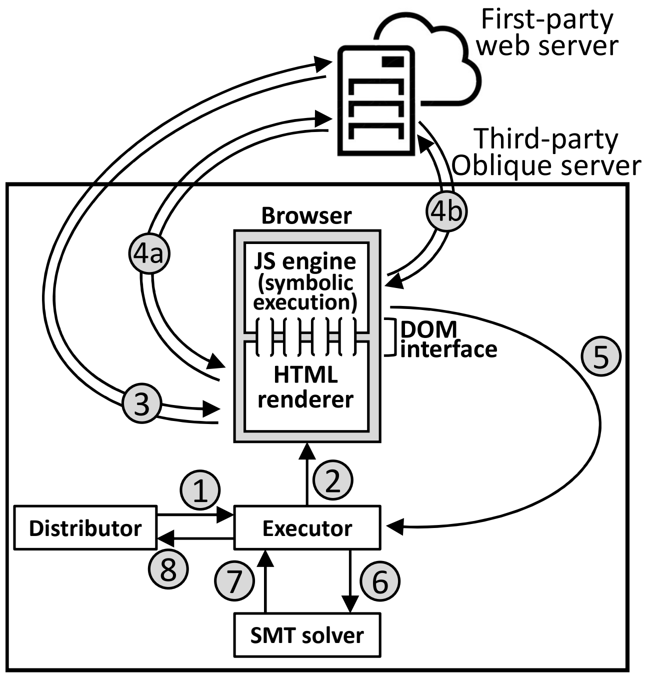


Figure 3: Overview of Oblique's approach for symbolically evaluating a client-side browser. See the mainline paper text for a description of each step.

default values for mobile Chrome. The distributor hands these values to the executor (①). The executor launches a modified web browser (②) that fetches the page's top-level HTML (③). The HTTP request for the top-level HTML uses the environmental values selected by the distributor. Note that the returned HTML will be a concrete string, not a symbolic one.

As the browser parses the HTML, the browser fetches and evaluates non-JavaScript files like CSS and images (4a). When a JavaScript file is fetched (4b), Oblique evaluates it using a modified version of the ExpoSE concolic engine [17]. As the JavaScript code executes, Oblique records the path constraints, and updates JavaScript variables with concrete values and symbolic constraints. When JavaScript code dynamically fetches an HTTP object (e.g., via `fetch(url)`), Oblique uses the *concrete* value of `url` to issue a real fetch. However, Oblique also records the symbolic constraints on `url`. These constraints, which represent a *symbolic URL*, are added to the prefetch list for the current execution path. As a contrived example, a symbolic URL might have the value "`x.com/?{{encodeURI(navigator.userAgent)}}`"; this URL would allow a web server to return different HTML to mobile clients and desktop clients.

In the prior example, the `{{}}` notation indicates a symbolic expression. The example also demonstrates how Oblique is enlightened about certain native functions like `encodeURI()`. Native functions are JavaScript-invocable methods whose implementations are provided by C++ code inside the browser.

Input name	HTTP header	JavaScript variable	Description
User agent	User-Agent	`navigator.userAgent`	The local browser type, e.g., "Mozilla/5.0 (Windows; U; Win98; en-US; rv:0.9.2) Gecko/20010725 Netscape6/6.1"
Platform	Included in User-Agent	`navigator.platform`	The local OS, e.g., "Win64"
Screen characteristics	N/A	`window.screen.*`	Information about the local display, e.g., the dimensions and pixel depth
Host	Host	`location.host`	Specifies the host and port number used by request
Referrer	Referer	`document.referrer`	The URL of the page whose link was followed to generate a request for the current page
Origin	Origin	`location.origin`	Like Referrer, but only includes the origin, omitting path information
Last modified	Last-Modified (response)	`document.lastModified`	Set by the server to indicate the last modification date for the returned resource
Cookie	Cookie (request), Set-Cookie (response)	`document.cookie`	A string containing "key=value" pairs

Table 1: Symbolic inputs to a client-side page load.

Oblique intentionally avoids the concolic execution of native code, since JavaScript-level semantics are the only ones of importance. However, to ensure that native methods correctly propagate JavaScript-level symbolic constraints, Oblique must associate a symbol policy with each native method. A policy describes how the symbolic inputs to a native method should be translated to symbolic outputs for the method. Oblique assigns policies to the most popular native methods that were seen in our test corpus (§5.1). Those methods include the ones defined by the `Math`, `String`, and `RegExp` objects. If a page invokes a native method that lacks a symbol policy, Oblique uses the concrete return value as the symbolic constraint; in other words, the native function acts as a black box that never returns symbolic data.

An HTML renderer maintains an internal data structure called the DOM tree. The DOM tree mirrors the structure of a page's HTML, with each HTML tag having a corresponding DOM node. JavaScript code uses the DOM interface to query or modify the DOM tree, e.g., to implement animations and register event handlers for GUI activity. During a symbolic page load, Oblique associates the DOM tree with a concrete HTML string and a symbolic one; the latter allows JavaScript-level symbols to flow into and out of the DOM tree via DOM methods. For example, given a reference `r` to a `<div>` tag's DOM node, JavaScript code could display the browser type using the assignment `r.innerHTML = navigator.userAgent`. A read of `r`'s parent in the DOM tree (e.g., `r.parentNode.innerHTML`) would return a string whose symbolic value contains `{{UserAgent}}`.

As the page load unfolds, Oblique logs the symbolic URLs that are passed to network APIs like `fetch()`. Oblique also interposes on the DOM interface, and logs the symbolic URLs which cross that interface. For example, suppose that JavaScript code uses the `Node.appendChild(imgNode)` method to add a new `<img>` tag to the page. Oblique would log the symbolic URL associated with the `imgNode.src` attribute; logging the URL reflects the fact that executing `Node.appendChild(imgNode)` causes the browser to fetch an image from a remote server.

Oblique's HTML renderer also logs the static, non-symbolic URLs in a page. These URLs are directly specified in a page's static HTML (e.g., `<link rel="stylesheet" href="styles.css">`) or dynamically injected by JavaScript via the DOM interface. The prefetch list for an execution path contains the static, non-symbolic URLs and the dynamic, possibly-symbolic URLs that are fetched by the path.

Oblique declares the page load to be done when the JavaScript `onload` event fires. The browser fires this event when the browser has finished the HTML parse, fetched all objects discovered by the parse, and evaluated all of those objects. As shown in Figure 3, the JavaScript engine informs the executor about the path constraints for the page load (⑤). The executor asks the SMT solver to invert a branch direction at some point along the path (⑥). Inverting the branch direction changes the symbolic constraints on the input values (§3.1). The SMT solver generates concrete input values that satisfy the new constraints (⑦). The executor returns those concrete input values to the distributor (⑧). These values represent a new test case that would cause the page to explore a different execution path.

The distributor launches many executors in parallel, running each one on a separate core. As the executors complete and return new test cases, the distributor launches new executors to explore new test cases. The distributor stops creating new executors once a predetermined time budget expires, or there are no more paths to explore. Higher budgets allow Oblique to discover more execution paths, but are more expensive in terms of VM costs. We evaluate these tradeoffs in Section 5.2.

When an executor completes its concolic page load, it logs two things: the list of symbolic URLs fetched by the page load, and the symbolic constraints on client-specific inputs like cookies. Once all executors have finished, the distributor analyses the aggregate set of executor logs to generate a tree of path constraints. Figure 2 provides an example of such a tree. Each leaf contains a set of symbolic URLs; each root-to-leaf path represents the client-specific input values which indicate that a page load will fetch the URLs at the leaf. The distributor translates the constraint tree into a JSON data structure. Finally, the distributor generates a JavaScript library that traverses the tree; at each node, the library applies regular expressions and comparison operators to the JavaScript representation of client-specific inputs (see Table 1). For example, the JavaScript code `/CriOS(54|55)/.test(navigator.userAgent)` determines whether the local browser is Chrome version 54 or 55 that runs atop iOS. Upon arriving at a leaf, the library concretizes the symbolic URLs in the leaf, and then prefetches those URLs using `XMLHttpRequest`.

Oblique sends the prefetching library (which embeds the JSON constraint tree) to the first-party web developer. The developer adds the library as an inline `<script>` tag at the beginning of the associated page's HTML. Later, when a real client browser loads the page, the library issues asynchronous prefetches, populating the local browser cache. As the browser's HTML parse examines the rest of the page and discovers references to external objects, the browser can pull those objects from its cache, avoiding wide-area fetch latencies.

3.3 Nondeterministic JavaScript Functions

JavaScript defines two categories of nondeterministic functions. Timestamp functions like `Date()` and `Performance.now()` read the system clock. Random number generators like `Math.random()` and `crypto.getRandomValues()` create pseudorandom or cryptographically-random byte sequences.

JavaScript code may consult nondeterministic functions during the construction of a dynamic URL. For example, a page might contain code like `if(Math.random() > 0.7){url="a.jpg"}else{url="b.jpg"}`. In that example, the URL embeds no symbols, but its value is controlled by the output of a nondeterministic function. Code like `url=Date() + ".jpg"` would create a URL that directly embeds the output of a nondeterministic function.

Both kinds of dynamic URLs will induce prefetch misses for RDR. The reason is that RDR uses a headless browser to generate a page's dependency graph (§2). The headless browser and the client-side browser will likely generate different nondeterministic values; thus, the two browsers will likely generate different dynamic URLs. To prevent such divergence, RDR could log the nondeterminism observed by the headless browser, and then force clients to use the logged sequence. This approach is the same one used by deterministic replay debuggers to faithfully recreate previously-observed program executions [8, 20]. However, in the context of accelerating page loads, this approach can break functionality. Clients will receive old wall-clock readings, and calculate elapsed time periods that do not accurately reflect the client's true perception of time. As a result, clients may fetch stale content or improperly calculate frame rates for animations. From the security perspective, exposing a client's `crypto.getRandomValues()` sequence to a third party is undesirable, because the client might use the sequence to derive keys or nonces.

Vroom will also suffer prefetch misses for dynamic URLs that are influenced by nondeterministic functions. Vroom's offline analysis identifies a stable set of URLs that are fetched by several different loads of a page (§2). Vroom's stable set analysis will drop URLs that only differ by a timestamp or a random number. The analysis will also drop URLs that do not directly embed nondeterminism, but are fetched via branching paths whose directions are chosen by nondeterminism.

Oblique handles these dynamic URLs without forcing clients to divulge their nondeterminism to third parties. During an offline symbolic execution, Oblique creates a unique, hidden variable for each invocation of a nondeterministic function. Oblique treats this variable as a client-specific input, akin to `document.cookie` or `User-Agent`. This approach enables Oblique to track how the outputs of nondeterministic functions influence branch decisions and the construction of dynamic URLs. For example, suppose that during symbolic execution, a page's JavaScript code invokes `Math.random()` twice, and then calls `Performance.now()`. Oblique generates the hidden variables $\mathtt{rand}_0$, $\mathtt{rand}_1$, and $\mathtt{pnow}_0$. As the symbolic page load continues, the load may generate dynamic URLs like `https://foo.com/?{{rand`$_0$`}}.js`. Oblique places these URLs in the prefetch list as normal. The symbolic execution may also branch on the values of $\mathtt{rand}_1$ and $\mathtt{pnow}_0$, just like the symbolic execution might branch on `User-Agent`. Later, during a real client-side page load, Oblique's prefetch library concretizes hidden variables before traversing the path constraint tree. In the previous example, the prefetch library would make two calls to `Math.random()`, and one call to `Performance.now()`. With the hidden variables now concretized, and with client-specific values like `User-Agent` in hand, the prefetch library can now traverse the path constraint tree and concretize all of the URLs that reside at the appropriate leaf.

The library prefetches the concretized URLs. Finally, the library dynamically patches [20] nondeterministic functions like `Math.random()` and `Performance.now()`, forcing those methods to return the values in the log of concretized hidden variables. The prefetching library is the first JavaScript code that executes in a page. Thus, as the rest of the page's JavaScript code executes, that code will craft dynamic URLs using the same nondeterministic values that Oblique used to construct prefetched URLs.

This approach may still result in unnatural calculations of elapsed time. For example, a page's normal JavaScript code may call `Performance.now()`, execute a lengthy computation, call `Performance.now()` again, and then use the elapsed time to construct a dynamic URL. If Oblique's prefetching library concretizes the two hidden variables using back-to-back calls to `Performance.now()`, the elapsed time used to influence prefetching will be much smaller than the elaspsed time used by the page's normal JavaScript. At worst, this will cause a wasted prefetch; Oblique only prefetches HTTP GET requests which (unlike POST requests) cannot induce side effects on the server. In future work, we hope to devise mechanisms to allow concolic execution to estimate wall clock time. This ability would enable Oblique to concretize hidden timestamp variables with higher fidelity.

JavaScript is an event-driven language. Thus, the execution order of event handlers (e.g., timers and GUI events) is another source of nondeterminism. Oblique does not attempt to control these sources of randomness, because the event loop only goes live after a page's HTML parse completes. This means that event-loop nondeterminism cannot affect URLs fetched during the HTML parse (e.g., via the `.src` attribute of HTML tags, or `XMLHttpRequests` issued by JavaScript). Event-loop nondeterminism *can* affect URLs fetched after the HTML parse completes.

3.4 Analyzing Server-side Behavior

When a web server receives a request for a page's top-level HTML, the server might dynamically construct the returned HTML. For example, the server might inspect the `User-Agent` string in the HTTP request, and return mobile content or desktop content as appropriate. As another example, the server might use the request's cookie to populate the HTML with user-specific URLs, e.g., corresponding to images of a user's previous purchases on an e-commerce site. Oblique's analysis from the previous sections will not detect this potential diversity of embedded URLs. The reason is that the prior analysis assumes that a page has only one version of its top-level HTML, and thus only one set of embedded JavaScript files; if this assumption is true, then the only goal of symbolic analysis is to explore branch paths in the fixed JavaScript code, identifying the dynamically-fetched URLs.

3.4.1 The Workflow

To generate more accurate prefetch lists for dynamically-generated pages, Oblique can optionally perform symbolic execution of both client-side JavaScript (that runs in a browser) and server-side JavaScript (that runs in the Node framework [25]). The end-to-end workflow looks like this:

- **Phase 1:** Oblique first performs a concolic execution of the server-side request handling code. For each test, the inputs are the HTTP request state, as well as nondeterministic function values (e.g., from Node's `crypto.randomBytes()` method). For each concolic path that is explored, Oblique logs the concrete HTML string that is generated, building a *server-side* path constraint tree. Each leaf contains a concrete HTML string, with each root-to-leaf path representing the constraints on server-side inputs that enable the concrete HTML string to be generated.
- **Phase 2:** Each concrete HTML string is fed to the client-side symbolic execution pipeline from Section 3.2. The output of that pipeline is a *client-side* path constraint tree. Each leaf contains symbolic URLs to prefetch, and each root-to-leaf path represents the symbolic constraints on client state that trigger the fetching of the leaf's URLs.
- **Phase 3:** Once Oblique has finished all of the symbolic executions (both client-side and server-side), Oblique creates a "super-constraint tree" which combines the knowledge gleaned from the individual constraint trees. The super tree maps Phase 1 path constraints on server-side inputs to the appropriate client-side path constraint tree from Phase 2; in other words, each leaf in the super tree is a client-side path constraint tree.

When a real client loads the page, the web server uses the values in the HTTP request to traverse the super tree; if the super tree branches on the return values of server-side nondeterministic function, the web server concretizes those values using the approach from Section 3.3. When the server reaches a leaf in the super tree, the server injects the leaf's prefetching library into the dynamically-constructed HTML. The subsequent construction process for the HTML is guided by the values in the HTTP request, and possibly by nondeterministic functions; those functions return the already-concretized values which guided the traversal of the super tree. When the client receives the HTML, Oblique's prefetching library executes as described in Section 3.2.

3.4.2 Templating Engines

In Phase 1, Oblique symbolically executes the server-side request handler. A developer has two options for specifying an entry point into request-handling code. First, a developer can register an `http.Server.request` event handler with Oblique. When a client request arrives, Node creates a new `http.IncomingMessage` object and invokes the handler. Oblique uses the object's HTTP headers as test inputs for concolic execution of the handler.

```
------ Server-side JavaScript ------
app.get('/', function(req, res) {
  //...examine req and derive the template parameters,
  //and then...
  res.render('template.ejs',
             {userAgent: req.headers['user-agent'],
              userID: 'alice',
              userName: 'Alice',
              nonce: random_value});
});
----------- template.ejs -----------
<html>
  <head></head>
  <body>
    <p1> Welcome to foo.com, <%= userName %>! </p1>
    <% if (userAgent.includes('Android')) { %>
      <img src='site-logo-mobile.jpg'>
    <% }  else { %>
      <img src='site-logo-desktop.jpg'>
    <% } %>
    <img id='session-<%= nonce %>'
         src='<%= userID %>.jpg'>
  </body>
</html>
```

Figure 4: An example of dynamic HTML generation using EJS templates. EJS directives are shown in bold.

The disadvantage of the prior approach is that, during the construction of dynamic HTML, a server may consult *IO-based* sources of nondeterminism. For example, the server may issue a database query, or send an RPC to an external server. Oblique does not log and replay such IO responses. Thus, the concretized Phase 1 HTML that Phase 2 consumes may be different than the dynamic HTML that is generated at the time of an actual page fetch. Such a mismatch would hurt Oblique's prefetching accuracy.

Oblique can avoid this problem if server-side code uses a template engine to generate dynamic HTML. For example, consider EJS [9], a popular template framework. EJS defines a `render(html, dict)` method. The first argument is a template string (e.g., "`<html>Hello {{name}} at {{tstamp}}`"). The second argument is a dictionary which maps template arguments to program variables (e.g., `{name: httpReq.cookie.uid, tstamp: Date.now()}`). EJS examines the template and automatically generates a JavaScript program; this program, which is executed by `render()`, performs the necessary computations to parse `dict` and emit the customized HTML. Figure 4 provides a more complex example of an EJS template.

If a developer uses EJS, then she can tell Oblique to concolically analyze the EJS-created templating JavaScript. The output of Phase 1 is now different: it consists of server-side path constraint trees that are associated with just the templating JavaScript, not the overall handler call chain. Each leaf still contains a concrete HTML string that is passed to the concolic client-side analysis in Phase 2. However, a leaf also contains the *symbolic* HTML string that was output by the Phase 1 analysis. The symbols in this string come from the `dict` argument to `render()`. In the example from Figure 4, the symbolic HTML references the `dict` arguments `userName`, `nonce`, and `userID`. Note that the `dict` argument `userAgent` does not appear in symbolic HTML; that argument is branched upon in the path conditions, but is not directly embedded in the HTML itself.

With template integration, Phase 3 is altered as well. When the web server receives a request, the server executes the request handler up to the invocation of `render()`. At that point, the server has queried any sources of nondeterminism (IO-based or otherwise); the server now possesses concrete values for all the inputs to `render()`. The server can then traverse the super tree, find the appropriate symbolic HTML, concretize it, extract the static URLs inside the concrete HTML, and then inject the appropriate prefetching library. Note that extracting static URLs from the concretized HTML is faster than a naïve top-to-bottom HTML parse, since Oblique has a priori knowledge of the offsets where the URLs will be.

3.5 Security Analysis

Oblique's security properties depend on whether symbolic analysis examines only client behavior, or both client and server behavior. Consider the scenario in which Oblique only analyzes client-side activity. In this case, Oblique only requires access to first-party content that is already publicly accessible via first-party web servers. From the perspective of a first-party web server, Oblique's third-party analysis engine looks like a normal end-user browser that issues normal HTTPS fetches. During a concolic page load, Oblique does track symbolic constraints on sensitive user values like cookies and `User-Agent` strings. However, these constraints represent a universe of possible values for sensitive variables; the constraints are insufficiently precise to allow Oblique to determine *the specific sensitive values that belong to a particular user*. For instance, we did empirically find JavaScript code which tested cookies for substrings that were user-agnostic; a common pattern was to inspect a cookie for a string representing the current date. However, JavaScript code did not contain the logical equivalent of a giant regular expression which scanned the local cookie, testing whether the cookie contained any value from an explicit list of valid user ids. Such JavaScript code does not exist because it would allow anyone to download the enclosing JavaScript file and learn all of the valid user ids for a site! Thus, Oblique's symbolic constraints on cookies are insufficient to induce concrete cookie values belonging to specific users. Similarly, if Oblique analyzes a page and determines that a possible load path will target Android users that possess a certain set of screen dimensions, this information does not allow Oblique to infer the screen dimensions and platform value for a particular user.

To analyze server-side behavior, Oblique requires access to server-side code; that code is inaccessible to public web clients. During the concolic execution of that code, Oblique might also query sensitive databases, or contact sensitive network hosts that are inaccessible to public internet hosts. Thus, if a developer wants Oblique to analyze both client-side and server-side behavior, Oblique should be run on first-party machines. Compared to Vroom (which is also a first-party accelerator), Oblique will provide faster page loads (§5.2).

3.6 Limitations

Oblique is not guaranteed to optimize every object fetch made by every page. For example, during concolic execution, a page's JavaScript may invoke unmodeled native functions, i.e., browser-provided C++ functions for which Oblique lacks a symbolic execution policy (§3.2). If concolic execution reaches one of those functions, Oblique must always treat the return value as fully concrete. Doing so will hurt path coverage if the program later branches on the value, since concrete values cannot be "inverted" to force a new branch direction to be explored.

Even if a page avoids unmodeled native functions, path coverage may suffer when symbolic path constraints are difficult to invert. If the constraint solver times out while trying to generate concrete inputs for a new path to explore, the path will not be explored. If this happens, Oblique can miss opportunities to discover prefetchable URLs. We evaluate Oblique's sensitivity to time-out parameters in Section 5.2.

During concolic execution, Oblique may trigger interactions with external entities. For example, a concolically-executed browser may issue `XMLHttpRequests` to remote servers. Oblique should only be used with pages for which such interactions are idempotent (either literally or for practical purposes). This limitation is shared by all prefetching systems which issue queries to live services to perform content analysis.

4 Implementation

To implement Oblique's symbolic analysis, we modified ExpoSE [17]. ExpoSE performs concolic execution of pure JavaScript code, but does not handle environmental interactions like network IO. We modified ExpoSE to interface with two different environmental interfaces: the Node runtime and the Electron [10] HTML renderer. Oblique uses the Node runtime when analyzing server-side code, and uses the Electron runtime when simulating client-side loads. As explained in Section 3.2, we enlightened ExpoSE to model a DOM tree symbolically, so that JavaScript-level symbolic values can flow into and out of the DOM interface. Our changes to ExpoSE were non-trivial, totalling roughly 4,300 lines of code.

Oblique's client-side prefetching library is small, containing approximately 300 lines of Javascript code. When Oblique runs in third-party mode (§3.2), web servers require no modifications (other than having to include Oblique's prefetching library at the top of each page's HTML). When Oblique runs in first-party mode (§3.4), web servers must be enlightened to traverse the super-constraint tree, concretize nondeterministic values, and interact with Oblique's templating infrastructure. To implement an Oblique-compatible web server, we created a front-end HTTP layer that sat in front of a commodity web server. The front-end layer used the nghttp2 HTTP library and the myhtml HTML parser to implement the activities described above.

5 Evaluation

In this section, we compare Oblique's performance to that of Vroom and RDR, two state-of-the-art accelerators for mobile page loads. Our evaluation primarily focuses on the variant of Oblique that only analyzes client-side behavior, since we can evaluate this variant on a large number of commercial sites. Using a corpus of 200 real pages, we find that Oblique reduces page loads by up to 31%, outperforming Vroom and RDR by up to 17% while also reducing VM costs for popular sites (§5.2). Oblique provides these advantages while also enabling secure outsourcing of prefetch analysis (§5.2). In Section 5.3, we use a site that we control to provide a case study of the benefits of analyzing both client-side behavior and server-side behavior. We demonstrate that, if first parties are willing to run Oblique, they can unlock even greater reductions in load time than what client-only analysis provides.

5.1 Methodology

Our experiments used a Galaxy S10e phone that ran Chromium v78. The browser ran atop Linux on Dex [30], a runtime that enables Samsung phones to execute traditional Linux executables; Linux on Dex made it easier for us to write testing scripts and other experimental infrastructure. We automated the initiation of page loads and the collection of load time metrics using the Browsertime [33] library. Internally, Browsertime manipulated Chrome via Selenium's WebDriver APIs [36, 43].

To test Oblique, Vroom, and RDR with real websites, we built a Mahimahi-style tool [23] to record the objects in live web pages. Afterwards, when our test phone sent an HTTP request to an Oblique web server, a Vroom web server, or an RDR proxy, the web server or proxy responded with recorded content if the request hit in the replay cache; otherwise, the web server or proxy issued a live fetch to the appropriate content server. We ran Oblique and Vroom web servers, and RDR proxies, on a Digital Ocean VM with 8 2.3 GHz cores, 16 GB of RAM, and a 2 Gbps NIC. The RDR proxy used headless Chrome [28] to load pages. Vroom's offline analysis also used headless Chrome. The online Vroom web server was a derivative of nghttp2 [38] that used MyHTML [4] and Katana [27] to parse HTML and CSS.

Our phone had an LTE connection with a round-trip time of 47 ms to our Digital Ocean VM. Our test corpus contained 200 pages from the Majestic Million [19]. We selected the 200 most popular pages for which the RTT between our phone and a page's web server was less than the the RTT between our phone and our Digital Ocean VM. This setup resulted in conservative estimates of the benefits provided by Oblique, Vroom, and RDR, relative to the baseline scenario in which our phone contacted normal web servers directly. For each combination of <page, load time metric, acceleration technique>, we loaded each page 5 times and recorded the average. By default, Oblique and Vroom pages were loaded one hour after the completion of offline analysis, but we perform sensitivity analysis on this parameter in Section 5.2.

5.2 Client-only Analysis

PLT: We first explored Oblique's performance when only the client side of a page load is analyzed. Figure 5 shows results for the page load time (PLT) metric. PLT, as measured by the time to the browser's `onload` event, captures how long a page needs to fetch and evaluate all objects referenced by a page's static HTML. Note that PLT only waits for some dynamically-generated fetches to complete. In particular, PLT waits for fetches triggered by the insertion of new DOM nodes (e.g., `document.body.appendChild(newImg)`), but not for fetches triggered directly by network APIs like `fetch(url)`. Thus, PLT underestimates the extent to which Oblique, Vroom, and RDR reduce overall fetch latencies for a page.

Figure 5a shows that, for a 47 ms RTT and a cold browser cache, Oblique provided the average page with a 24.1% reduction in PLT, relative to a baseline (i.e., non-accelerated) page load. Oblique reduced PLTs by 17.3% more than RDR, and 5.4% more than Vroom. To explore Oblique's benefits with higher RTTs; we connected the smartphone to a desktop machine via WiFi, and used netem [18] to inject additional latency along the smartphone/desktop link. As expected, Oblique's benefits improved as phone-server RTTs grew, because of the increasing value of hiding last-mile latency. For example, Figure 5c shows PLT results for an emulated RTT of 150 ms. Oblique improved the average baseline PLT by 31.4%, outperforming RDR by 16.3% and Vroom by 6.2%.

A page load's *prefetch hit rate* is the fraction of requested objects that hit in the browser cache due to a successful prefetch. As shown in Figure 6a, Oblique enjoyed better prefetch hit rates than both Vroom and RDR. Indeed, Oblique's primary advantage over Vroom was the ability to successfully prefetch dynamic URLs that embedded nondeterministic symbols (§3.3); this advantage is reflected in Figure 6b.

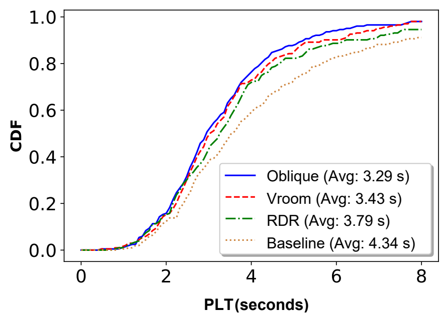
We define a page load's *wasted prefetch rate* as the fraction of prefetched objects that were never requested during the page load. Figure 6c demonstrates that RDR has a much higher percentage of wasted prefetches. The reason is that, for each client-initiated page load, RDR loads the page twice: once on the proxy, and once on the real client machine. Both client-side and server-side nondeterminism may cause the URLs fetched by the proxy's page load to be different than the URLs fetched by the client's browser. Oblique avoids this problem by handling nondeterministic URLs symbolically. In contrast, Vroom's stable-set algorithm simply filters out many nondeterministic URLs. Thus, Vroom has fewer wasted prefetches than RDR, because Vroom does not prefetch nondeterministic URLs that RDR erroneously pulls; however, as shown in Figure 6b, Vroom has a worse hit rate than Oblique due to worse handling of nondeterministic dynamic URLs.

In comparison to RDR, both Oblique and Vroom benefited from informing clients early about the URLs to prefetch. For example, Oblique discovered all of these URLs offline, and prefetched them via the first JavaScript code that executed on a page. Vroom included `<link>` preload tags at the beginning of a page's HTML, and server-pushed other objects to prefetch. In contrast, RDR streamed objects to a client as the proxy discovered those objects; the deeper a page's dependency graph was (§2), the larger the comparative advantage provided by Oblique offline discovery approach. Vroom discovered some prefetch URLs offline, and others during the online, server-side HTML parse. However, Vroom aggressively notified clients about the offline-discovered URLs using server push and `<link>` preload tags.

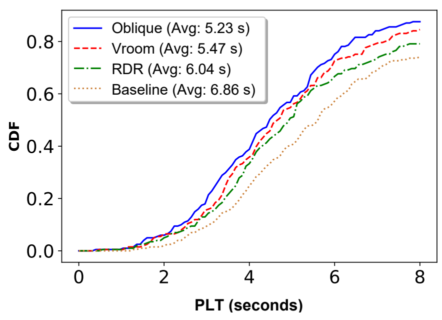
Warm caches: Figure 7a depicts PLTs for all four systems when browser caches were warm. As expected, all systems enjoyed lower PLTs. Oblique and Vroom had similar performance, but still outperformed RDR.

Speed Index: We also evaluated the ability of Oblique, Vroom, and RDR to improve a page's Speed Index [41]. Speed Index is a visual metric that represents how quickly a page's above-the-fold content is rendered. A page's Speed Index is $\int_0^{end} 1 - \frac{p(t)}{100}\,dt$, where *end* is the time of the last pixel change, and $p(t)$ is the percentage of pixels that have already received their final value; lower Speed Indices are better. The formula rewards page loads whose overall rendering time is fast (meaning that *end* values are small). Given two pages with the same *end* value, the formula rewards the page which renders more pixels earlier. Note that Speed Index ignores whether JavaScript files or below-the-fold content has arrived; thus, like PLT, Speed Index underestimates the extent to which accelerators have successfully prepositioned objects.

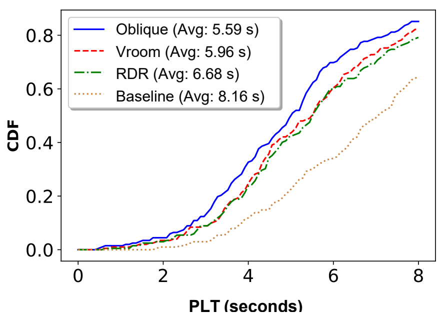
Figures 7b and 7c show Speed Index results for RTTs of 47 ms and 150 ms. The basic trend is the same one observed for PLT: Oblique has better performance than Vroom, and Vroom has better performance than RDR. However, all of the acceleration systems improve PLT more than Speed Index. For example, with cold caches and a 150 ms RTT, Oblique reduces PLT by 31.4%, but Speed Index by only 20.4%. The reason is that Speed Index only considers visual content, and only cares about the loading of JavaScript files to the extent that the evaluated code modifies a page's above-the-fold graphics. However, deep chains in a page's dependency graph are often caused by JavaScript files whose evaluation triggers the loading of additional JavaScript files [21]. All three accelerators let clients resolve those dependency chains more quickly, but this has less impact on Speed Index than PLT.



(a) 47 ms RTT

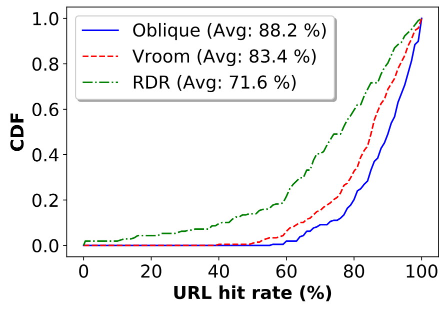


(b) 100 ms RTT

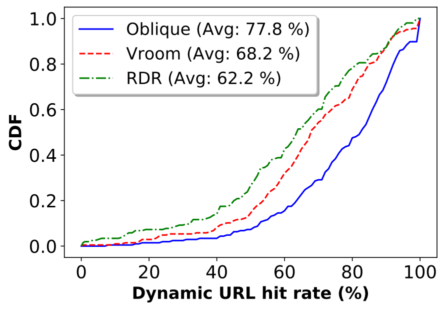


(c) 150 ms RTT

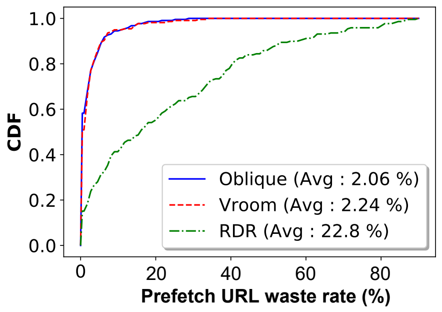
Figure 5: Cold-cache PLTs for Oblique, Vroom, RDR, and a baseline, non-accelerated browser.



(a) Prefetch hit rate (static+dynamic URLs)

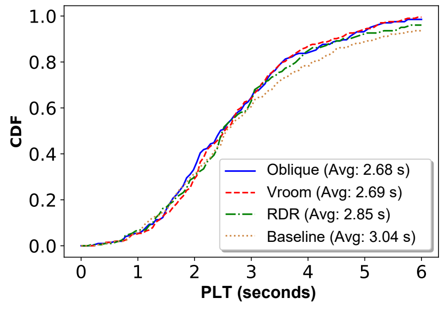


(b) Prefetch hit rate (only dynamic URLs)

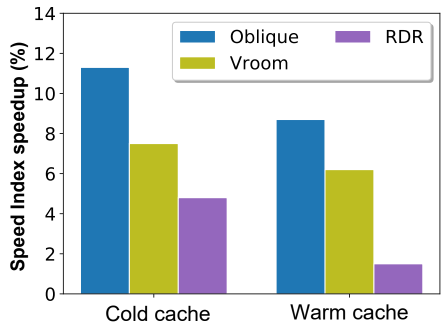


(c) Wasted prefetch rate

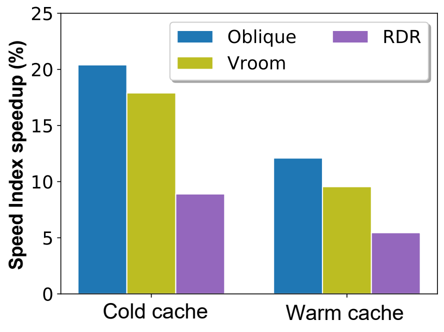
Figure 6: Prefetch efficiency for Oblique, Vroom, and RDR.



(a) PLT (warm cache, 47 ms RTT)



(b) Speed Index (47 ms RTT)



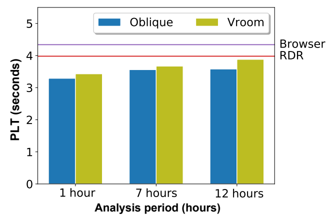
(c) Speed Index (150 ms RTT)

Figure 7: Warm-cache PLTs and Speed Indices (both cold and warm caches). Note that subfigures (b) and (c) have different y-axis scales.

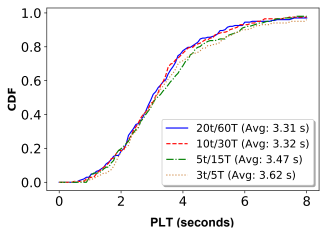
Stale analytic results: In Figures 5 and 7, the offline analyses for Oblique and Vroom occurred one hour before a page load. Figure 8a depicts average PLTs when Oblique and Vroom used analytic data from farther in the pass. Unsurprisingly, Oblique and Vroom performed better with more recent analytic data. However, for up to 12 hours of staleness, Oblique maintained its advantage over Vroom; both Oblique and Vroom also maintained their advantages over RDR and a non-accelerated browser.

Additional analysis time: Given an infinite amount of time, Oblique's offline analysis would be guaranteed to find a complete tree of path constraints; in other words, every possible concretization of client-side symbols would be covered by some root-to-leaf tree path. In practice, Oblique's symbolic analysis is constrained by two parameters: t represents the maximum execution time for a particular execution path,[2] and T represents the overall amount of time that Oblique will analyze the page. By default, Oblique uses $t = 10$ minutes and $T = 30$ minutes. If fully exploring a particular path requires more time than t, Oblique will only discover a subset of the URLs associated with the path. If discovering all paths in a page takes longer than T, Oblique will not generate a prefetch list for the undiscovered paths.

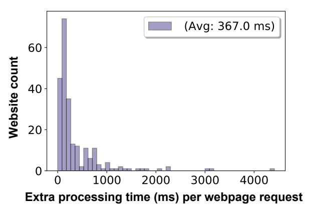
[2] A timeout occurs when the SMT solver cannot negate a branch condition in the current path (§3.1).



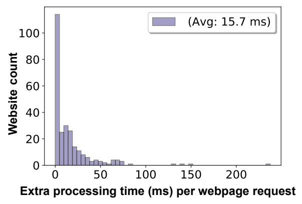
(a) PLT as a function of the staleness of the offline analytic data. These results use a 47 ms RTT and cold caches.



(b) PLT as a function of the time budget given to Oblique's offline symbolic analysis. These results use a 47 ms RTT and cold caches.



(c) RDR: Per-page-load computational time required by a proxy.



(d) Vroom: Additional per-page-load computational time required.

Figure 8: The impact of stale Oblique/Vroom analyses, and the additional computational overheads of RDR and Vroom.

Figure 8b depicts Oblique's PLT benefits for different values of t and T. In those experiments, the PLT for a page was defined as the average PLT across all discovered paths; to test the PLT for a particular discovered path, our test browser used concretized client-side symbols that triggered the path. Figure 8b shows that Oblique is basically insensitive to t values above 10 minutes and T values above 30 minutes. The reason is that, for the average page in our test corpus, only 7 minutes were needed to completely explore a path; furthermore, the median page only contained 7 execution paths.

Economic costs: For a given version of a page, Oblique performs an offline analysis once, constructs a path constraint tree, and then incurs no online costs during a real client load. In contrast, RDR must launch an RDR proxy for each client load, and Vroom must perform online HTML parsing. Figures 8c and 8d depict those per-page-load CPU costs.

A VM owner pays for a virtual CPU by the second or by the hour. Once a virtual CPU is fully loaded, any additional computation to perform will force the VM owner to rent more virtual CPU seconds. For a fully-loaded virtual CPU, Vroom requires a VM owner to pay for an additional 15.7 ms of additional compute time per page load. Thus, Oblique's offline analysis becomes cheaper than Vroom's smaller (but repeated) online costs after $T/15.7$ page loads, where T is Oblique's offline analysis time in units of milliseconds. For example, with a T of 30 minutes, Oblique becomes financially cheaper after 114,650 page loads; with a T of an hour, Oblique becomes cheaper after 229,299 loads. Since RDR imposes much heavier computational overheads than Vroom, Oblique becomes cheaper much faster—after 4,904 loads or 9,809 loads for a T of 30 minutes or a T of an hour. Importantly, these estimates assume that, when a page changes, Oblique's prior analysis is totally invalidated. We are currently investigating how Oblique can use incremental symbolic execution [13, 15] to amortize our analysis costs even more aggressively.

5.3 Oblique in First-party Mode

When Oblique runs on first-party infrastructure, Oblique can symbolically evaluate client-side and server-side behavior. However, to do this, Oblique must be able to examine backend code. We had no access to server-side code for the commercial sites in our test corpus; thus, we had to evaluate first-party Oblique on a collection of modified open-source sites that we ran ourselves. Due to space restrictions, we focus on a single case study of an open-source EJS site. In the text below, Oblique-C refers to a setup in which Oblique can only analyze client-side activity. Oblique-SC refers to a setup in which Oblique can evaluate both server and client behavior.

Gallery Viewer [39] is a site whose core functionality is displaying a rotating set of images. Each image is associated with metadata like an author, a category (e.g., "nature scenes"), and a description of the image; metadata is stored in on-disk JSON files. Users can also chat with each other in real time, and submit comments on particular images. From the perspective of Oblique, the site is interesting because of how it uses cookies and random number generators. The site assigns a unique cookie to each user. When a user requests the page's top-level HTML, the server uses the cookie to query a server-side table of user preferences. The table indicates the types of images that a user likes to view. Given those preferences, the server leverages a random number generator to select random images from the user's preferred image categories. The server inserts the associated image URLs into the dynamically-generated HTML that is returned to the user's browser.

For this particular site, no client-side symbols are relevant to prefetching. However, two kinds of server-side symbols are relevant: the cookie value in the HTTP request for the top-level HTML, and the random numbers that are used to select image URLs.

- Oblique-SC correctly prefetches all of the image URLs. During Phase 1 of analysis (§3.4), Oblique-SC symbolically evaluates the templating JavaScript, creating a symbolic HTML string. In Phase 3, i.e., during a real page load, Oblique-SC runs the server-side event handler up to the call to `render()`. At that point, Oblique-SC concretizes the symbolic HTML using the live cookie data and logged values from the random number generator. Oblique-SC then extracts the image URLs from the concretized HTML, and creates a prefetch library that downloads those URLs.

- Oblique-C lacks visibility into server-side behavior. Thus, an Oblique-C client incorrectly prefetches the URLs in the concretized HTML that was seen during offline analysis.
- Vroom correctly prefetches the image URLs; the Vroom web server identifies the URLs during the on-the-fly HTML parse.
- RDR incorrectly prefetches the image URLs. The HTML returned to the proxy's headless browser will contain different URLs than the ones in the HTML returned to the user's browser; the URLs in the first HTML file are prefetched by the client.

For a cold browser cache and a 47 ms RTT, Oblique-SC and Vroom had similar performance, with PLTs of 2.01 seconds and 2.06 seconds, respectively. Oblique-C did only slightly better than RDR (2.29 seconds versus 2.37 seconds). The non-accelerated page load required 2.76 seconds.

Oblique-SC has larger computational costs than Oblique-C; during offline analysis, more symbolic execution is required, and during an actual page load, web servers must participate in Phase 3 activity. The extent to which Oblique-SC is preferable to Oblique-C depends on whether first parties want to pay these costs, and the extent to which a site uses server-side symbols to generate HTML. However, the results from Section 5.2 demonstrate that Oblique-C alone can provide impressive reductions in page load time.

6 Conclusion

Oblique is a new system for accelerating mobile page loads. Oblique uses symbolic execution to analyze the various ways that a page load could proceed. For each potential outcome, Oblique creates a list of symbolic URLs that the corresponding page load would fetch. These URLs are concretized at the time of an actual page load, and then prefetched using Oblique's client-side JavaScript library. Oblique works on unmodified browsers, and provides faster page loads than current state-of-the-art approaches. When run in third-party mode, Oblique enables secure outsourcing of prefetch analysis while also enabling reductions in VM costs.

References

[1] Amazon. What Is Amazon Silk?, 2020. https://docs.aws.amazon.com/silk/latest/developerguide/introduction.html.

[2] M. Belshe, BitGo, R. Peon, Google, M. Thomson, and Mozilla. Hypertext Transfer Protocol Version 2 (HTTP/2), May 2015. RFC 7540. https://tools.ietf.org/html/rfc7540.

[3] D. Bhattacherjee, M. Tirmazi, and A. Singla. A Cloud-based Content Gathering Network. In *Proceedings of HotCloud*, Santa Clara, CA, July 2017.

[4] A. Borisov. Fast C/C++ HTML 5 Parser, January 8, 2020. https://github.com/lexborisov/myhtml.

[5] Broadband Search. Mobile vs. Desktop Usage (Latest 2020 Data), 2020. https://www.broadbandsearch.net/blog/mobile-desktop-internet-usage-statistics.

[6] M. Butkiewicz, D. Wang, Z. Wu, H. V. Madhyastha, and V. Sekar. Klotski: Reprioritizing Web Content to Improve User Experience on Mobile Devices. In *Proceedings of NSDI*, Oakland, CA, May 2015.

[7] L. de Moura and N. Bjorner. Z3: An Efficient SMT Solver. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*, Budapest, Hungary, April 2008.

[8] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proceedings of OSDI*, Boston, MA, December 2002.

[9] M. Eernisse. EJS: Embedded JavaScript Templating, 2020. https://ejs.co/.

[10] Electron Community. Electron Documentation, 2020. https://www.electronjs.org/docs/development/v8-development.

[11] F. Rizzato and I. Fogg. How AT&T, Sprint, T-Mobile and Verizon differ in their early 5G approach, February 20, 2020. https://www.opensignal.com/2020/02/20/how-att-sprint-t-mobile-and-verizon-differ-in-their-early-5g-approach.

[12] V. Ganesh and D. L. Dill. A Decision Procedure for Bit-Vectors and Arrays. In *Proceedings of the International Conference in Computer Aided Verification*, Berlin, Germany, July 2007.

[13] P. Godefroid. Compositional Dynamic Test Generation. *ACM SIGPLAN Notices*, 42, January 2007.

[14] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proceedings of PLDI*, Chicago, IL, June 2005.

[15] P. Godefroid, S. K. Lahiri, and C. Rubio-Gonzalez. Statically Validating Must Summaries for Incremental Compositional Dynamic Test Generation. In *Proceedings of the International Static Analysis Symposium*, Venice, Italy, September 2011.

[16] Google. Google Transparency Report: HTTPS encryption on the web, 2020. https://transparencyreport.google.com/https/overview?hl=en.

[17] B. Loring, D. Mitchell, and J. Kinder. ExpoSE: Practical Symbolic Execution of Standalone JavaScript. In *Proceedings of SPIN*, Santa Barbara, CA, July 2017.

[18] F. Ludovici and H. P. Pfeifer. NetEm - Network Emulator. http://man7.org/linux/man-pages/man8/tc-netem.8.html.

[19] Majestic. The Majestic Million: The million domains we find with the most referring subnets, 2020. https://majestic.com/reports/majestic-million.

[20] J. Mickens, J. Elson, and J. Howell. Mugshot: Deterministic Capture and Replay for JavaScript Applications. In *Proceedings of NSDI*, San Jose, CA, April 2010.

[21] R. Netravali, A. Goyal, J. Mickens, and H. Balakrishnan. Polaris: Faster Page Loads Using Fine-grained Dependency Tracking. In *Proceedings of NSDI*, Santa Clara, CA, March 2016.

[22] R. Netravali and J. Mickens. Prophecy: Accelerating Mobile Page Loads Using Final-State Write Logs. In *Proceedings of NSDI*, Renton, WA, USA, April 2018.

[23] R. Netravali, A. Sivaraman, S. Das, A. Goyal, K. Winstein, J. Mickens, and H. Balakrishnan. Mahimahi: Accurate Record-and-Replay for HTTP. In *Proceedings of ATC*, Santa Clara, CA, July 2015.

[24] R. Netravali, A. Sivaraman, J. Mickens, and H. Balakrishnan. WatchTower: Fast, Secure Mobile Page Loads Using Remote Dependency Resolution. In *Proceedings of Mobisys*, Seoul, South Korea, June 2019.

[25] OpenJS Foundation. Node.js Homepage, 2020. https://nodejs.org/en/.

[26] Opera Norway. Opera Mini, 2020. https://www.opera.com/mobile/mini/android.

[27] QFish. A CSS Parsing Library in Pure C99, 2020. https://github.com/hackers-painters/katana-parser.

[28] J. Ribeiro. Chrome Headless, 2020. Docker Hub. https://hub.docker.com/r/justinribeiro/chrome-headless.

[29] V. Ruamviboonsuk, R. Netravali, M. Uluyol, and H. V. Madhyastha. Vroom: Accelerating the Mobile Web with Server-Aided Dependency Resolution. In *Proceedings of SIGCOMM*, Los Angeles, CA, August 2017.

[30] Samsung. Web Development on a Phone. Updated for Linux on DeX., 2020. https://webview.linuxondex.com/.

[31] K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of ESEC/FSE*, Lisbon, Portugal, September 2005.

[32] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy. BlindBox: Deep Packet Inspection over Encrypted Traffic. In *Proceedings of SIGCOMM*, London, United Kingdom, 2015.

[33] Sitespeed.io. Browsertime: Your browser, your page, your scripts, April 15, 2020. https://github.com/sitespeedio/browsertime.

[34] A. Sivakumar, C. Jiang, Y. S. Nam, S. Puzhavakath Narayanan, V. Gopalakrishnan, S. G. Rao, S. Sen, M. Thottethodi, and T. N. Vijaykumar. Nutshell: Scalable whittled proxy execution for low-latency web over cellular networks. In *Proceedings of Mobicom*, Snowbird, Utah, October 2017.

[35] A. Sivakumar, S. Puzhavakath Narayanan, V. Gopalakrishnan, S. Lee, S. Rao, and S. Sen. PARCEL: Proxy Assisted BRowsing in Cellular Networks for Energy and Latency Reduction. In *Proceedings of CoNEXT*, Sydney, Australia, December 2014.

[36] Software Freedom Conservancy. SeleniumHQ: Browser Automation, 2020. https://www.selenium.dev/.

[37] F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[38] T. Tsujikawa. Nghttp2 Proxy, 2020. https://nghttp2.org.

[39] R. Villalobos. Building a Website with Node.js and Express.js, 2018. https://github.com/planetoftheweb/expressjs.

[40] X. S. Wang, A. Krishnamurthy, and D. Wetherall. Speeding Up Web Page Loads with Shandian. In *Proceedings of NSDI*, Santa Clara, CA, March 2016.

[41] WebPageTest.org. Documentation: Speed Index, 2020. https://sites.google.com/a/webpagetest.org/docs/using-webpagetest/metrics/speed-index.

[42] World Wide Web Consortium (W3C). Resource Hints, July 2, 2019. W3C Working Draft. https://www.w3.org/TR/resource-hints.

[43] World Wide Web Consortium (W3C). WebDriver, March 27, 2020. W3C Working Draft. https://www.w3.org/TR/webdriver/.

SENSEI: Aligning Video Streaming Quality with Dynamic User Sensitivity

Xu Zhang
University of Chicago

Yiyang Ou
University of Chicago

Siddhartha Sen
Microsoft Research

Junchen Jiang
University of Chicago

Abstract

This paper aims to improve video streaming by leveraging a simple observation—users are more sensitive to low quality in certain parts of a video than in others. For instance, rebuffering during key moments of a sports video (*e.g.,* before a goal is scored) is more annoying than rebuffering during normal gameplay. Such *content-dependent* dynamic quality sensitivity, however, is rarely captured by current approaches, which predict QoE (quality-of-experience) using one-size-fits-all heuristics that are too simplistic to understand the nuances of diverse video content.

The problem is that none of these approaches know the true dynamic quality sensitivity of a video they have never seen before. Therefore, instead of proposing yet another heuristic, we take a different approach: we run a *separate crowdsourcing experiment for each video* to derive the quality sensitivity of users at different parts of the video. Of course, the cost of doing this at scale can be prohibitive, but we show that careful experiment design combined with a suite of pruning techniques can make the cost negligible for content providers. For example with a budget of just \$31.4/minute video, we can predict QoE 37.1% more accurately than recent QoE models.

Our ability to accurately profile time-varying user sensitivity inspires a new approach to video streaming—*dynamically aligning higher (lower) quality with higher (lower) sensitivity periods*. We present a new video streaming system called SENSEI that profiles and incorporates dynamic quality sensitivity into existing quality adaptation algorithms. We apply SENSEI to two state-of-the-art adaptation algorithms, one rule-based and one based on deep reinforcement learning. SENSEI can take seemingly unusual actions, *e.g.,* lowering quality even when bandwidth is sufficient to prepare for higher quality sensitivity in the near future. Compared to state-of-the-art approaches, SENSEI improves QoE by 15.1% or achieves the same QoE with 26.8% less bandwidth on average.

1 Introduction

An inflection point in Internet video traffic is afoot, driven by more ultra-high resolution videos, more large-screen devices, and ever-lower user patience for low quality [2, 10]. At the same time, the video streaming industry, after several decades of evolution, is seeing diminishing improvements: recent adaptive bitrate (ABR) algorithms (*e.g.,* [45, 56, 83]) achieve near-optimal balance between bitrate and rebuffering events, and recent video codecs (*e.g.,* [54, 72]) improve encoding efficiency but require an order of magnitude more computing power than their predecessors. The confluence of these trends means that the Internet may soon be overwhelmed by online video traffic,[1] and new ways are needed to attain fundamentally better tradeoffs between *bandwidth usage* and user-perceived *QoE* (quality of experience).

We argue that a key limiting factor is the conventional wisdom that users care about quality in the same way throughout a video, so video quality should be optimized using the same standard *everywhere* in a video. This means that lower quality—due to rebuffering, low visual quality, or quality switches—should be avoided identically from the beginning to the end. We argue that this assumption is not accurate. In sports videos (*e.g.,* the one in Figure 1), a rebuffering event that coincides with scoring tends to inflict a more negative impact on user experience than rebuffering during normal gameplay. But there are also sports videos where scoring is not the most quality-sensitive part. Thus, user quality sensitivity *varies with the video content dynamically over time*.

Unfortunately, both the literature on ABR algorithms and the literature on QoE modeling adopt the conventional wisdom. Most ABR algorithms completely ignore the content of each video chunk: they focus on balancing high bitrates and low rebuffering times, and thus consider only the size and download speed of the chunks. Traditional ways of modeling QoE are also agnostic to the substance of videos, although recent QoE models—*e.g.,* PSNR [38], SSIM [80], VMAF [11], and deep-learning models [33, 48]—try to find frames that users are more sensitive to by studying the structure of pixels and motions to gauge their saliency. These heuristics seek to generalize across *all videos* and thus resort to generic measures (like pixel-level differences), but it is unclear if any heuristic can capture the diverse and dynamic influence a video's content can have on users' sensitivity to quality.

For example, models like LSTM-QoE [33] assume that users are more sensitive to rebuffering events in more "dy-

[1] This is vividly illustrated by the recent actions taken by YouTube and Netflix (and many others) to lower video quality in order to save ISPs from collapsing as more people stay at home and binge watch online videos [12].

namic" scenes. In sports videos, however, non-essential content like ads and quick scans of the players can be highly dynamic, but users may care less about quality during those moments. In the video in Figure 1, LSTM-QoE considers normal gameplay to be the most dynamic part, but the most quality-sensitive part of the video according to our user study is the goal. A key insight is that the impact of the substance of a video on users' sensitivity to quality cannot be fully explained by pixel-level patterns or cross-frame motions. Some recent work tries to predict user's dynamic sensitivity, but they either need access to users' viewing history [35] or use off-the-shelf computer-vision saliency models [34] whose predictions have little correlation with quality sensitivity on videos they have never seen before (§2.3 elaborates on this).

The dynamic nature of quality sensitivity suggests a new avenue for improvement. One can achieve *higher QoE with the same bandwidth* by carefully lowering the current quality in order to save bandwidth and allow higher quality when users become more sensitive. Similarly, one can attain *similar QoE with less bandwidth* by judiciously lowering the quality when quality sensitivity is indeed low. In short, we seek to *align higher (lower) quality of video chunks with higher (lower) quality sensitivity of users*.

We present SENSEI, a video streaming system that incorporates dynamic quality sensitivity into its QoE model and video quality adaptation. SENSEI addresses two key challenges.

Challenge 1: How do we profile the unique dynamic quality sensitivity of each video in an accurate and scalable manner?

Crowdsourcing the true quality sensitivity per video: Instead of proposing another heuristic, SENSEI takes a different approach. We run a *separate crowdsourcing experiment for each video* to derive the quality sensitivity of users at different parts of the video. Specifically, we elicit quality ratings directly from real users (obtaining a "ground truth" of their QoE) for multiple renderings of the same video, where each rendering includes a quality degradation in some part of the video. SENSEI automates and scales this process out using a public crowdsourcing platform (Amazon MTurk), which provides a large pool of raters, while using pruning techniques to reduce the number of rendered videos that need to be rated. We then use these ratings to estimate a *weight* for each video chunk that encodes its quality sensitivity, independent of the quality of other chunks. While crowdsourcing has previously been used to model QoE, SENSEI is to our knowledge the first to scale it to *per-video* QoE modeling.

Challenge 2: How do we incorporate dynamic quality sensitivity into a video streaming system to enable new decisions? Today's video players are designed to be "greedy": they pick a bitrate that maximizes the quality of the next chunk while avoiding rebuffering events. But in order to utilize dynamic quality sensitivity, a player must "schedule" bitrate choices over *multiple* future chunks, each having a potentially different quality sensitivity. This means that some well-established behaviors of video players, *e.g.,* only rebuffer when the buffer is empty, may need to be revisited.

Refactoring ABR logic to align with dynamic quality sensitivity: SENSEI works within the popular DASH framework. It integrates the aforementioned per-chunk weights into existing ABR algorithms to leverage the dynamic quality sensitivity of upcoming video chunks when making quality adaptation decisions. The per-chunk weights enable new adaptation actions that "borrow bandwidth" from low-sensitivity chunks and give them to high-sensitivity chunks. For example, SENSEI may lower the bitrate even when bandwidth is sufficient, or initiate a rebuffering event with a non-empty buffer, to afford higher bitrates when quality sensitivity becomes higher. We apply SENSEI to two state-of-the-art ABR algorithms: Fugu [83], a more traditional rule-based algorithm, and Pensieve [56], a deep reinforcement learning-based algorithm.

Using its scalable crowdsourcing approach, SENSEI can predict QoE more accurately than state-of-the-art QoE models. For example, with a budget of just \$31.4/minute video, SENSEI achieves 55% less QoE prediction error than existing models. Compared to state-of-the-art ABR algorithms, SENSEI improves QoE on average by 15.1% or achieves the same QoE with 26.8% less bandwidth across various video genres.

Contributions and roadmap: Our key contributions are:

- A measurement study revealing substantial temporal variability in users' quality sensitivity and its potential for improving video streaming QoE and bandwidth usage (§2).
- The design and implementation of SENSEI, including: 1) a scalable crowdsourcing solution to profiling the true dynamic quality sensitivity of each video (§4,§5),[2] and 2) a new ABR algorithm that incorporates dynamic user sensitivity into existing algorithms and frameworks (§6).

2 Motivation

We begin by showing that existing approaches to modeling video streaming QoE fail to accurately capture the true user-perceived QoE (2.1). We then present user studies that reveal a missing piece in today's QoE modeling: users' quality sensitivity varies dynamically throughout a video (§2.2), and this dynamic quality sensitivity is hard to capture using prior heuristics or vision models (§2.3). However, by incorporating dynamic quality sensitivity into existing ABR algorithms, we can significantly improve QoE and save bandwidth (§2.4).

2.1 Prior QoE modeling and their limitations

QoE models are crucial to modern video streaming systems. A QoE model takes a streamed video as input and returns a predicted QoE as output. When streaming a video, the video player optimizes QoE by adapting the bitrate of each video chunk to the available bandwidth. QoE is often measured by the mean opinion score (MOS) assigned by a group of users to the quality of a video.[3]

[2]Our study was IRB-approved (IRB18-1851).

[3]Our methodology extends to other QoE metrics as well.

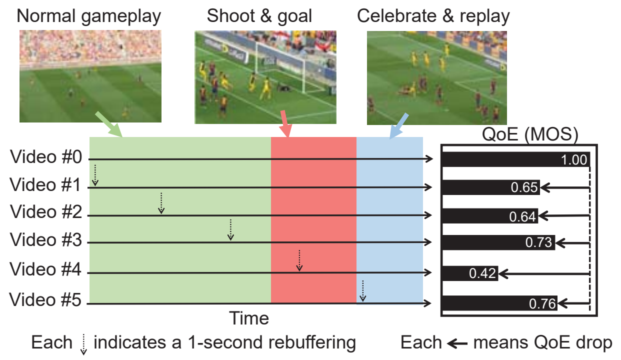


Figure 1: *Example of dynamic quality sensitivity. Users are asked to rate the quality (on a scale of 1 to 5) of different renderings of a source video (`Soccer1`), where a 1-second rebuffering event occurs at a different place in each rendering. We observe substantial differences in the QoE impact (measured by mean opinion score, or MOS) across the renderings.*

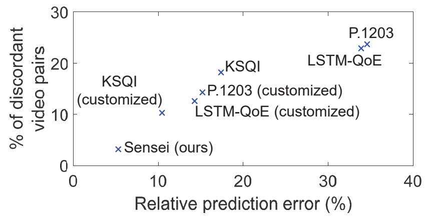


Figure 2: *Existing QoE models exhibit substantial QoE prediction errors (x-axis), which cause them to frequently mispredict the relative QoE ranking between two ABR algorithms on the same video, i.e., a discordant pair (y-axis).*

Quality metrics: Today's QoE models consider two aspects.

- *Pixel-based visual quality* tries to capture the impact of visual distortion on QoE. These metrics, such as PSNR and VMAF, are based on pixel/motion-based patterns [11, 38, 48, 63, 68, 79, 80] and recently on visual attention [27, 44, 86].
- *Streaming quality incidents* during the streaming process can negatively impact user experience, such as rebuffering, low bitrate, and bitrate switches. Their impact is modeled by metrics, such as rebuffering ratio, average bitrate, and frequency of bitrate switches during a video (*e.g.,* [18, 28]).

Some work also considers contextual factors (*e.g.,* viewer's emotion, acoustic conditions, etc.), but these are orthogonal to our focus on the video's content.

QoE models: Recent QoE models combine both pixel-based visual quality metrics and quality-incident metrics for more accurate QoE prediction. We consider three such QoE models: KSQI [29], P.1203 [66], and LSTM-QoE [33], which were proposed within the past two years and have open-source implementations. KSQI combines VMAF, rebuffering ratio, and quality switches in a linear regression model. P.1203 combines QP (quantization parameter) and quality incident metrics in a random-forest model. Most recently, LSTM-QoE takes STRRED [68] and individual quality incidents as input to a long short-term memory (LSTM) neural network designed to capture the "memory effect" of human perception of past quality incidents. (We discuss related work in §8.)

User study methodology: We evaluate these QoE models (KSQI, P.1203, LSTM-QoE) on 16 source videos randomly selected from four public datasets [21, 30, 36, 78], covering a wide range of content genres (sports, scenic, movies, etc.). These videos are streamed using one of three ABR algorithms: Fugu [83], Pensieve [56], and BBA [45], over 7 throughput traces randomly selected from real-world cellular networks [5, 65], with bandwidths ranging from 200Kbps to 6Mbps. §7.1 and Appendix A provide more details on the videos and network traces. This creates 336 (16×7×3) rendered videos. To obtain the ground truth QoE of each rendered video, we elicit QoE ratings from crowdsourced workers on Amazon MTurk [1]. We obtain at least 30 ratings from different MTurkers and use the MOS over these ratings as the true QoE of the rendered video. §4 and §5 describe our crowdsourcing methodology in detail.

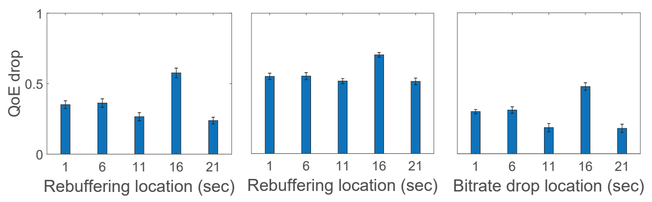
QoE prediction accuracy: Given the ground-truth QoE, we evaluate the three QoE models both with their pre-trained parameters and after customizing (retraining) them on 315 of the rendered videos selected at random. All models are tested on the remaining 21 videos; we scale their output range and the true QoE to $[0,1]$. The x-axis of Figure 2 shows the mean relative prediction error of each QoE model on the test set; relative prediction error is defined as $|Q_{predict} - Q_{true}|/Q_{true}$, where $Q_{predict}$ and Q_{true} are the predicted and true QoE of the video. We see that these errors are nontrivial; even the most accurate QoE model has over 10.4% error on average.

We also examine whether these models can correctly *rank* the QoE achieved by two different ABR algorithms. For each pair of source video and throughput trace, we first rank every two of the three ABR algorithms using their true QoE and then again using the predicted QoE. If the rank is different, this pair is called a discordant pair. The y-axis of Figure 2 shows the fraction of discordant pairs among all possible pairs (a common measure used in rank correlation): over 10.2% of pairs are discordant even for the most accurate QoE model. This suggests that using QoE predictions to compare different algorithms (*e.g.,* [45, 56, 83]) may not be reliable.

2.2 Temporal variability of quality sensitivity

Figure 2 shows that, unlike prior methods, our QoE model (§4) can predict QoE and rank ABRs significantly more accurately when applied on the same train/test set. We argue that this gap stems from a common assumption shared by all previous QoE models, which is that all factors affecting QoE can be captured by a handful of objective metrics. This premise ignores the impact of high-level video content (rather than low-level pixels and frames) on users' sensitivity to quality at different parts of the video. We now demonstrate how this quality sensitivity varies as video content changes.

Quantifying dynamic quality sensitivity: Users' sensitiv-



(a) 1-sec rebuffering (b) 4-sec rebuffering (c) Bitrate drop

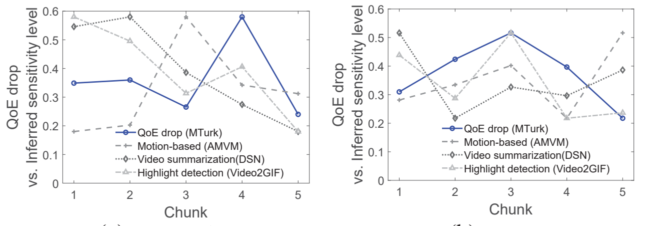
Figure 3: *Impact of different quality incidents at different points in the video in Figure 1. The pattern of variability remains the same across the different quality incidents. Error bars show standard deviation of the means.*

ity to quality at a certain part of a video is reflected by the *QoE drop* when a low-quality incident occurs at that part of the video, *i.e.,* $\Delta = Q_{before} - Q_{after}$, where Q_{before} is the MOS of the video without the low-quality incident and Q_{after} is the MOS of the video with the low-quality incident. To measure the true quality sensitivity at different parts of a source video, we create a *rendered video series* as follows. Rendered videos in a video series have the same source content and highest quality (highest bitrate without rebuffering), except that a low-quality incident (a rebuffering event or a bitrate drop) is deliberately added at different positions, *e.g.,* at the 4th second, 8th second, and so forth. Then, as before, we use Amazon MTurk to crowdsource the true QoE of each rendered video, following our crowdsourcing methodology (§4,§5).

Figure 1 shows an example video series created using a 25-second soccer video as the source video and a one-second rebuffering event as the low-quality incident. We observe significant differences between the QoE drops caused by the rebuffering event at different parts of the video. The highest QoE drop (caused by rebuffering at the 15th second) is 2.1× higher than the lowest QoE drop (rebuffering at the 10th second). This shows that a low-quality incident can have a significantly higher/lower impact on user experience if it occurs a few seconds earlier or later.

Quality sensitivity is inherent to video content: Our user study also suggests that the type of low-quality incident does not affect the ranking of QoE drops within a video series, even though it affects the absolute QoE drops. In other words, quality sensitivity seems to be *inherent* to different parts (contents) of the video. Figure 3 shows the dynamic user sensitivity of three low-quality incidents on the same source video: 1-second rebuffering, 4-second rebuffering, and a bitrate drop from 3Mbps to 0.3Mbps for 4 seconds. Although the absolute values of the QoE drops depend on the particular quality incident, the relative rankings are identical. The strong rank correlation (measured by Spearman's rank coefficient) is persistent across all videos in our dataset: 0.95 rank coefficient between the 1-second and 4-second rebuffering events and 0.94 between the 1-second rebuffering and bitrate drop.

Sources of dynamic quality sensitivity: We speculate that the dynamic quality sensitivity stems from users paying different degrees of attention to different parts of a video. In our



(a) `Soccer1` (b) `FPS`

Figure 4: *QoE rating drop when adding a 1-sec rebuffering at different points in the video, compared to chunk sensitivity levels inferred by saliency models.*

dataset, we identify at least three types of moments when users tend to be more (or less) attentive to video quality than usual. The first are key moments in the storyline of a video when tensions have built up; *e.g.,* in `BigBuckBunny` (animation) when the bullies fall into a trap set by the bunny, or in `Soccer1` when a goal is scored. The second are moments when users must pay attention to get important information; *e.g.,* showing the scoreboard in sports videos (`Soccer2`), or acquiring supplies after killing an enemy (`FPS2`). The third are transitional moments with scenic backgrounds, when users tend to be less attentive to quality; *e.g.,* the universe background in `Space`.

2.3 Modeling quality sensitivity

Can it be captured by QoE models? Traditional QoE models predict the same QoE for all rendered videos in a video series. Even models that do predict different QoE assume that the impact of video content can be captured by pixels and motions; *e.g.,* VMAF [11] (the visual quality metric used by KSQI) gives lower QoE estimates if a bitrate drop occurs when the frame pixels are more "complex". Unfortunately, the impact of content on user sensitivity discussed above cannot be fully captured by pixel-level patterns. In Figure 1, the true highest QoE drop occurs when the low-quality incident occurs during the goal, but both VMAF and LSTM-QoE predict that it occurs during normal gameplay.

Can it be captured by vision saliency? User sensitivity is conceptually similar to temporal saliency in computer vision. Can saliency/highlight detection models capture user sensitivity to quality? We examine three representative approaches.

- *Traditional motion-based models*, such as AMVM (average motion-vector magnitude) [13, 52], use the motion vector magnitudes of pixels in a chunk to indicate user sensitivity—*i.e.,* users are more sensitive to more dynamic scenes.
- *Interestingness score per frame (highlight detection)*, such as Video2GIF [39] and [34], train a regression model (using C3D [75] neural network as the spatio-temporal feature extractor) on videos with human-annotated per-frame interestingness scores.[4] The model then produces a per-frame interestingness score which might indicate user sensitivity.

[4]We notice that some content providers passively monitor the number of viewers at different parts of a video (*e.g.,* [6]), which is an alternative way of identifying highlights or high-interestingness chunks.

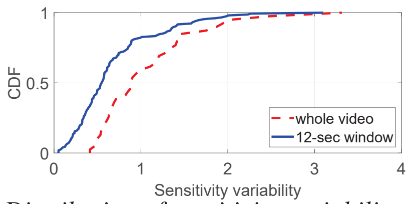


Figure 5: *Distribution of sensitivity variability when a low-quality incident (1-second rebuffering, 4-second rebuffering, or a bitrate drop for 4 seconds) is added at different points in the same video. The trend is similar even if the low-quality incident and QoE gap are localized to a 12-second window.*

- *Video summarization models*, such as dppLSTM [87] and DSN [90], infer how important each frame is to the whole story of a video, by extracting vision features [74] and using an LSTM to model temporal dependencies. The more important a frame is, the higher user sensitivity might be.

Figure 4 shows the average saliency scores (normalized to $[0,1]$) returned by these models at each chunk of two example videos. We see a weak correlation between the QoE drops caused by a 1-sec rebuffering event at different chunks and the true user sensitivity. Overall, such correlation is low for all videos in our dataset: less than 0.23 (Pearson's correlation) and 0.18 (Spearman's rank correlation). To see an example, in the soccer video (Figure 1), the part right before the goal is the most quality sensitive. However, the highlight detection and motion-based models highlight the highly dynamic scenes that pan across the audience, and the video summarization model picks diverse moments of a video, such as shot/rewind clips, whereas users pay more attention to when a goal might be scored. Appendix D gives more discussions. As a result, ABR logic based on saliency scores performs poorly (§7).

2.4 Potential gains

Dynamic quality sensitivity is prevalent: We repeat the same experiment from Figure 1 on all 16 source videos in our dataset and three low-quality incidents: 1-second rebuffering, 4-second rebuffering, and a bitrate drop from 3Mbps to 0.3Mbps for 4 seconds. This creates 48 video series in total. Figure 5 plots the *sensitivity variability* defined by $(\Delta_{max}-\Delta_{min})/\Delta_{min}$ for each video series, where Δ_{max} and Δ_{min} are the maximum and minimum QoE drop of the videos in a series. We see that 21 of the 48 video series have a sensitivity variability of over 0.99, while some have less than 0.20 variability. A similar trend holds even if we localize the low-quality incident and sensitivity gap measurement to 12-second windows. The fact that quality sensitivity varies substantially even among very nearby chunks suggests a new opportunity: we can lower the quality when sensitivity is low in order to save bandwidth for nearby chunks whose sensitivity is high.

Potential sensitivity-aware improvement: The above suggests that we can improve ABR algorithms to optimize QoE and save bandwidth by *aligning quality adaptation with dynamic user sensitivity*. We demonstrate the potential gains

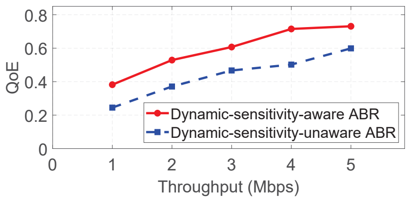


Figure 6: *Being aware of dynamic quality sensitivity can significantly improve QoE and save bandwidth.*

using an idealistic but clean experiment. We create two simple ABR algorithms whose only difference is the QoE model they optimize: one algorithm optimizes KSQI, the most accurate QoE model from Figure 2 that is *unaware* of dynamic quality sensitivity, and the other optimizes our eventual QoE model from §4, which *is* aware of dynamic quality sensitivity. Both algorithms take as input an entire throughput trace and the same 4-second video chunks encoded using the same bitrate levels. They then determine a bitrate-to-chunk assignment that maximizes their respective QoE model. Note that these ABR algorithms are idealistic because they have access to the entire throughput trace in advance, and hence know the future throughput variability. However, this allows us to eliminate the confounding factor of throughput prediction. We pick one of the throughput traces (results are similar with the other traces) and rescale it to $\{20,40,\ldots,100\}\%$ to emulate different average network throughputs.

For each source video, we create the rendered video as if it were streamed by each ABR algorithm (with bitrate switches, rebufferings, etc.). We use Amazon MTurk as before to assess the true QoE of the rendered video. Figure 6 shows the average QoE of the two ABR algorithms across 16 source videos and different average bandwidths. We see that being aware of dynamic quality sensitivity could improve QoE by 22-52% while using the same bandwidth, or save 39-49% bandwidth while achieving the same QoE.

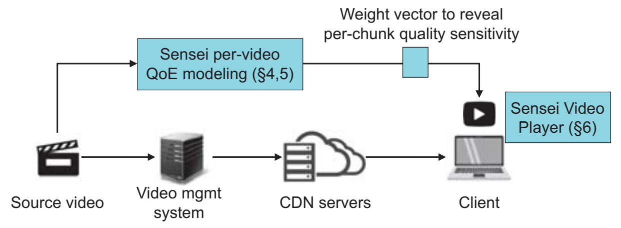
3 Overview of SENSEI

We have shown that knowing the true quality sensitivity of a video can lead to significant performance improvements when streaming the video. To unleash this potential, we present SENSEI, a video streaming system that unearths and leverages dynamic quality sensitivity. Here, we overview SENSEI (§3.1) and then introduce our crowdsourcing-based approach to per-video QoE modeling and its limitations (§3.2).

3.1 SENSEI's approach

As shown in Figure 7, SENSEI has two main components.

Per-video QoE modeling: Before streaming a video, SENSEI profiles the quality sensitivity of its chunks. As we saw in §2.2, prior QoE models fail to capture content-induced user sensitivity to quality. Instead, we advocate for directly asking human viewers to rate the quality of rendered videos with quality incidents inserted at various chunks. This reveals the true user sensitivity to quality incidents. Since quality sensitivity is unique to each video, this user study must be scaled


Weight vector to reveal per-chunk quality sensitivity
Sensei per-video QoE modeling (§4,5)
Sensei Video Player (§6)
Source video
Video mgmt system
CDN servers
Client
Figure 7: *Overview of* SENSEI.

to many videos. SENSEI uses crowdsourcing to automate and scale the per-video QoE modeling, by addressing two challenges: (1) how many (and which) rendered videos must be rated to build a sensitivity-aware QoE model (§4); and (2) how to get reliable ratings from crowdsourced workers (§5).

Sensitivity-aware ABR: Video players today are designed to maximize bitrate without rebuffering on *every* chunk. This is ill-suited to our goal of aligning quality adaptation with dynamic quality sensitivity: quality should be optimized in proportion to the quality sensitivity of the content. To achieve this, SENSEI refactors the control logic of video players to enable new adaptation actions that "borrow" resources from low-sensitivity chunks and give them to high-sensitivity chunks. We discuss the details in §6.

Instead of building a separate QoE model for each video, SENSEI reuses existing QoE models but reweights each chunk by its quality sensitivity. This is inspired by our observation that relative quality sensitivity is inherent to the content, rather than the specific quality incident (§2.2). Thus we assign a weight to each chunk to encode its inherent quality sensitivity. The abstraction of *per-chunk weights* has two benefits. First, it allows us to reuse existing QoE models by simply reweighting the quality of different chunks. Second, by using the sensitivity weights as input, the same SENSEI ABR algorithm can be used to optimize QoE for any new video.

3.2 Crowdsourcing quality sensitivity per video

SENSEI directly elicits quality ratings from human viewers to reveal their quality sensitivity to various quality incidents. However, these user ratings must be elicited *per video* and the sheer scale of this feedback can be prohibitive! To put it into perspective, QoE models are usually built from user ratings on just a handful of source videos [21,31], but getting enough user ratings requires a lab environment (or survey platform) to recruit participants and have them watch over two orders of magnitude more video content than the source videos. [5] This does not scale if we repeat the process per video.

To address this, we use crowdsourcing platforms like Amazon MTurk [1] to automate the user studies and scale them out to more videos. Crowdsourcing reduces the overhead of participant recruitment, survey dissemination, and result collection (down to about 78 minutes), and provides a large pool of participants. This allows for repeated experiments to

[5]For instance, in the WaterlooSQOE-III dataset [31], each video is streamed over 13 throughput traces with 6 ABR algorithms, and each rendered video is then rated by 30 users.

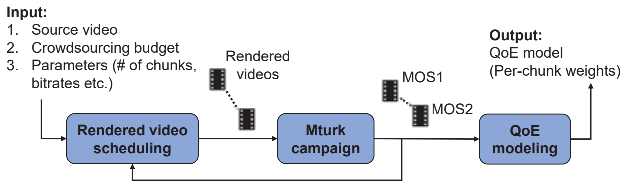


Figure 8: *Workflow of profiling dynamic quality sensitivity using a crowdsourcing platform. The arrow back to the scheduler means that crowdsourced ratings may be used to suggest more rendered videos to iteratively refine the QoE modeling.*

help control for human-related statistical noise. Although the crowdsourcing cost grows with video length, SENSEI offers several techniques to reduce the cost (see §4). Thus, the content providers can decide whether and how to initiate profiling given their budgets. Note that our reliance on crowdsourcing makes some scenarios, *e.g.,* live video streaming, currently inapplicable (see §9).

4 Profiling Quality Sensitivity at Scale

In this section, we show how to build an accurate and cost-efficient QoE model using crowdsourcing. We overview our workflow (§4.1) and then discuss low-cost methods for chunk-level reweighting (§4.2) and crowdsourcing scheduling (§4.3).

4.1 QoE modeling workflow

Figure 8 shows SENSEI's workflow for QoE modeling. SENSEI takes a source video and a monetary budget as input and returns a QoE model that incorporates dynamic quality sensitivity (customized for this video) as output.

- *Rendered video scheduling (§4.3):* We first generate a set of *rendered videos* from the source video. Each rendered video is created by injecting a carefully selected low-quality incident at a certain point in the video.
- *MTurk campaign (§5):* The rendered videos are published on the MTurk platform and we specify how many *participants* (MTurkers) to recruit for this *campaign*. When an MTurker signs up, they start a *survey* that asks them to watch K rendered videos and, after each video, rate its QoE.
- *QoE modeling (§4.2):* Finally, we use the MOS of each rendered video as its QoE and use regression to derive the per-chunk weights, which are then incorporated into an existing QoE model to derive the QoE model for this video.

4.2 Cutting cost via chunk-level reweighting

While crowdsourcing scales QoE profiling elastically, profiling each video can still be prohibitively expensive. Since a QoE model must capture the impact of both quality incidents and the quality sensitivity of each chunk, a strawman solution would build a QoE model with $O(N \cdot P)$ parameters, where N is the number of chunks and P is the number of parameters in a traditional QoE model. This could require a prohibitive number of ratings to build (*e.g.,* KSQI has tens of parameters).

Encoding quality sensitivity with per-chunk weights: We

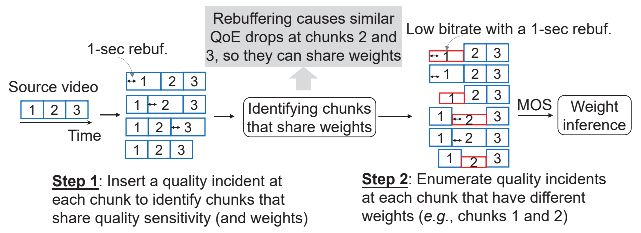

1-sec rebuf.
Rebuffering causes similar QoE drops at chunks 2 and 3, so they can share weights
Low bitrate with a 1-sec rebuf.
Source video
1 2 3
Time
Identifying chunks that share weights
MOS
Weight inference
Step 1: Insert a quality incident at each chunk to identify chunks that share quality sensitivity (and weights)
Step 2: Enumerate quality incidents at each chunk that have different weights (e.g., chunks 1 and 2)

Figure 9: *A running example of the crowdsoucring scheduler for a source video with 3 chunks, 2 bitrate levels (high and low), 2 rebuffering event levels (0 and 1 second).*

leverage the insight that quality sensitivity at a chunk is inherent to its video content (§2.2). Thus, SENSEI assigns a single weight to each chunk irrespective of the quality incident, reducing the number of model parameters to $O(N)$. Then, SENSEI *reuses an existing QoE model but reweights the chunks by their quality sensitivity.* If the QoE model is *additive*, *e.g.,* the overall QoE is the sum of the QoE estimates of individual chunks q_i, or $Q = \sum_{i=1}^{N} q_i$, then SENSEI can directly reweight the chunks by their quality sensitivity. Though some QoE models are non-additive (*e.g.,* LSTM-QoE), many mainstream QoE models including KSQI and others [56, 85] are. For KSQI, the q_i take into account the impact of visual quality, rebuffering, and quality switches. SENSEI reweights the QoE model as follows:

$$Q = \sum_{i=1}^{N} w_i q_i, \qquad (1)$$

where w_i is the weight of the i^{th} chunk, reflecting how much more sensitive users are to quality incidents in this chunk than in other chunks.

Weight inference: Given any V rendered videos, if Q_j is the QoE (MOS) of the j^{th} rendered video and $q_{i,j}$ is the estimated QoE of the i^{th} chunk of the j^{th} rendered video, then we can write V equations, $Q_j = \sum_{i=1}^{N} w_i q_{i,j}$ for $j = 1,\ldots,V$. We can then infer the w_i using a linear regression.

In the remainder of the paper, we assume that KSQI reweighted by Equation 1 is the QoE model of SENSEI.

4.3 Crowdsourcing scheduler

We now turn our attention to compiling a small set of rendered videos that, after being rated, will produce enough data to reliably estimate the per-chunk weights.

Two-step scheduling: Given a source video, SENSEI's scheduler uses a two-step process to decide which rendered videos to publish and how many participants to elicit ratings from.

- First, SENSEI creates a set of N rendered videos, each with a single 1-second rebuffering event at a different chunk (recall N is the number of chunks). It then publishes these videos and asks M_1 participants to rate each video. The total rendered video duration is $O(N \cdot M_1)$. Once the videos are rated, we infer the per-chunk weights as described above.
- Second, we pick $N' \ll N$ chunks whose inferred weights are α-high or low (*e.g.,* 6 % higher or lower than the average weight). We then repeat the first step with two differences: (1) low-quality incidents are added only to these chunks, and (2) the quality incidents include B bitrates (below the highest bitrate) and F rebuffering events (1,2,... seconds). We publish the rendered videos and ask M_2 participants to rate them, for a total video duration of $O(N' \cdot B \cdot F \cdot M_2)$.

The purpose of the first step is to use a small number of participants (M_1) to get a noisy but indicative estimate of which chunks have quality sensitivity that is very high or low, so we can *focus* the second iteration on these chunks using a larger number of participants (M_2). In general, for an ABR algorithm to improve QoE-bandwidth tradeoffs, it is more important to identify which chunks have very high/low quality sensitivity than to precisely estimate the quality sensitivity of every chunk. §5 discusses the number of participants; we evaluate the effect of α, B and F in §7.4. These parameters are empirically selected and held constant throughout our tests.

Figure 9 shows an example two-step schedule for a source video. In the first step, we generate a series of rendered video with the same rebuffering event injected at different chunks. By examining the ratings of these videos from the MTurkers, we determine that chunks 2 and 3 have similar sensitivity to the rebuffering event, allowing them to share the same chunk-level weight. Thus, in the second step, we only need to enumerate the quality incidents for chunks 1 and 2. In practice, for a 20-second video, we generate 5 rendered videos in the first step for the $N = 4$ chunks, of which $N' = 2$ chunks may have high/low sensitivity, and generate 15 rendered videos in the second step for these chunks.

Quality incidents used in profiling: For the set of B bitrates, we use the bitrate levels of YouTube videos and pick three of them to cover high, medium and low visual quality; we found this to be a practical compromise. The set of rebuffering events F are chosen to match those we plan to proactively add to the video (see §6). Testing on a larger set of quality incidents would yield more data points, but our microbenchmarking results in §7.4 show that this only marginally improves model accuracy, while significantly increasing the cost.

5 Reliable QoE Crowdsourcing

SENSEI's QoE model crucially depends on the *reliability* of MTurkers' quality ratings. This section describes our user survey procedure and techniques for increasing reliability. While SENSEI mostly follows known practices [41, 43, 57], we provide some key details that arose from our experience (described below and in Appendix B).

Single-survey procedure: As shown in Figure 21, each survey starts with the instructions and rejection criteria under which the ratings will be rejected. The MTurker then watches an example video that includes a quality incident, so they know what their ratings should be based on. Then the MTurker is asked to watch a sequence of rendered videos (determined by the scheduler) and, after each video, rate its quality on a scale of 1-5. Finally, the MTurker does an exit survey.

Quality control per survey: Several measures are taken to prevent and filter out spurious user ratings. First, we show the test videos in a *randomized order* to each MTurker. This eliminates biases due to viewing order and which videos were previously watched. Second, we add *reliability checks*: we show a video without any quality incident at a random position among the test videos, and if an MTurker does not give the highest score to this video, we discard all of their ratings. We also ask the MTurker what quality incident(s) they just saw in the last video, and if they report more quality incidents than were included, that rating is discarded. This may occur if the MTurker's network connection is poor and new quality incidents are introduced. Third, we implement an engagement test to verify if the MTurker watched the video in its entirety, by monitoring the time spent on the video playback page and discarding the rating if the time is shorter than the video length. We also implement other filters, such as limiting the number or length of videos per MTurker to prevent fatigue.

Use of Master MTurkers: We follow a common practice (*e.g.,* [53]) and restrict our tests to *Master MTurkers*, a class of reliable MTurkers who have participated in over 1000 surveys and whose feedback was accepted for over 99% of their prior surveys. We find that our rejection rate from Master MTurkers is over 4× lower than normal MTurkers. One lesson we learned is that Master MTurkers are more willing to participate if the publisher (us) historically has a low rejection rate because they wish to maintain their rejection rate below 1%.

Sanity check of our dataset: To check if MTurker ratings are similar to prior lab studies [41, 77], we select three 12-second videos from a public dataset [31] whose quality ratings are collected in a lab environment, and obtain MTurker ratings for these videos. We find that the MTurker responses are similar to the in-lab study: after normalizing the ratings to the same range, the MTurkers' MOS differs by less than 3% from the in-lab study's MOS on the same video.

How many MTurkers are needed? We did a head-to-head comparison with WaterlooSQOE-III [31] and found that we need 17% more MTurkers to reduce the variance of QoE ratings down to the levels of the in-lab study. §7.4 shows how the number of MTurkers affects SENSEI's performance.

Despite the above, we acknowledge that our MTurk survey methodology could be susceptible to human factors.

6 SENSEI's ABR Logic

The key difference between SENSEI's ABR logic and traditional ABR logic is that SENSEI aligns quality adaptation with the temporal variability of quality sensitivity. We first show how SENSEI modifies a traditional ABR framework (§6.1), and then show how existing ABR algorithms can be minimally modified to benefit from SENSEI (§6.2).

6.1 Enabling new adaptation actions

SENSEI takes a pragmatic approach by working within the framework of existing players. It proposes specific changes to their input and output, as highlighted in Figure 10.

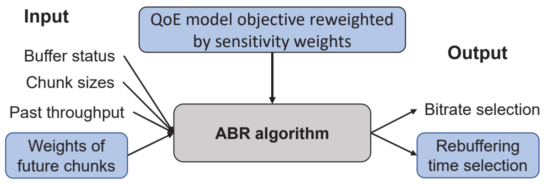


Figure 10: *ABR framework of* SENSEI. *The differences with traditional ABR framework are highlighted.*

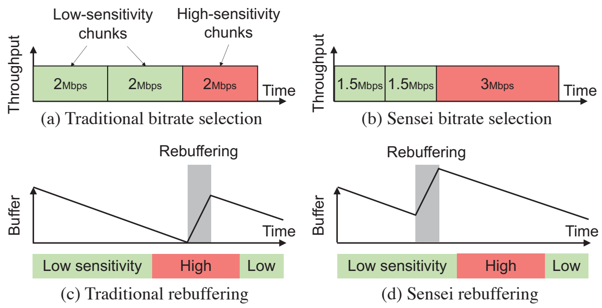


Figure 11: *Illustrative examples of* SENSEI *vs traditional ABR logic: how* SENSEI *improves quality (a vs. b) or avoids bad quality (c vs. d) for high-sensitivity chunks.*

Input: Besides the current buffer length, next chunk sizes, and history of throughput measurements, SENSEI's ABR algorithm takes as input the sensitivity weights of the next h chunks, where h is the *lookahead horizon*. A larger h allows us to look farther into the future for opportunities to trade current quality for future quality, or vice versa. In practice, we are also constrained by the reliability of our bandwidth prediction for future chunks. We microbenchmark the selection of h in §7.5.

Output: SENSEI's ABR algorithm selects the bitrate for future chunks as well as when the next rebuffering event should occur.[6] In contrast, traditional players only initiate rebuffering events when the buffer is empty.

QoE model objective: If the ABR algorithm explicitly optimizes an additive QoE model, SENSEI can modify its objective as described in §4.2. While SENSEI can be applied to most ABR algorithms (*e.g.,* [56, 83, 85]), some (*e.g.,* BBA) do not have an explicit objective that SENSEI can build on.

In theory, these changes are sufficient to enable at least the following optimizations, which traditional ABR algorithms are unlikely to explicitly do. (1) Lowering the current bitrate so that it can raise the bitrate for the next few chunks, if they have higher quality sensitivity (Figures 11(a) and (b)). (2) Raising the current bitrate slightly over the sustainable level if quality sensitivity is expected to decrease in the next few chunks. (3) Initiating a short rebuffering event now in order to ensure smoother playback for the next few chunks, if they have higher quality sensitivity (Figures 11(c) and (d)).

[6] SENSEI currently makes adaptation decisions only for the next chunk, but in principle it could plan adaptations for multiple chunks in the future.

6.2 Refactoring current ABR algorithms

We apply SENSEI to two ABR algorithms: Pensieve [56], based on deep reinforcement learning, and Fugu [83], a more traditional algorithm based on bandwidth prediction.

Applying SENSEI to Pensieve: SENSEI leverages the flexibility of deep neural networks (DNNs) and augments Pensieve's input, output and QoE objective—its states, actions, and reward, in the terminology of reinforcement learning—as described in §6.1. It then retrains the DNN model in the same way as Pensieve; we call this variation SENSEI-Pensieve. SENSEI-Pensive makes two minor changes to reduce the action space (which now includes rebuffering). First, we restrict possible rebuffering times to three levels ($\{0,1,2\}$ seconds) that can only happen at chunk boundaries. Second, instead of choosing among combinations of bitrates and rebuffering, SENSEI-Pensieve either selects a bitrate or initiates a rebuffering event at the next chunk. If it chooses the latter, SENSEI-Pensieve will increment the buffer state by the chosen rebuffering time and rerun the ABR algorithm immediately.

Applying SENSEI to Fugu: Let us first explain how Fugu works. At a high level, before downloading the i^{th} chunk, Fugu considers the throughput prediction for the next h chunks. For any throughput variation γ (with predicted probability $p(\gamma)$) and bitrate selection $B = (b_i, \ldots, b_{i+h-1})$, where b_j is the bitrate of the j^{th} chunk, it simulates when each of the next h chunks will be downloaded and estimates the rebuffering time $t_j(B,\gamma)$ of the j^{th} chunk (which could be zero). It then picks the bitrate vector $(b_i, \ldots, b_{i+h-1})$ that maximizes the expected total quality over the next h chunks and possible throughput variations: $\sum_\gamma p(\gamma) \sum_{j=i}^{i+h-1} q(b_j, t_j(B,\gamma))$. Here, $q(b,t)$ estimates the quality of a chunk with bitrate b and rebuffering time t using a simplified model of KSQI.

The SENSEI variation of Fugu, which we call SENSEI-Fugu, uses Fugu's throughput prediction and the sensitivity weights w_j of the next h chunks. SENSEI-Fugu picks the bitrate vector $B = (b_i, \ldots, b_{i+h-1})$ and the rebuffering time vector $T = (t_i, \ldots, t_{i+h-1})$, where t_j is the rebuffering time of the j^{th} chunk, that maximizes the expected total quality over the next h chunks and possible throughput variations:

$$\sum_\gamma p(\gamma) \sum_{j=i}^{i+h-1} w_j q(b_j, t_j) \tag{2}$$

Here, the chosen rebuffering times must be feasible, *i.e.,* the buffer length can never be negative.

In short, SENSEI-Pensieve and SENSEI-Fugu add an extra action (rebuffering time per chunk), and their objective function reweights the contribution of each chunk's quality using the sensitivity weights provided by our QoE model.

6.3 Player implementation and integration

We implement SENSEI on DASH.js [3], an open-source player that several commercial players are based on. We add a new field in the DASH manifest file (under `Representation`) to represent per-chunk sensitivity weights and change the parser `ManifestLoader` to parse these weights. Unlike other ABR players, SENSEI may initiate rebuffering when the buffer is not empty. We use Media Source Extensions [7] (an API that allows browsers to change player states) to delay a downloaded chunk in the browser buffer from being loaded into the player buffer. We also describe the implementation of our crowdsourcing pipeline for MTurk surveys in Appendix C.

7 Evaluation

Our evaluation of SENSEI shows several key findings:

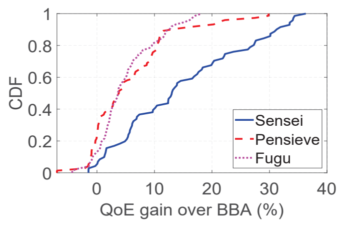
- Compared to recent proposals, SENSEI can improve QoE by 7.7-52.5% without using more bandwidth or can save 12.1-50.3% bandwidth while achieving the same QoE.
- The performance gains of SENSEI come at a cost of \$31.4/minute video, which is marginal compared to the investments made by content providers.
- SENSEI can improve QoE prediction accuracy by 11.8-37.1% over state-of-the-art QoE models.
- SENSEI's ABR algorithm consistently outperforms baseline ABR algorithms even when bandwidth fluctuates.

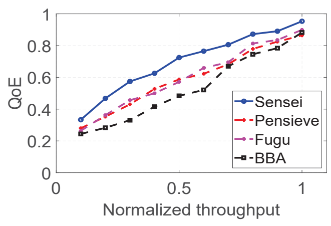
7.1 Experimental setup

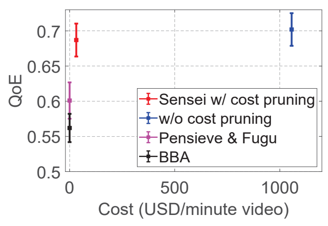
Test videos and throughput traces: Our test videos are selected from four datasets: LIVE-MOBILE [36], LIVE-NFLX-II [21], and WaterlooSQOE-III [31] are professional-grade datasets often used to train/compare QoE models in the literature. We complement these sources with videos from a user-generated dataset, YouTube-UGC [78]. The videos are randomly selected from four video genres: sports, gaming, nature, and animation. Appendix §A provides more details about the videos. To create an adaptive video streaming setup, we chop videos into 4-second chunks and encode each chunk at 5 bitrate levels: $\{300, 750, 1200, 1850, 2850\}$Kbps. We randomly select 10 throughput traces from two public datasets, FCC [24] and 3G/HSDPA [65], restricting our selection to those whose average throughput is between 0.2Mbps and 6Mbps, forcing the ABR algorithms to adapt their bitrates.

Baselines: We compare SENSEI's ABR algorithm with three baselines: Buffer-based adaptation (BBA) [45], Fugu [83], and Pensieve [56]. We keep their default settings (*e.g.,* same DNN architecture and training network traces for Pensieve, etc.). For fairness, we use KSQI as the QoE model for Pensieve, Fugu, and the SENSEI variants. This modification should improve the quality of Pensieve and Fugu, because the QoE models used in their original implementations are special cases of KSQI. We use SENSEI-Pensieve (*i.e.,* the application of SENSEI to Pensieve) as SENSEI, but confirm that the improvements of SENSEI-Fugu are similar (Figure 18a).

Performance metrics: We use three performance metrics. For a given source and video and throughput trace, we report the **QoE gain** of one ABR algorithm (Q_1) over another (Q_2), *i.e.,* $(Q_1 - Q_2)/Q_2$, where Q_1 and Q_2 are *rated by MTurkers*.


(a) QoE gains over BBA


(b) QoE vs. bandwidth usage


(c) MTurk cost vs. QoE

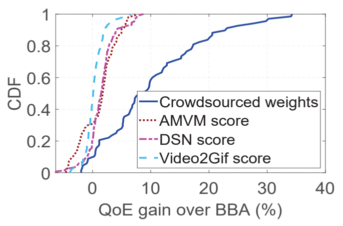

(d) SENSEI with crowdsourced weight vs. saliency-based weights

Figure 12: *End-to-end performance of* SENSEI *over traditional and saliency-based ABR baselines, across all videos.*

We calculate SENSEI's **bandwidth saving** by scaling down the throughput traces and determining how much bandwidth each ABR algorithm needs to achieve the same QoE. We normalize all QoE values to the range $[0,1]$. We measure the **crowdsourcing cost** paid to MTurk to get enough ratings to profile a 1-minute video. Only SENSEI incurs this cost.

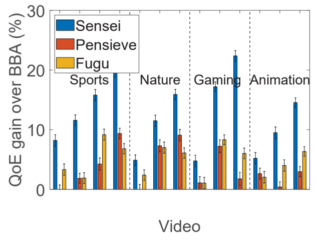
7.2 End-to-end improvement

QoE gains: Figure 12a shows the distributions of QoE gains of SENSEI, Pensieve, and Fugu over BBA, across all combinations of the 16 source videos and 10 network traces. Compared to BBA, SENSEI has at least 14.4% QoE gain for half of the trace-video combinations, whereas Pensieve's and Fugu's median QoE gains are about 5.7%. The tail improvement of SENSEI is greater: SENSEI's QoE gain at the 80th percentile is 4.8%, whereas Pensieve's and Fugu's are 0.2% and 0.7% respectively. The fact that SENSEI's gains over Pensieve (its base ABR logic) are similar to Pensieve's gains over BBA suggests the significant potential in making an existing ABR algorithm aware of dynamic quality sensitivity.

Bandwidth savings: Figure 12b shows the average QoE of different ABR algorithms across the source videos, under one throughput trace scaled down by different ratios (x-axis). We confirm the results are consistent across different throughput traces. We see that when setting a target QoE of 0.8, the bandwidth savings of SENSEI is about 27% higher compared to Pensieve and Fugu, and 32% higher compared to BBA.

QoE vs. crowdsourcing cost: Figure 12c shows the crowdsourcing cost and resulting QoE of SENSEI relative to Pensieve, both with and without the cost-pruning optimization (which is evaluated separately in Figure 16). Compared to enumerating all combinations of the quality incidents, we see that costs can be reduced by more than 32× with only a 3.1% degradation in QoE, and SENSEI is still 14.7% better on average than its base ABR logic (Pensieve with KSQI). This cost is equivalent to ∼\$31.4 per 1-minute video, which is a negligible cost for large content providers that may spend on the order of \$10 billion annually [4] for licensing popular TV shows (or making such shows).

Improvements by video and trace: Figure 13a shows the QoE gains for each video across the network traces. We see significant variability in the QoE gains across videos and even within the same genre. Figure 13b shows the QoE gains


(a) QoE gain grouped by genre

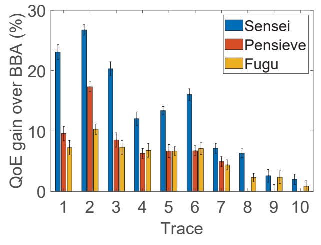

(b) QoE gain grouped by trace

Figure 13: *QoE gains over BBA for genre and for each throughput trace (ordered by increasing average throughput)*

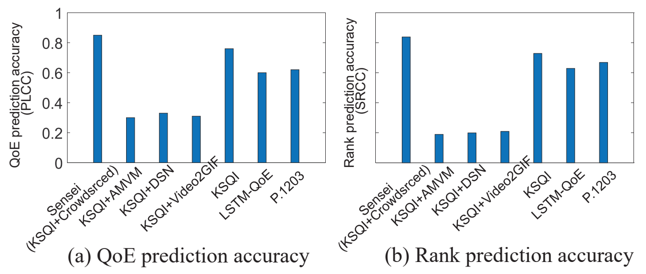
for each network trace across all videos. Overall, SENSEI yields more improvement when the average throughput is lower (towards the left). This shows that SENSEI can better maintain high QoE even when the network is under stress.

SENSEI vs. saliency-reweighted ABR: Finally, Figure 12d shows the QoE gains of SENSEI when the per-chunk weights are based on crowdsourcing results (our approach) and when the weights are based on the saliency scores produced by various saliency models (see §2.3). We normalize each model's saliency scores to sum to the sum of chunk weights of SENSEI. We see that SENSEI's gain significantly reduces if the weights are based on these saliency models, because as explained in §2.3, they fail to capture users' quality sensitivity.

7.3 QoE prediction accuracy

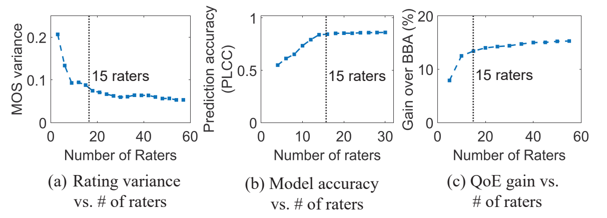
We now microbenchmark SENSEI's QoE model introduced in §4 using all 640 rendered videos generated by running SENSEI and the baseline ABR algorithms on all combinations of source videos and network traces. We obtain the "ground truth" QoE of each rendered video using our MTurk survey procedure (§4,§5). We calculate Pearson's linear correlation coefficient (PLCC) and Spearman's rank correlation coefficient (PRCC) between the predicted QoE and actual user-rated QoE. Figure 14 compares SENSEI with three baselines QoE models (KSQI, LSTM-QoE, P.1203). The PLCC (and PRCC) of SENSEI's QoE prediction is over 0.85 (and 0.84), whereas the baselines are below 0.76 (and 0.73). We evaluated several variants of KSQI (the best baseline QoE model) re-weighted by per-chunk saliency scores from the saliency models in §2.3, but their accuracies are even lower.

7.4 Cost savings on crowdsourcing



(a) QoE prediction accuracy (b) Rank prediction accuracy

Figure 14: *QoE prediction accuracy of* SENSEI*,* SENSEI*'s variants, and baseline QoE models.*



(a) Rating variance vs. # of raters (b) Model accuracy vs. # of raters (c) QoE gain vs. # of raters

Figure 15: *The effect of the number of raters*

We microbenchmark the effects of different crowdsourcing parameters on SENSEI's QoE model.

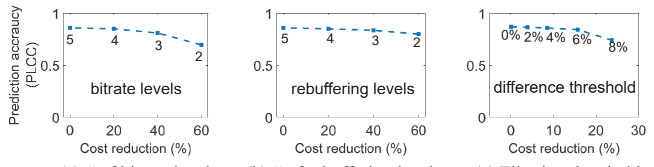
Impact of number of raters per video: Figure 15(a) shows that while the quality ratings have substantial variance with less than 5 raters, their mean value (MOS) stabilizes with more than 15 raters. As a result, having 15 raters per video (as used in SENSEI) produces a similar QoE prediction accuracy (b) and QoE gains (c) as having 30 raters.

Impact of crowdsourcing schedule granularity: Figure 16 shows the effect of reducing MTurk cost by considering (a) fewer bitrate levels (B), (b) fewer rebuffering events (F), or (c) higher threshold α used to pick which chunks to investigate in the second step. These terms are defined in §4.3. By reducing B to 3, F to 2, or raising α to 6%, we greatly reduce the cost while incurring less than 3% drop in accuracy.

7.5 SENSEI's ABR logic

Finally, we microbenchmark SENSEI's ABR logic (§6). To scale this experiment out, we use real videos and throughput traces but use the QoE predicted by SENSEI (instead of real user ratings) to evaluate QoE. We have confirmed that this yields the same QoE estimates on average as real user ratings under the same setting.

Impact of bandwidth variance: Figure 17 shows the performance of SENSEI under increasing throughput variance. We pick one throughput trace and increase its throughput variance by adding unbiased Gaussian noise. The graph begins at the variance of the original throughput trace; as variance increases, SENSEI's QoE degrades gracefully, but it still maintains a significant gain over its base ABR logic (Pensieve or Fugu). This is because SENSEI only needs to predict how likely low throughput will occur on high quality-sensitivity chunks, not all future chunks, so if the average throughput until the next such chunk is predictable, it will work well. We



(a) # of bitrate level (b) # of rebuffering level (c) Filtering threshold

Figure 16: *QoE model accuracy changes with cost.*

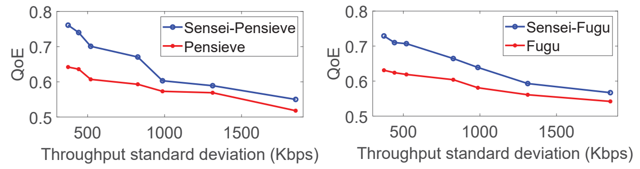


Figure 17: *QoE under increasing bandwidth variance.*

confirm the results are similar on other throughput traces.

Performance breakdown: Figure 18a shows that SENSEI achieves comparable improvement when either Pensieve or Fugu is the base ABR logic. This suggests that SENSEI's gains do not depend on the choice of the base ABR logic. Figure 18b shows that both aspects of SENSEI's control logic contributes to its improvements: (1) making ABR logic aware of dynamic quality sensitivity (1st vs. 2nd bar), and (2) injecting rebuffering judiciously (2nd vs. 3rd bar). Thus, even if a content provider cannot control rebuffering, it can still benefit significantly from SENSEI's dynamic quality sensitivity.

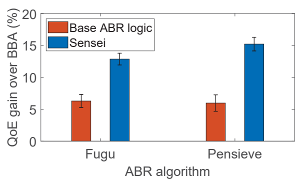
Impact of video contents: While the videos in our dataset have varying fractions (from 20% to 60%) of high-sensitivity chunks, Figure 18c tests Sensei's performance under an even wider range of high-sensitivity chunk fractions (from 0% to 100%). We create source videos with the specific fractions of high and low quality-sensitivity chunks and randomize the positions of the chunks. SENSEI has marginal improvement when the video is dominated by either high or low quality-sensitivity chunks. However, SENSEI significantly improves QoE when high quality-sensitivity chunks are 20-40% of a video (most of our videos fall in this range).

Lookahead horizon: Figure 18d tests the impact of lookahead horizon—the number of future chunks h whose quality-sensitivity weights are revealed to the ABR algorithm. A longer horizon increases SENSEI's ability to schedule quality events between low and high quality-sensitivity chunks. Empirically in our dataset, the QoE gains diminish after the lookahead horizon is greater than 4 chunks.

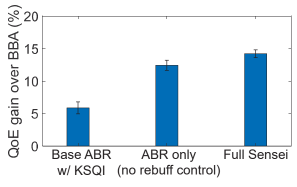
Systems overhead: We confirm empirically that compared to a video player without SENSEI, the runtime overhead of SENSEI is less than 1% in both CPU cycles and RAM usage.

8 Related Work

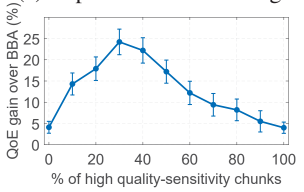
ABR algorithms: Mainstream ABR algorithms maximize bitrate under dynamic available bandwidth. Traditional ones are buffer-based (*e.g.,* [46, 50]) or rate-based algorithms (*e.g.,* [45, 69, 70]). Recent ABR algorithms explicitly optimize a given QoE objective via control theory [85], ML-based



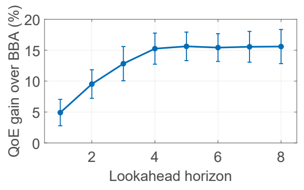
(a) Impact of base ABR logic



(b) Improvement breakdown



(c) Chunk sensitivity



(d) Lookahead horizon

Figure 18: *Understanding* SENSEI*'s improvements.*

throughput prediction [73, 83], or deep reinforcement learning [34, 35, 56]. Some ABR algorithms also rely on server-side processing [17, 47, 84]. Key parameters of the ABR logic can be customized to the network conditions or devices [16]. Though SENSEI reuses existing ABR algorithms, its contribution lies in identifying minimum changes (*e.g.,* adaptation actions they never would have taken) needed for these algorithms to fully leverage users' dynamic quality sensitivity.

Modeling and optimizing user-perceived quality: Visual quality assessment (VQA) traditionally models user's perception of encoded video using pixel-level patterns (*e.g.,* [38, 64, 68, 79]) as well as advanced data-driven models, such as SVM [11] and deep learning models (*e.g.,* [48]). Adaptive quality assessment (AQA), on the other hand, models streaming-related incidents, including join time, bitrate switches, rebuffering (*e.g.,* [18, 28, 49]). Recent QoE models combine VQA and AQA (*e.g.,* [19, 20, 22, 25, 29, 31, 33]) and sometimes uses spatial/temporal visual attention (*e.g.,* [30, 34, 34, 37, 59, 60, 82]). These perception-centric QoE models have inspired a large body of work that maximizes user-perceived quality with bitrate adaptation [58, 61], adaptive video encoding [17, 67, 89], adaptive bitrate levels [14, 15], dynamic chunk lengths [51], and super resolution [47, 84, 88].

Since the user-perceived quality metrics can vary across chunks, they may also treat video chunks differently, like SENSEI does. However, as elaborated in §2.3, SENSEI is complementary to these efforts: while they propose heuristics to how pixel-/motion-based visual features affect QoE, SENSEI customizes itself for each video (in a cost-efficient way) to capture the impact of the *substance of video content* on true user sensitivity to video quality. That said, actions like dynamic bitrate levels, chunk lengths, and super resolution could be used in SENSEI too, though SENSEI only considers actions directly supported by current DASH players.

QoE research using crowdsourcing: Prior work (*e.g.,* [23, 42, 43, 57, 62, 81]) provides methodologies for using commercial crowdsourcing platforms, *e.g.,* Amazon MTurk [1] and Microworkers [8], to systematically model user perception using objective quality metrics [23, 42, 43, 62, 81], investigate QoE impact of different types of low-quality events (*e.g.,* [32]), and build crowdsourcing platforms themselves for similar purposes (*e.g.,* [77, 83]). While SENSEI follows conventional crowdsourcing methodology (§5), SENSEI faces a unique challenge of scaling crowdsourcing to *per-video* QoE modeling. The cost of modeling QoE of each video separately is prohibitive, and SENSEI drastically prunes the cost by reusing an existing QoE model while profiling only a single weight per representative chunk to encode the content-induced quality sensitivity of each chunk.

9 Discussion

Participant selection bias: A concern of any crowdsourced user study is that the results could be biased because the workers who are willing to participate in the user study might have different characteristics than the real video viewers. A common approach to address this bias is to reweight the participant responses based on the demographics of real users (*e.g.,* [26, 55, 71, 76]). SENSEI could apply reweighting to the user study if we have knowledge of the target viewers' demographics, or it could directly recruit the user study participants from the target viewers themselves (*e.g.,* subscribers of the content provider).

Inapplicable scenarios: SENSEI does not apply to live video streaming and copyrighted videos. Live videos have strict delay requirements which our crowdsourcing-based video profiling cannot meet. Showing copyrighted videos to crowdsource workers poses the risk of copyright violation, though SENSEI could be used on already-released videos. Moreover, the profiling cost of $\sim$ \$31.4/minute video may still be impractical for videos with only a few views. Instead, we envision that SENSEI will be used for popular on-demand video by content providers who seek to improve their QoE-bandwidth tradeoffs. For example, providers such as Amazon, Netflix and YouTube recently lowered the default bitrate in Europe due to increased network traffic during the COVID lockdown [9].

10 Conclusion

We have described SENSEI, a video streaming system that optimizes video quality by exploiting dynamic quality sensitivity. Observing that quality sensitivity is inherent to video content and hence unique to each video, SENSEI scales out the profiling of quality sensitivity using a reliable crowdsourcing methodology. We show that with minor modifications to state-of-the-art ABR algorithms, SENSEI can improve their QoE by 15.1% or save bandwidth by 26.8% on average.

Acknowledgement

We thank the anonymous reviewers and our shepherd Dongsu Han. Xu Zhang and Junchen Jiang are supported by NSF (CNS-1901466), CERES Center, and a Google Faculty Research Award.

References

[1] Amazon Mechanical Turk. https://www.mturk.com/.

[2] Cisco Annual Internet Report (2018–2023) White Paper. https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html/.

[3] DASH.js. https://github.com/Dash-Industry-Forum/dash.js/wiki.

[4] How Netflix Pays for Movie and TV Show Licensing. https://www.investopedia.com/articles/investing/062515/how-netflix-pays-movie-and-tv-show-licensing.asp.

[5] HSDPA dataset. http://skuld.cs.umass.edu/traces/mmsys/2013/pathbandwidth/.

[6] Live stream on YouTube, See your live stream's metrics. https://support.google.com/youtube/answer/2853833.

[7] Media Source Extensions. https://www.w3.org/TR/media-source/.

[8] Microworkers. https://www.microworkers.com/.

[9] Reuters: YouTube, Amazon Prime forgo streaming quality to relieve European networks. https://uk.reuters.com/article/us-health-coronavirus-youtube-exclusive/exclusive-youtube-to-reduce-streaming-quality-in-europe-due-to-coronavirus-idUKKBN2170OP.

[10] Video Quality of Experience: Requirements and Considerations for Meaningful Insight. https://www.sandvine.com/hubfs/downloads/archive/whitepaper-video-quality-of-experience.pdf.

[11] VMAF - Video Multi-Method Assessment Fusion. https://github.com/Netflix/vmaf.

[12] YouTube joins Netflix in reducing video quality in Europe. https://www.theverge.com/2020/3/20/21187930/youtube-reduces-streaming-quality-european-union-coronavirus-bandwidth-internet-traffic.

[13] Golnaz Abdollahian, Cuneyt M Taskiran, Zygmunt Pizlo, and Edward J Delp. Camera motion-based analysis of user generated video. *IEEE Transactions on Multimedia*, 12(1):28–41, 2009.

[14] Hasti Ahlehagh and Sujit Dey. Adaptive bit rate capable video caching and scheduling. In *2013 IEEE Wireless Communications and Networking Conference (WCNC)*, pages 1357–1362. IEEE, 2013.

[15] Hasti Ahlehagh and Sujit Dey. Video-aware scheduling and caching in the radio access network. *IEEE/ACM Transactions on Networking*, 22(5):1444–1462, 2014.

[16] Zahaib Akhtar, Yun Seong Nam, Ramesh Govindan, Sanjay Rao, Jessica Chen, Ethan Katz-Bassett, Bruno Ribeiro, Jibin Zhan, and Hui Zhang. Oboe: auto-tuning video abr algorithms to network conditions. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 44–58, 2018.

[17] Ramon Aparicio-Pardo, Karine Pires, Alberto Blanc, and Gwendal Simon. Transcoding live adaptive video streams at a massive scale in the cloud. In *Proceedings of the 6th ACM Multimedia Systems Conference*, pages 49–60, 2015.

[18] Athula Balachandran, Vyas Sekar, Aditya Akella, Srinivasan Seshan, Ion Stoica, and Hui Zhang. Developing a predictive model of quality of experience for internet video. *ACM SIGCOMM Computer Communication Review*, 43(4):339–350, 2013.

[19] Christos G Bampis and Alan C Bovik. An augmented autoregressive approach to http video stream quality prediction. *arXiv preprint arXiv:1707.02709*, 2017.

[20] Christos G Bampis, Zhi Li, and Alan C Bovik. Continuous prediction of streaming video qoe using dynamic networks. *IEEE Signal Processing Letters*, 24(7):1083–1087, 2017.

[21] Christos G Bampis, Zhi Li, Ioannis Katsavounidis, Te-Yuan Huang, Chaitanya Ekanadham, and Alan C Bovik. Towards perceptually optimized end-to-end adaptive video streaming. *arXiv preprint arXiv:1808.03898*, 2018.

[22] Chao Chen, Lark Kwon Choi, Gustavo De Veciana, Constantine Caramanis, Robert W Heath, and Alan C Bovik. Modeling the time—varying subjective quality of http video streams with rate adaptations. *IEEE Transactions on Image Processing*, 23(5):2206–2221, 2014.

[23] Kuan-Ta Chen, Chi-Jui Chang, Chen-Chi Wu, Yu-Chun Chang, and Chin-Laung Lei. Quadrant of euphoria: a crowdsourcing platform for qoe assessment. *IEEE Network*, 24(2):28–35, 2010.

[24] Federal Communications Commission et al. Raw data-measuring broadband america. https://www.fcc.gov/reports-research/reports/. *Retrieved June*, 19:2018, 2016.

[25] Johan De Vriendt, Danny De Vleeschauwer, and David Robinson. Model for estimating qoe of video delivered using http adaptive streaming. In *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*, pages 1288–1293. IEEE, 2013.

[26] Djellel Difallah, Elena Filatova, and Panos Ipeirotis. Demographics and dynamics of mechanical turk workers. In *Proceedings of the eleventh ACM international conference on web search and data mining*, pages 135–143, 2018.

[27] Li Ding, Hua Huang, and Yu Zang. Image quality assessment using directional anisotropy structure measurement. *IEEE Transactions on Image Processing*, 26(4):1799–1809, 2017.

[28] Florin Dobrian, Vyas Sekar, Asad Awan, Ion Stoica, Dilip Joseph, Aditya Ganjam, Jibin Zhan, and Hui Zhang. Understanding the impact of video quality on user engagement. *ACM SIGCOMM Computer Communication Review*, 41(4):362–373, 2011.

[29] Zhengfang Duanmu, Wentao Liu, Diqi Chen, Zhuoran Li, Zhou Wang, Yizhou Wang, and Wen Gao. A knowledge-driven quality-of-experience model for adaptive streaming videos. *arXiv preprint arXiv:1911.07944*, 2019.

[30] Zhengfang Duanmu, Abdul Rehman, and Zhou Wang. A quality-of-experience database for adaptive video streaming. *IEEE Transactions on Broadcasting*, 64(2):474–487, 2018.

[31] Zhengfang Duanmu, Kai Zeng, Kede Ma, Abdul Rehman, and Zhou Wang. A quality-of-experience index for streaming video. *IEEE Journal of Selected Topics in Signal Processing*, 11(1):154–166, 2016.

[32] Sebastian Egger, Bruno Gardlo, Michael Seufert, and Raimund Schatz. The impact of adaptation strategies on perceived quality of http adaptive streaming. In *Proceedings of the 2014 Workshop on Design, Quality and Deployment of Adaptive Video Streaming*, pages 31–36, 2014.

[33] Nagabhushan Eswara, S Ashique, Anand Panchbhai, Soumen Chakraborty, Hemanth P Sethuram, Kiran Kuchi, Abhinav Kumar, and Sumohana S Channappayya. Streaming video qoe modeling and prediction: A long short-term memory approach. *IEEE Transactions on Circuits and Systems for Video Technology*, 2019.

[34] Guanyu Gao, Linsen Dong, Huaizheng Zhang, Yonggang Wen, and Wenjun Zeng. Content-aware personalised rate adaptation for adaptive streaming via deep video analysis. In *ICC 2019-2019 IEEE International Conference on Communications (ICC)*, pages 1–8. IEEE, 2019.

[35] Guanyu Gao, Huaizheng Zhang, Han Hu, Yonggang Wen, Jianfei Cai, Chong Luo, and Wenjun Zeng. Optimizing quality of experience for adaptive bitrate streaming via viewer interest inference. *IEEE Transactions on Multimedia*, 20(12):3399–3413, 2018.

[36] Deepti Ghadiyaram, Janice Pan, and Alan C Bovik. A subjective and objective study of stalling events in mobile streaming videos. *IEEE Transactions on Circuits and Systems for Video Technology*, 29(1):183–197, 2017.

[37] Deepti Ghadiyaram, Janice Pan, and Alan C Bovik. Learning a continuous-time streaming video qoe model. *IEEE Transactions on Image Processing*, 27(5):2257–2271, 2018.

[38] Rafael C Gonzalez, Richard Eugene Woods, and Steven L Eddins. *Digital image processing using MATLAB*. Pearson Education India, 2004.

[39] Michael Gygli, Yale Song, and Liangliang Cao. Video2gif: Automatic generation of animated gifs from video. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1001–1009, 2016.

[40] Kotaro Hara, Abigail Adams, Kristy Milland, Saiph Savage, Chris Callison-Burch, and Jeffrey P Bigham. A data-driven analysis of workers' earnings on amazon mechanical turk. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, pages 1–14, 2018.

[41] Tobias Hoßfeld, Matthias Hirth, Judith Redi, Filippo Mazza, Pavel Korshunov, Babak Naderi, Michael Seufert, Bruno Gardlo, Sebastian Egger, and Christian Keimel. Best practices and recommendations for crowdsourced qoe-lessons learned from the qualinet task force" crowdsourcing". 2014.

[42] Tobias Hossfeld, Christian Keimel, Matthias Hirth, Bruno Gardlo, Julian Habigt, Klaus Diepold, and Phuoc Tran-Gia. Best practices for qoe crowdtesting: Qoe assessment with crowdsourcing. *IEEE Transactions on Multimedia*, 16(2):541–558, 2013.

[43] Tobias Hoßfeld, Michael Seufert, Matthias Hirth, Thomas Zinner, Phuoc Tran-Gia, and Raimund Schatz. Quantification of youtube qoe via crowdsourcing. In *2011 IEEE International Symposium on Multimedia*, pages 494–499. IEEE, 2011.

[44] Sudeng Hu, Lina Jin, Hanli Wang, Yun Zhang, Sam Kwong, and C-C Jay Kuo. Compressed image quality metric based on perceptually weighted distortion. *IEEE Transactions on Image Processing*, 24(12):5594–5608, 2015.

[45] Te-Yuan Huang, Ramesh Johari, Nick McKeown, Matthew Trunnell, and Mark Watson. A buffer-based approach to rate adaptation: Evidence from a large video streaming service. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 187–198, 2014.

[46] Junchen Jiang, Vyas Sekar, and Hui Zhang. Improving fairness, efficiency, and stability in http-based adaptive video streaming with festive. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 97–108, 2012.

[47] Jaehong Kim, Youngmok Jung, Hyunho Yeo, Juncheol Ye, and Dongsu Han. Neural-enhanced live streaming: Improving live video ingest via online learning. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 107–125, 2020.

[48] Woojae Kim, Jongyoo Kim, Sewoong Ahn, Jinwoo Kim, and Sanghoon Lee. Deep video quality assessor: From spatio-temporal visual sensitivity to a convolutional neural aggregation network. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 219–234, 2018.

[49] S Shunmuga Krishnan and Ramesh K Sitaraman. Video stream quality impacts viewer behavior: inferring causality using quasi-experimental designs. *IEEE/ACM Transactions on Networking*, 21(6):2001–2014, 2013.

[50] Zhi Li, Xiaoqing Zhu, Joshua Gahm, Rong Pan, Hao Hu, Ali C Begen, and David Oran. Probe and adapt: Rate adaptation for http video streaming at scale. *IEEE Journal on Selected Areas in Communications*, 32(4):719–733, 2014.

[51] Jan Lievens, Shahid M Satti, Nikos Deligiannis, Peter Schelkens, and Adrian Munteanu. Optimized segmentation of h. 264/avc video for http adaptive streaming. In *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*, pages 1312–1317. IEEE, 2013.

[52] Yao Liu, Sujit Dey, Fatih Ulupinar, Michael Luby, and Yinian Mao. Deriving and validating user experience model for dash video streaming. *IEEE Transactions on Broadcasting*, 61(4):651–665, 2015.

[53] Matt Lovett, Saleh Bajaba, Myra Lovett, and Marcia J Simmering. Data quality from crowdsourced surveys: A mixed method inquiry into perceptions of amazon's mechanical turk masters. *Applied Psychology*, 67(2):339–366, 2018.

[54] Guo Lu, Wanli Ouyang, Dong Xu, Xiaoyun Zhang, Chunlei Cai, and Zhiyong Gao. Dvc: An end-to-end deep video compression framework. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 11006–11015, 2019.

[55] Jianguo Lu and Dingding Li. Estimating deep web data source size by capture–recapture method. *Information retrieval*, 13(1):70–95, 2010.

[56] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 197–210, 2017.

[57] M Sajid Mushtaq, Brice Augustin, and Abdelhamid Mellouk. Crowd-sourcing framework to assess qoe. In *2014 IEEE International Conference on Communications (ICC)*, pages 1705–1710. IEEE, 2014.

[58] Vikram Nathan, Vibhaalakshmi Sivaraman, Ravichandra Addanki, Mehrdad Khani, Prateesh Goyal, and Mohammad Alizadeh. End-to-end transport for video qoe fairness. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 408–423. 2019.

[59] Anh Nguyen, Zhisheng Yan, and Klara Nahrstedt. Your attention is unique: Detecting 360-degree video saliency in head-mounted display for head movement prediction. In *Proceedings of the 26th ACM international conference on Multimedia*, pages 1190–1198, 2018.

[60] Cagri Ozcinar, Julian Cabrera, and Aljosa Smolic. Visual attention-aware omnidirectional video streaming using optimal tiles for virtual reality. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(1):217–230, 2019.

[61] Benjamin Rainer, Stefan Petscharnig, Christian Timmerer, and Hermann Hellwagner. Statistically indifferent quality variation: An approach for reducing multimedia distribution cost for adaptive video streaming services. *IEEE Transactions on Multimedia*, 19(4):849–860, 2016.

[62] Benjamin Rainer, Markus Waltl, and Christian Timmerer. A web based subjective evaluation platform. In *2013 Fifth International Workshop on Quality of Multimedia Experience (QoMEX)*, pages 24–25. IEEE, 2013.

[63] ITUT Rec. H. 264: Advanced video coding for generic audiovisual services. *ITU-T Rec. H. 264-ISO/IEC 14496-10 AVC*, 2005.

[64] Abdul Rehman, Kai Zeng, and Zhou Wang. Display device-adapted video quality-of-experience assessment. In *Human Vision and Electronic Imaging XX*, volume 9394, page 939406. International Society for Optics and Photonics, 2015.

[65] Haakon Riiser, Paul Vigmostad, Carsten Griwodz, and Pål Halvorsen. Commute path bandwidth traces from 3g networks: analysis and applications. In *Proceedings of the 4th ACM Multimedia Systems Conference*, pages 114–118, 2013.

[66] Werner Robitza, Marie-Neige Garcia, and Alexander Raake. A modular http adaptive streaming qoe model—candidate for itu-t p. 1203 ("p. nats"). In *2017 Ninth International Conference on Quality of Multimedia Experience (QoMEX)*, pages 1–6. IEEE, 2017.

[67] Ivan Slivar, Lea Skorin-Kapov, and Mirko Suznjevic. Cloud gaming qoe models for deriving video encoding adaptation strategies. In *Proceedings of the 7th international conference on multimedia systems*, pages 1–12, 2016.

[68] Rajiv Soundararajan and Alan C Bovik. Video quality assessment by reduced reference spatio-temporal entropic differencing. *IEEE Transactions on Circuits and Systems for Video Technology*, 23(4):684–694, 2012.

[69] Kevin Spiteri, Ramesh Sitaraman, and Daniel Sparacio. From theory to practice: Improving bitrate adaptation in the dash reference player. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 15(2s):1–29, 2019.

[70] Kevin Spiteri, Rahul Urgaonkar, and Ramesh K Sitaraman. Bola: Near-optimal bitrate adaptation for online videos. In *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9. IEEE, 2016.

[71] Neil Stewart, Christoph Ungemach, Adam JL Harris, Daniel M Bartels, Ben R Newell, Gabriele Paolacci, Jesse Chandler, et al. The average laboratory samples a population of 7,300 amazon mechanical turk workers. *Judgment and Decision making*, 10(5):479–491, 2015.

[72] Gary J Sullivan, Jens-Rainer Ohm, Woo-Jin Han, and Thomas Wiegand. Overview of the high efficiency video coding (hevc) standard. *IEEE Transactions on circuits and systems for video technology*, 22(12):1649–1668, 2012.

[73] Yi Sun, Xiaoqi Yin, Junchen Jiang, Vyas Sekar, Fuyuan Lin, Nanshu Wang, Tao Liu, and Bruno Sinopoli. Cs2p: Improving video bitrate selection and adaptation with data-driven throughput prediction. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 272–285, 2016.

[74] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[75] Du Tran, Lubomir Bourdev, Rob Fergus, Lorenzo Torresani, and Manohar Paluri. Learning spatiotemporal features with 3d convolutional networks. In *Proceedings of the IEEE international conference on computer vision*, pages 4489–4497, 2015.

[76] Beth Trushkowsky, Tim Kraska, Michael J Franklin, and Purnamrita Sarkar. Answering enumeration queries with the crowd. *Communications of the ACM*, 59(1):118–127, 2015.

[77] Matteo Varvello, Jeremy Blackburn, David Naylor, and Konstantina Papagiannaki. Eyeorg: A platform for crowdsourcing web quality of experience measurements. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pages 399–412, 2016.

[78] Yilin Wang, Sasi Inguva, and Balu Adsumilli. Youtube ugc dataset for video compression research. In *2019 IEEE 21st International Workshop on Multimedia Signal Processing (MMSP)*, pages 1–5. IEEE, 2019.

[79] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004.

[80] Zhou Wang, Eero P Simoncelli, and Alan C Bovik. Multiscale structural similarity for image quality assessment. In *The Thrity-Seventh Asilomar Conference on Signals, Systems & Computers, 2003*, volume 2, pages 1398–1402. Ieee, 2003.

[81] Chen-Chi Wu, Kuan-Ta Chen, Yu-Chun Chang, and Chin-Laung Lei. Crowdsourcing multimedia qoe evaluation: A trusted framework. *IEEE transactions on multimedia*, 15(5):1121–1137, 2013.

[82] Mai Xu, Ali Borji, Ce Zhu, Edward Delp, Marta Mrak, and Patrick Le Callet. Introduction to the issue on perception-driven 360 video processing. *IEEE Journal of Selected Topics in Signal Processing*, 14(1):2–4, 2020.

[83] Francis Y Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and Keith Winstein. Learning in situ: a randomized experiment in video streaming. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 495–511, 2020.

[84] Hyunho Yeo, Youngmok Jung, Jaehong Kim, Jinwoo Shin, and Dongsu Han. Neural adaptive content-aware internet video delivery. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 645–661, 2018.

[85] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. A control-theoretic approach for dynamic adaptive video streaming over http. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 325–338, 2015.

[86] Junyong You, Touradj Ebrahimi, and Andrew Perkis. Attention driven foveated video quality assessment. *IEEE Transactions on Image Processing*, 23(1):200–213, 2013.

[87] Ke Zhang, Wei-Lun Chao, Fei Sha, and Kristen Grauman. Video summarization with long short-term memory. In *European conference on computer vision*, pages 766–782. Springer, 2016.

[88] Yulun Zhang, Yapeng Tian, Yu Kong, Bineng Zhong, and Yun Fu. Residual dense network for image super-resolution. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2472–2481, 2018.

[89] Yuanhuan Zheng, Di Wu, Yihao Ke, Can Yang, Min Chen, and Guoqing Zhang. Online cloud transcoding and distribution for crowdsourced live game video streaming. *IEEE Transactions on Circuits and Systems for Video Technology*, 27(8):1777–1789, 2016.

[90] Kaiyang Zhou, Yu Qiao, and Tao Xiang. Deep reinforcement learning for unsupervised video summarization with diversity-representativeness reward. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

A Dataset

Figure 19 provides screenshots and descriptions of the 16 source videos used in our dataset. Table 1 summarizes the test videos.

B Reliable QoE Crowdsourcing

We provide a few additional details on our crowdsourcing methodology.

Population bias of MTurk: As mentioned in §9, the per-chunk quality sensitivity could be biased by the population distribution of MTurkers. We confirm that about 43.8% (and 67.3%) of the received ratings come from MTurkers who participate in our survey only once (at most twice). This suggests that the pool of MTurkers is large enough to avoid small population bias, which corroborates our sanity-check results (§5) that on average MTurker quality ratings are strongly correlated with in-lab survey results.

Name	Genre	Length	Source dataset
(a) Basket1	Sports	3:40	LIVE-MOBILE
(b) Soccer1	Sports	3:20	LIVE-NFLIX-II
(c) Basket2	Sports	3:40	YouTube-UGC
(d) Soccer2	Sports	3:40	YouTube-UGC
(e) Discus	Sports	3:40	YouTube-UGC
(f) Wrestling	Sports	3:40	YouTube-UGC
(g) Motor	Sports	3:40	YouTube-UGC
(h) Tank	Gaming	3:40	YouTube-UGC
(i) FPS1	Gaming	3:40	YouTube-UGC
(j) FPS2	Gaming	3:40	YouTube-UGC
(k) Mountain	Nature	1:24	LIVE-MOBILE
(l) Animal	Nature	3:40	YouTube-UGC
(m) Space	Nature	3:40	YouTube-UGC
(o) Girl	Animation	3:40	YouTube-UGC
(n) Lava	Animation	3:40	LIVE-NFLIX-II
(p) BigBuckBunny	Animation	9:56	WaterlooSQOE-III

Table 1: *Summary of the test video set.*

Fast MTurker recruitment: While the MTurk platform cuts the overhead to publish our survey, if MTurkers sign up slowly, this can slow down the entire process. We take following steps to speedup the recruitment of MTurkers.

- *Competitive compensation:* We offer an hourly rate of $10, a competitive compensation on the MTurk platform — only 4% MTurkers are paid more than $7.25/hour [40], though we have not explored the impact of raising/lowering this rate. To prevent people from gaming the system by sitting on a job for too long, we pay each MTurker by the estimated amount of time needed to finish a survey (which is proportional to the total length of the videos per MTurker), rather than by how much time the MTurker actually spends. In practice, this only weeds out MTurkers who spend too much time on a survey.
- *Maintaining good reputation:* The MTurkers' signing-up speed also depends largely on the reputation of the publisher (*i.e.,* us), because MTurkers tend to sign up if the publisher historically has a low rejection rate. Thus, it is critical to be clear upfront about our study's rejection criteria. In the meantime, to keep our rejection rate low, we try to target reliable MTurkers by restricting ourselves to Master MTurkers (a common practice for publishers on MTurk [53]).

C Implementation

Automation of MTurk tests: We implement the pipeline shown in Figure 8 in Python (for the logic) and Javascript (for the video server). Given a source video, it first creates the

(a) Basket1 : A buzzer beater in a basketball game

(b) Soccer1 : A goal after a failed shoot

(c) Basket2 : A free throw followed by a one-on-one defense.

(d) Soccer2: presenting the scoreboard after a goal

(e) Discus: A man throwing a discus

(f) Wrestling: Two wrestling players

(g) Motor: Motor racing

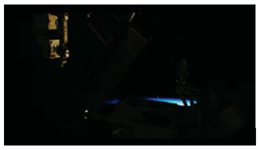
(h) Tank: A tank attacking a house

(i) FPS1: A first-person shooting game

(j) FPS2: A player robbing supplies

(k) Mountain: Mountain scene

(l) Animal: Warthogs that are bathing and grooming


(m) Space: A satellite taking pictures for our Earth

(o) Girl: A girl falling off the cliff

(n) Lava: A lava is waking up

(p) BigBuckBunny: A rabbit dealing with three tiny bullies

Figure 19: Summary of source videos in our dataset. They span four genres: sports (a - g), gaming (h - j), natural (k - m), and animation (n - p). They are compiled randomly from public QoE datasets: LIVE-MOBILE [36](a,k), LIVE-NFLX-II [21] (b, n), YouTube-UGC [78] (c - j and l - o); and WaterlooSQOE-III [31] (p).

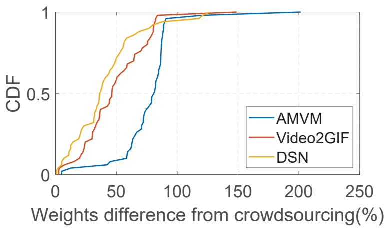


Figure 20: Chunk weight difference

rendered videos by adding specific low-quality incidents in the source video (via ffmpeg). It then uploads these videos to a video server, from which MTurkers later download the video. After that, it generates a unique link for this campaign and posts it on the MTurk website (the only step that requires manual intervention). MTurkers can join the test by clicking the link, which redirects them to our video server. Once an MTurker has rated all assigned videos, the server logs the ratings and notifies us. Once enough ratings are received, the server trains the per-chunk weights as described in §4.2.

Single survey procedure: As shown in Figure 21:

- (a) Each survey starts with the instructions and rejection criteria under which ratings of an MTurker will be rejected. Each MTurker is expected to read the instructions carefully.
- (b) The MTurker then watches an example video that includes a quality incident so that they know what their ratings should be based on.
- (c, d) After that, the MTurker is asked to watch a sequence of rendered videos (determined by the scheduler) and, after each video, rate the quality on a scale of 1-5.
- (d) Finally, the MTurker does an exit survey.

D More discussion on saliency models

Saliency models used in §2.3: We test four models in total, a traditional motion-based heuristic (AMVM [52]), a highlight detection model (video2GIF [39]), and a video summarization model (DSN [90], dppLSTM [87]). We used the pretrained models of Video2GIF (`https://github.com/gyglim/video2gif_code`), DSN (`https://github.com/KaiyangZhou/pytorch-vsumm-reinforce`), and dppLSTM (`https://github.com/kezhang-cs/Video-Summarization-with-LSTM`).

Saliency score vs. quality sensitivity: Although a variety of saliency models have been proposed, we argue that these visual heuristics can be misaligned with video quality sensitivity. For example, in the soccer video (Figure 1), the scene

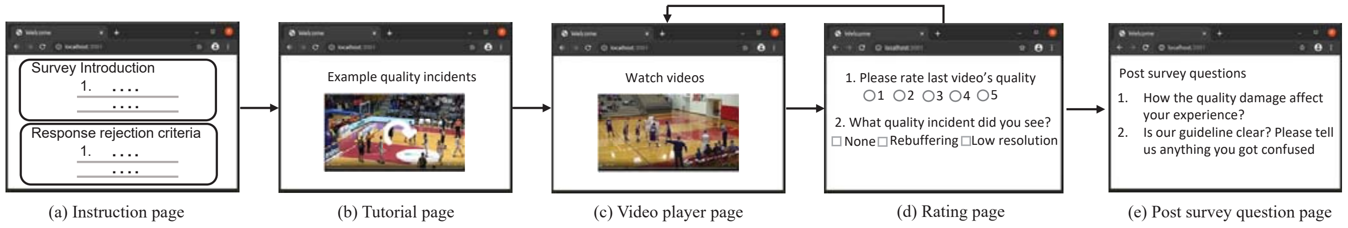

Survey Introduction
1. ....
....
Response rejection criteria
1. ....
....
Example quality incidents
Watch videos
1. Please rate last video's quality
○1 ○2 ○3 ○4 ○5
2. What quality incident did you see?
☐ None ☐ Rebuffering ☐ Low resolution
Post survey questions
1. How the quality damage affect your experience?
2. Is our guideline clear? Please tell us anything you got confused
(a) Instruction page
(b) Tutorial page
(c) Video player page
(d) Rating page
(e) Post survey question page

Figure 21: *A diagram of our QoE survey interface. In each survey, an MTurker is asked to rate K rendered videos; after watching each rendered video, an MTurker is asked to rate its quality on a scale of 1 (worst) to 5 (best).*

right before the goal is most quality sensitive, but the highlight detection models and motion-based heuristics we evaluated believe the scenes showing the audience are the most important (probably because they show more human movements). Video summarization models pick all diverse moments of a video, but many of them may not be quality-sensitive. For example, in the same soccer video, the video summarization models identify every shot, rewind, and celebration clip as important, but the users pay more attention to shots that might score a goal.

We also acknowledge the potential use of viewership information to detect video highlights. While the popularity of a chunk is closely related to highlights (or high interestingness scenes) in a video, as we found in §2.3 and §7.3, these incidents may not perfectly align with users' sensitivity to video quality. Below are two examples specifically regarding the potential misalignment between content popularity and quality sensitivity.

- Example 1: A less popular part of a video can still be quality-sensitive. The "animal" video in our dataset is a part of a video about wildlife in Africa. Although a more "popular" or "interesting" scenes is one where the lions chase antelopes, we find that users are still highly quality-sensitive in the scene where warthogs jump into a small pond for bathing.
- Example 2: A quality-sensitive part of a video may be a small fraction of a popular segment. One of the soccer videos in our dataset is a compilation of highlight moments from a long game, so all of its content is supposed to be "popular". However, we still see that there is heterogeneity in the sensitivity of its chunks.

GAIA: A System for Interactive Analysis on Distributed Graphs Using a High-Level Language

Zhengping Qian *Alibaba Group* Chenqiang Min *Alibaba Group* Longbin Lai *Alibaba Group* Yong Fang *Alibaba Group* Gaofeng Li *Alibaba Group*

Youyang Yao *Alibaba Group* Bingqing Lyu *Alibaba Group* Xiaoli Zhou *Alibaba Group* Zhimin Chen *Alibaba Group* Jingren Zhou *Alibaba Group*

Abstract

GAIA (GrAph Interactive Analysis) is a distributed system designed specifically to make it easy for a variety of users to *interactively* analyze big graph data on large clusters at low latency. It adopts a high-level language called Gremlin for graph traversal, and provides automatic parallel execution. In particular, we advocate a powerful new abstraction called Scope that caters to the specific needs in this new computation model to scale graph queries with complex dependencies and runtime dynamics, while at the same time maintaining the simple and concise programming model. GAIA has been deployed in production clusters at Alibaba to support a variety of business-critical scenarios. Extensive evaluations using both benchmarks and real-world applications have validated the effectiveness of the proposed techniques, which enables GAIA to execute complex Gremlin traversal with orders-of-magnitude better performance than existing high-performance engines, and at much larger scales than recent state-of-the-art Gremlin-enabled systems such as JanusGraph.

1 Introduction

Nowadays an increasing number of Internet applications generate large volume of data that are inherently connected in various forms. Examples include data in social networks, e-commerce transactions, and online payments. Such data are naturally modeled as graphs to encode complex relationships among entities with rich set of attributes. Unlike traditional graph processing that requires programming for each individual task, it is now very common for domain experts, typically non-technical users, to directly explore, examine, and present graph data in an interactive environment in order to locate specific or in-depth information in time.

As an example, consider the graph depicted in Figure 1, which is a simplified version of a real query employed at Alibaba for credit card fraud detection. By using a fake identifier, the "criminal" may obtain a short-term credit from a bank (vertex 1). He/she tries to illegally cash out money by forging a purchase (edge $2 \rightarrow 3$) at time t_1 with the help of a merchant (vertex 3). Once receiving payment (edge $1 \rightarrow 3$) from the bank (vertex 1) at time t_2, the merchant tries to send the money back (edges $3 \rightarrow 4$ and $4 \rightarrow 2$) to the "criminal" via multiple accounts of a middle man (vertex 4) at time t_3 and t_4, respectively. This pattern eventually forms a cycle ($2 \rightarrow 3 \rightarrow 4 \cdots \rightarrow 2$). Such fraudulent activities have become one of the major issues for online payments, where the graph could contain billions of vertices (e.g., users) and hundreds of billions to trillions of edges (e.g., payments). In reality, the entire fraudulent process can involve a complex chain of transactions, through many entities, with various constraints, which thus requires complex interactive analysis to identify.

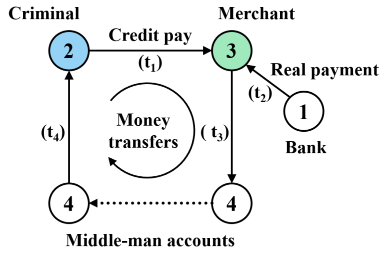


Figure 1: An example graph model for fraud detection.

Our goal is to make it easy for a variety of users to *interactively* analyze big graph data on large clusters at low latency. Achieving this goal requires a different distributed infrastructure than the popular batch-oriented big graph processing systems [4, 15, 16, 26, 39, 49] in two aspects:

Programming Model. Existing systems, including the most recent high-performance data engines such as Naiad [27], demonstrate that it is possible to scale well-known graph algorithms such as PageRank [5] and connected components [23] to large clusters. Even so, their programming interfaces all leave room for improvement for our target users, who typically lack the background on distributed computing or programming in general [13].

Memory Management. Existing systems[1] typically base their execution on the bulk synchronous parallel (BSP) model [44], where the computation proceeds iteratively, and

[1] Here, we focus on the distributed graph analytical systems. Other systems such as Neo4j, ZipG, and JanusGraph, etc. will be surveyed in Section 7.

```
Q1: g.V('account').has('id','2').as('s')
      .repeat(out('transfer').simplePath())
      .times(k-1)
      .where(out('transfer').as('s'))
      .path().limit(1)
```

Figure 2: An example Gremlin query for cycle detection.

in each iteration, all vertices in a graph will conduct the same computation , and send any updates along their edges to drive the computation of the next iteration. The BSP-based engines, however, are not suitable for interactive graph queries because of two reasons. Firstly, the interactive queries typically require maintaining application state along with the traversal paths to enable complex analysis [14, 37], which can grow exponentially with the number of iterations, and cause memory crisis in the underlying execution platforms. Secondly, in interactive environments, there are typically multiple queries sharing the limited amount of memory on the same set of machines, on which (a large part of) the input graph is cached in memory to provide required performance, making the above memory crisis a more critical issue.

In this work, we exploit Gremlin [37] to provide a high-level language for interactive graph queries. Gremlin is widely adopted by leading graph system vendors [1, 6, 21, 29, 30], which offers a flexible and expressive programming model to enable non-technical users to succinctly express complex traversal patterns in real-world applications. For example, one can write the above fraud-detection query in just a couple of lines using Gremlin, as shown in Figure 2 (which we explain in Section 3). In contrast, even common operations like cycle detection, which is a core part of the fraud-detection use case, is tricky to implement in existing graph systems [16, 36].

The flexibility of Gremlin mainly stems from nested traversal with dynamic control flow such as conditionals and loops. While attempting to scale Gremlin queries, we are immediately confronted with the challenges of resolving fine-grained data dependencies [10] with dynamic control flow [45]. Therefore, existing Gremlin-enabled, large-scale systems either adopt a *sequential* implementation in centralized query processing with data being pulled from a remote storage (such as JanusGraph [21] and Neptune [1]), or offer a limited subset of the language constructs (such as the lack of nested loops in [20]). In addition, GAIA must handle dynamics related to variations in memory consumption in an interactive context.

In this paper, we present a system, GAIA, that takes on the challenges of making Gremlin traversal work efficiently at scale with low latency. In particular, GAIA makes the following technical contributions.

- *Scope Abstraction.* We propose the Scope abstraction to allow GAIA to dynamically track fine-grained data dependencies in a Gremlin query. This enables Gremlin traversal to be modeled as a dataflow graph for efficient parallel execution with correctness guarantee.
- *Bounded-Memory Execution.* Leveraging the Scope abstraction, we are able to devise advanced optimizations in parallel graph traversal, such as bounded-memory execution and early-stop optimization, which lead to further runtime improvement and memory saving.
- *GAIA System.* We have developed a full-fledged distributed system, GAIA, and made it available at: https://github.com/alibaba/GraphScope/tree/main/research/gaia. An extended version of GAIA with enterprise features has been deployed in real production clusters at Alibaba to support a variety of business-critical scenarios. Extensive evaluations using both benchmarks and real-world applications have validated the effectiveness of the proposed techniques, which enables GAIA to execute complex Gremlin traversal with orders-of-magnitude better performance than existing engines, and at much larger scales than the state-of-the-art Gremlin-enabled systems such as JanusGraph.

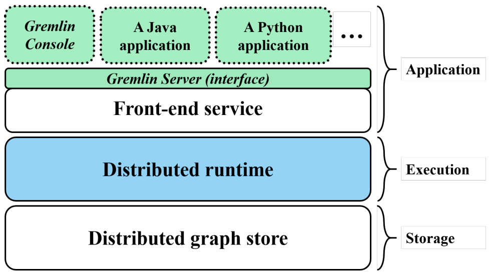


Figure 3: GAIA system architecture.

2 System Architecture

GAIA is a full-fledged, in-production system for interactive analysis on big graph data. Achieving this goal requires a wide variety of components to interact, including software for cluster management and distributed execution, language constructs, and development tools. Due to space limit, we highlight the three major layers that are sufficient to understand this paper, namely application, execution, and storage, in Figure 3, and give an overview to each of them below.

Apache TinkerPop [3] is an open framework for developing interactive graph applications using the Gremlin query language [37]. GAIA leverages the project to supply the application layer. GAIA implements the Gremlin Server [18] interface so that the system can seamlessly interact with the TinkerPop ecosystem, including development tools such as Gremlin Console [17] and language wrappers such as Java and Python.

The GAIA execution runtime provides automatic support for efficient execution of Gremlin queries at scale, which constitutes the main contribution of this paper. Each query is compiled by the front-end service into a distributed execution plan that is partitioned across multiple compute nodes for parallel execution. Each partition runs on a separate compute node, managed by a local executor, that schedules and executes computation on a multi-core server.

The storage layer maintains an input graph that is hash-partitioned across a cluster, with each vertex being placed

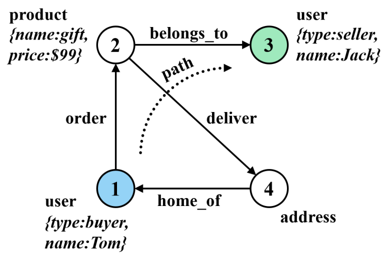


Figure 4: An example “e-commerce” property graph.

together with its adjacent (both incoming and outgoing) edges and their attributes. In this paper, we assume that the storage is coupled with the execution runtime for simplicity, that is each local executor holds a separate graph partition. In production, we implement a distributed graph storage with index and cache features, decoupled from the execution, that supports real-time updates with snapshot isolation (similar to Kineograph [11]), which allows users to query fast-changing graphs with consistency guarantee. Furthermore, GAIA provides multiple options for fault tolerance using checkpoints, replication, and/or relying on a Cloud storage. Production details are outside the scope of this paper.

3 Programming with GAIA

GAIA is designed to faithfully preserve the programming model of TinkerPop [3], and as a result it can be used to scale any existing TinkerPop applications to large compute clusters without any modification. In this section, we provide a high-level view of the programming model, highlighting the key concepts including the data model and query language.

Gremlin [37] enables users to define ad-hoc traversals on property graphs [2]. A property graph is a directed graph in which vertices and edges can have a set of properties. Every entity (vertex or edge) is identified by a unique identifier (`ID`), and has a (`label`) indicating its type or role. Each property is a key-value pair with combination of entity `ID` and property name as the key. Figure 4 shows an example property graph. It contains `user`, `product`, and `address` vertices connected by `order`, `deliver`, `belongs_to`, and `home_of` edges. A path following vertices $1 \to 2 \to 3$, shown as the dotted line, indicates that a buyer “Tom” ordered a product “gift” offered by a seller “Jack”, with a price of “$99”.

In a Gremlin traversal, a set of *traversers* walk a graph according to particular user-provided instructions, and the result of the traversal is the collection of all halted traversers. A traverser $T = (l, P)$[2] is the basic unit of data processed by a Gremlin engine. Each traverser maintains a location l that is a reference to the current vertex, edge or property being visited, and (optionally) the path history P. For example, consider a traversal which starts from vertex 1 (with only one traverser at the location of vertex 1), follows outgoing edges, and reaches its 2-hop neighbors in Figure 4. A possible intermediate result can be a collection of a single traverser located at vertex 2 with the corresponding path history. The final result is a collection of two traversers, located at vertex 3 and 4, respectively, with different paths, $1 \to 2 \to 3$ and $1 \to 2 \to 4$.

Nested traversal is another key concept in Gremlin. It allows a traversal to be embedded within another operator, and used as a function to be invoked by the enclosing operator for processing input. The role and signature of the function are determined by the type of the enclosing operator. For example, a nested traversal within the `where` operator acts as a predicate function for conditional filters, while that within the `select` or `order` operator maps each traverser to the output or ordering key for sorting the output, respectively.

Nested traversal is also critical to the support for loops, which are expressed using a pair of the `repeat` and `until`/`times` operators. A nested traversal within the `repeat` operator will be looped over until the given break predicate is satisfied. The predicate (or termination condition) is defined within the `until` operator, applied to each output traverser separately from each iteration. The `times` operator can also terminate a loop after a fixed number of k iterations.

Example 3.1. *Figure 2 shows a Gremlin query Q1 for the motivating example in Section 1 that tries to find cyclic paths of length k, starting from a given account. First, the source operator* `V` *(with the* `has` *filter) returns all the* `account` *vertices with an identifier of “2”. The* `as` *operator is a* modulator *that does not change the input collection of traversers but introduces a name (s in this case) for later references. Second, it traverses the outgoing* `transfer` *edges for exact $k-1$ times, skipping any repeated vertices (by the* `simplePath` *operator). Third, the* `where` *operator checks if the starting vertex s can be reached by one more step, that is, whether a cycle of length k is formed. Finally, for qualifying traversers, the* `path` *operator returns the full path information. The* `limit` *operator at the end indicates only one such result is needed.*

4 Compilation of Gremlin

GAIA compiles a Gremlin query into a dataflow graph, where each vertex (operator) performs a local computation on input streams from its incoming edges and produces output streams to its outgoing edges, and can optionally maintain a state. The input graph is modeled as a read-only state shared by all the dataflow operators. We map each Gremlin operator onto a dataflow operator, and the collections of traversers as data streams. In the following, we will use the term traverser interchangeably with data. Figure 5(b) shows an example dataflow graph corresponding to the following Gremlin query (Q2) that conducts a 2-hop traversal followed by an aggregation that counts the total number of traversed paths.

```
Q2: g.V(2).out().out().count()
```

We introduce source operators as special drivers that generate output only from the input graph to drive the rest of the

[2]In [37], a traverser is modelled as a 6-tuple set, while we include necessary elements to understand this paper.

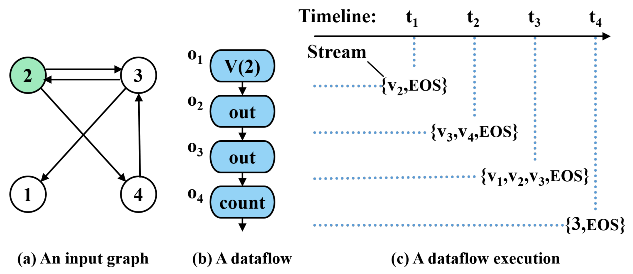


Figure 5: Dataflow graph and execution for query Q2.

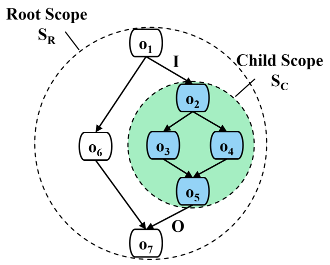


Figure 6: Dataflow and scope example: the filled circle highlights a scope with input stream I and output stream O.

dataflow computation (e.g., `V(2)`). We use sink operators to denote those that generate output streams for the computation to be consumed elsewhere (e.g., `count`). Since Gremlin imposes no restrictions on the execution order of traversers, we can pack a segment of traversers to a same operator into a batched input and schedule the computation at a coarse granularity for efficient execution.

To preserve the operator semantics for barriers, we insert an End-of-Stream (or EOS) marker at the end of the output streams of each source operators, as a special punctuation event that asserts the completeness of output. The EOS markers will be propagated through the dataflow, layer by layer, so that any downstream operators can be notified on the completeness of their inputs by waiting to collect those markers.

Example 4.1. *Figure 5(c) illustrates the progression of the dataflow execution of Q2 against the input graph in Figure 5(a). o_1 generates a data stream of {(v_2,∅), EOS} as output, where v_2 denotes the vertex with* `ID` *2. Note that the path history has been pruned (and omitted later) as the downstream operators do not need it. o_2 consumes v_2, generates output {v_3, v_4}, and finally propagates EOS to its output. Subsequently, o_3 outputs {v_1, v_2} after consuming v_3, and {v_3, EOS} for the rest of its input. Finally, o_4 outputs the counting of {3} - it can do so as the EOS marker has been received. The dataflow thus terminates.*

4.1 Challenges in Compiling Nested Traversal

Many of the salient features of Gremlin such as dynamic control flow rely on nested traversal, which introduces additional complexity to the above design. Let's look into another query Q3 slightly amended from Q2, in which a segment of operators (`out().count()`) is nested within a `select`-projection.

```
Q3: g.V(2).out()
      .select('neighbor_count')
      .by(out().count())
```

Given a set of vertices $N(v_2)$ as the outgoing neighbors of a vertex v_2, the query asks to count the number of k-hop paths starting from each vertex $u \in N(v_2)$ (let $k = 1$ for simplicity), and output pairs of (u, # paths starting from u). In this example, each input traverser that represents a vertex of $N(v_2)$ does its own computation (of the counting of paths), namely at a fine granularity. In other words, the `count` operation has to be executed separately for each vertex $u \in N(v_2)$.

We define a *context* as an execution environment for a dataflow that includes a unique (possibly empty) state for its computation. Without nested traversal (and/or dynamic control flow), all computation of each Gremlin operator, and the whole dataflow, can run in a single context. For example, in query Q2, only `count` maintains a state (for partial counting) - there is only one such state needed to count all traversed paths. With nested traversal, this property no longer holds as a stateful operator in a nested traversal can dynamically demand the separation of contexts. For example, in query Q3, due to the semantics of `select`, there must be an individual state (context) maintained for each vertex $u \in N(v_2)$ in order to produce correct results.

One may argue that the above example is not so hard to tackle. However, this is just a simplest example involving sub-traversals in Gremlin. Such context separation is also important in dynamic control flow such as loop, in which each iteration must run separately from another. One can even encounter sub-traversals involved with arbitrary combination of complex structuring constructs, making the system design uncontrollably complex. In addition, the number of separate contexts required for the correct execution of a *single* Gremlin traversal can be proportional to that of the intermediate traversers (e.g., `select` in query Q3), which can be of millions to billions in our case. While it is possible to dynamically create physical contexts as in [45], doing so at such a fine granularity for Gremlin is clearly infeasible in practice.

4.2 The Scope Abstraction

To address the issues posted by Gremlin traversal, we propose the Scope abstraction to help emancipate the system from the need of maintaining context information.We first define the concept of a Scope.

Definition 4.1. A Scope is a subgraph in a dataflow (sub-dataflow) that satisfies the following condition: for any operators o_1 and o_2 in the sub-dataflow and any operator o in the dataflow, o must also be in the sub-dataflow if o is on a directed path from o_1 to o_2.

A Scope has the same logical structure (and function) as a dataflow operator, which can thus be reduced to one virtual "operator" in the dataflow graph. Naturally, we refer to a Scope context, as the context of its enclosed operator. It is allowed that a Scope S_p contains another sub-dataflow as a nested Scope S_c as long as it satisfies the definition of Definition 4.1. S_p is called the parent Scope of S_c, and S_c is accordingly the child Scope of S_p. The whole dataflow is a special Scope that we call as *root* Scope. The dataflow regarding the nested relationships of Scopes naturally form a hierarchical structure.

Example 4.2. *In the dataflow graph as shown in Figure 6, the sub-dataflow comprised of o_2, o_3, o_4 and o_5 (as well as all their edges) is a* Scope *S_c (as highlighted) and can be reduced to one operator with I as its input stream, and O as its output stream. The whole dataflow is the root* Scope, *which is the parent* Scope *of S_c.*

As we mentioned earlier, it is costly to create physical dataflow operators as in [45] for a Gremlin query that potentially requires a separate context for each data item. We therefore propose the Scope abstraction to handle the separation of execution contexts in a Scope in a more light-weighted manner. A Scope abstraction consists of three primitives, namely `Enter`, `Exit`, and `GoTo`, and the interface of Scope policy. Specifically, `Enter` forwards a data item from a parent Scope[3] to a child Scope, while `Exit` sends data item back to a parent Scope. As `GoTo` is primarily used for loop control flow, we will introduce it in Section 4.3.

The Scope policy is installed by the compiler on each `Enter` and `GoTo` primitives to fulfil different context-switch mechanisms. Logically, we use a sequence number as context identifier to identify an execution context in a Scope, the Scope policy contains the following interfaces (their implementations are in Section 4.4):

- `CreateOrOpen(Data:`e`,CtxID:`s`)`: To create a new isolated context for the input data e, or open an existing context uniquely identified by s.
- `GetContext(Data:`e`)`: To obtain the context identifier of the data e.
- `Complete(Data:`e`,CtxID:`s`)`: To mark that there will be no more data for the context of s, after receiving e.

As an example, we present a built-in scope policy called `CONTEXT_PER_ENTRY` (more policies will be introduced as follows). `CONTEXT_PER_ENTRY` creates a new context for each input data. Let *seq* be a sequence number, initialized to 0. For each input e, the `CONTEXT_PER_ENTRY` policy first applies `CreateOrOpen(`e, *seq*`)` to create a new context for e. It then immediately calls `Complete(`e, *seq*`)` to indicate that there will be no more data for the context of *seq*. Finally, the policy increments *seq* by 1 such that any future data will enter a different context. In the following, we will detail how the Scope abstraction facilitates the compilation of a Gremlin query with nested traversals.

[3]It is more precisely a context of the Scope, while we refer to it as Scope for short.

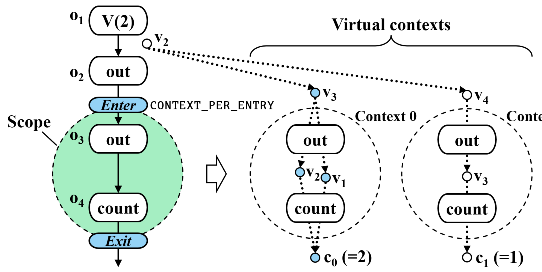


Figure 7: An example Scope execution with separate contexts.

4.3 Compilation of Gremlin using Scope

Compilation of a Gremlin query without dynamic control flow or nested traversal is as similar to that in existing systems [41, 46, 47], we do not elaborate on it further. Both dynamic control flow and nested traversal introduce sub-traversals in a Gremlin query. GAIA compiles each such sub-traversal into a Scope enclosed by a pair of `Enter` and `Exit` primitives (can be multiple of them nested within each other). The Scope abstraction handles the context separation in a unified way. Due to space limit, this section presents the compilation process of three representative Gremlin operators (`select`, `where`, and `repeat`) to highlight the common pattern of using the Scope abstraction.

Example 4.3. *Figure 7 illustrates an example that* GAIA *compiles the query Q3 (Section 4.1) into a dataflow using* Scope, *in which the* `select`*-projection introduces a* Scope *that encloses the sub-traversal of* `out().count()`*. As there requires a separate execution context for each data entering the* Scope, GAIA *installs a* `CONTEXT_PER_ENTRY` *policy on the* `Enter`*. This way, each data can drive their own computation of* `out().count()` *in isolation, without concerning about the context separation as posted in Section 4.1.*

Dynamic control flow such as `where`-conditionals and `repeat`-loops introduce addition complexity, as presented in the following query:

```
Q4: g.V(2).as('s')
      .repeat(out().simplePath())
      .times(k-1)
      .where(out().eq('s'))
      .path().limit(1)
```

We next focus on the compilation of these constructs, inspired by TensorFlow [45]. However, unlike [45], they can be applied to a much finer granularity of each individual traversal path in Gremlin. This is enabled by the Scope abstraction. We further introduce the following primitive operators:

- `Copy` takes in a data e and outputs two identical data.
- `Switch` takes a data from its input and a boolean value p, and forwards the data to either the `True` branch of d_t or `False` branch of d_f, based on the predicate p.
- `Merge` accepts two input streams and merges them into one single output stream.

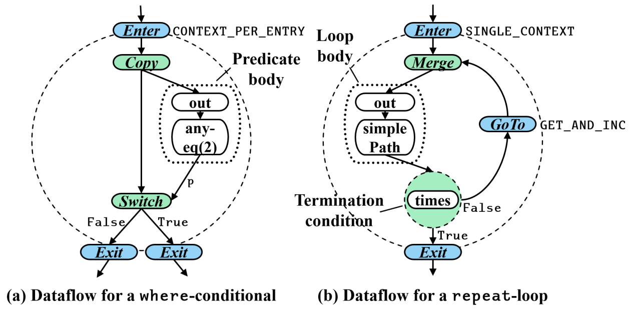


Figure 8: Compilation of control-flow constructs.

Conditional. Figure 8(a) shows an example of compiling a `where`-conditional. Conceptually, the `where` statement determines whether a data, while arriving at `where`, will continue to traverse, if the sub-traversal is evaluated to be true, or be abandoned otherwise. As the conditional check happens for each individual data, a `CONTEXT_PER_ENTRY` policy will be installed by the compiler in the `Enter` while entering the `where` Scope. Each data enters the `Copy`, where one data goes into the predicate body to drive the sub-traversal, and the other data goes to the `Switch`. Based on returned boolean value of the predicate body, the data with a `True` predicate will get out of the Scope via the `True` branch, and the data with a `False` predicate will go via the `False` branch (and get discarded if not further used).

Loop. We first introduce other two built-in Scope policies.

- `SINGLE_CONTEXT` policy calls `CreateOrOpen`$(e,0)$ for each data e indicating that they all enter one context of 0. It calls `Complete`$(e,0)$, if and only if $e = EOS$.
- `GET_AND_INC` policy first calls `GetContext`(e) to obtain the context of e as seq. Then it increases seq by 1 as seq', and calls `CreateOrOpen`(e, seq') to enter the new context. It finally calls `Complete`(e, seq'), if and only if $e = EOS$.

Figure 8(b) illustrates the compilation of `repeat`-loop. The compiler installs the `SINGLE_CONTEXT` policy on the `Enter` that forwards a data into the loop Scope, with a new context of 0, or, in the 0-th iteration. Additionally, it installs the `GET_AND_INC` policy on the `GoTo`. The `GoTo` primitive, as mentioned earlier in Section 4.2, is used to explicitly switch the context of data. Specifically in a loop, it leverages the `GET_AND_INC` policy to allow any data produced from current loop context to get switched to the next iteration. Naturally, the context identifier can now serve as the loop count. The loop body compiles any sub-traversal that will be run iteratively. Eventually, the data in the loop context will go though a conditional Scope as we have discussed above. This conditional Scope checks whether a termination condition is satisfied (such as arriving at the maximum iteration by `times`, or traversing to a certain vertex by `until`). The data with a `False` predicate is able to exit the loop, while the data with a `True` predicate will proceed to the next iteration as a feedback data stream via the `GoTo`, updating its context via the `GET_AND_INC` policy to indicate entering the next iteration. Note that a context must have been created or opened for each data e in a Scope, and thus `GetContext`(e) can be safely called. The feedback data will be eventually merged back to the input (of the sub-traversal) to drive the computation of next iteration.

4.4 Implementing Scope

It is challenging to implement Scope both correctly and efficiently. While it is always possible to create physical dataflow operators for each separate context, due to potentially unbounded number of such contexts in graph traversals (as described in Section 4.1), this is clearly infeasible in practice. GAIA instead dynamically tracks dependencies among input, output, and internal states for each operator in a dataflow.

GAIA labels each traverser with a tag, which is a k-ary vector of context identifiers, denoted as $T = [s_1, s_2, \dots, s_k]$[4], where the dimension indicates the level of potentially nested Scope. The root Scope is by default identified by a tag of $[\,]$. We define the following operations on a tag T:

$T[\wedge]$	To get the last context identifier of T.
$T[\wedge \to s]$	To replace the last context identifier of T with s.
$T[+1]$	To increase the dimension of T by 1, with the new slot filled with a $\emptyset$.
$T[-1]$	To reduce the dimension of T by 1.

From now, each data e will be tagged as $(T;e)$, which allows the system to be aware of the Scope and its different contexts. The primitives of `Enter` and `Exit`, and the interface functions in the Scope abstraction will explicitly modify the tag, as follows.

- `Enter` increases the dimension of the tag by 1 to indicate entering a Scope, as $(T[+1];e)$.
- `Exit` reduces the dimension of the tag by 1 to indicate leaving a Scope, as $(T[-1];e)$.
- `CreateOrOpen`$((T;e),s)$ return a newly tagged data with the last context identifier of T replaced as s, as $(T[\wedge \to s];e)$.
- `GetContext`$((T;e))$ returns the last context identifier of T, as $T[\wedge]$.
- `Complete`$((T;e),s)$ produces a tagged EOS marker to indicate the end of current context s, as $(T[\wedge \to s];EOS)$.

Such data tagging is automatically handled by GAIA system, and is transparent to any user interface. For the primitive operators introduced in Section 4.3, they do not need to worry about tags, and hence can still treat the tagged data as a "normal" data. For a computing operator o (with the logic f_o) in Gremlin, such as `out` and `count`, GAIA handles the computation as follows. It first extracts the actual data e, and apply the computation logic $f_o(e)$. The computation will generate a set

[4] Such tagging appears to be similar to the timestamps in Naiad [27], but it is used for dependency tracking in GAIA, without any physical meaning of event time as in Naiad [27].

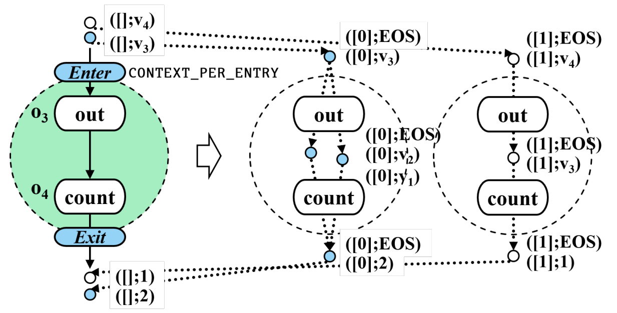


Figure 9: An execution with dynamic dependency tracking.

of traversers Ω, and potentially modify a state τ of the operator. Then for all $e' \in \Omega$, GAIA re-tags e' with T and sends it to the output stream. To handle any stateful computation, GAIA maintains an associated map with tag T as the key and state τ as the value, so that it can operate on the right state from different execution contexts transparently, as if the operator runs in isolation.

Example 4.4. *Figure 9 shows the above process for the dataflow in Figure 7. Initially, it accepts and computes inputs* $\{([\,];v_3),([\,];v_4)\}$ *(path history is omitted) from the parent* Scope *context. The* `Enter` *of the* `select` Scope *turns the inputs to* o_3 *as* $\{([0];v_3),([0];EOS),([1];v_4),([1];EOS)\}$ *according to the* `CONTEXT_PER_ENTRY` *policy. Next,* o_3 *outputs* $\{([0];v_1),([0];v_2),([1];v_3)\}$. *Note that EOS is omitted for now.* o_4 *can then maintain a hash table with the tag as key and the partial count as value. Finally, while* o_4 *receives the EOS for the corresponding context, it can output the results as* $\{([0];2),[[1];1]\}$. *The* `Exit` *restores the tags from* o_4*'s output and generates* $\{([\,];2),([\,];1)\}$.

Handling EOS Markers. An EOS marker can be introduced by both the source operator and the `Complete` function inside a Scope (`Enter`). An EOS marker can go through any computing operator without doing any de-facto computation, while it must be carefully handled in the primitive operators, especially `Enter` and `Exit` with the presence of Scopes.

Given a Scope, we call an EOS marker produced from outside the Scope as external EOS, and an EOS produced inside the Scope as internal EOS. An external EOS marks the termination of a context in the parent Scope, and must exit back to the parent Scope. Conversely, an internal EOS fulfills the same purpose only in the current Scope, and should only be propagated within. It is thus critical to differentiate the semantics of the EOS markers in a Scope. To do so, we implement the policy installed on the `Enter` to not call `CreateOrOpen` on the external EOS marker, which can then be recognized as a ∅ context. In the `Exit`, GAIA only allows the external EOS to leave the Scope.

Recall that `Switch` is another primitive operator used in conditional and loop Scope that delivers a data to either branch based on the predicate. The EOS marker, however, will always be propagated to both branches. In the loop Scope, the external EOS, once propagating through the nested conditional, will be held in the `Exit` of loop, and only released after the system verifies that all loop contexts terminate (using known techniques [45]). For the internal EOS, it will be tagged as the other data in the `GoTo`. As long as any data with a tag T is propagated to the next iteration, the EOS with T must also be propagated over to `GoTo` (meaning that the associated loop context has not terminated); otherwise, it will leave the loop Scope and get discarded.

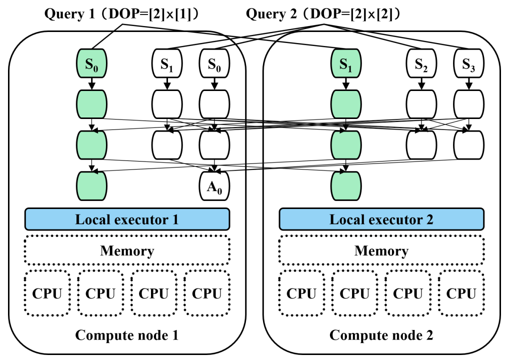


Figure 10: Distributed execution on two compute nodes.

5 Distributed Execution

GAIA runs queries via a set of worker processors in a shared-nothing cluster, where each worker executes a fragment of the computation. For each query, GAIA first compiles it into a dataflow graph using the techniques in Section 4, then it partitions the source operator in the dataflow according to the input graph partition, with the segment of operators that follow the source replicated across the set of workers. A local executor manages the computation on each worker by scheduling the operators to run. It starts from the source operator and repeatedly executes the following *ready* operators. Here, an operator is ready if all its inputs are available to consume. For now, GAIA requires the users to manually specify a degree of parallelism (DOP) for a query upon submission. We leave it as an interesting future work to automatically derive the DOP. According to the DOP, the local executor parallelizes the operators to execute on the multiple CPU cores, as illustrated in Figure 10. While GAIA can support multiple concurrent queries, we focus on single query processing in this paper.

5.1 Bounded-Memory Execution

Graph traversal can produce paths of arbitrary length, leading to memory usage growing exponentially with the number of hops. Although it is very common for Gremlin queries to terminate with a top-k constraint and/or aggregate operation, such an explosion of *intermediate* results can often lead to memory crisis, especially in an interactive environment with limited memory configuration. While several techniques exist for alleviating memory scarcity in dataflow execution, such as backpressure and memory swapping, they cannot be directly applied in GAIA due to potential deadlocks [25, 31] and/or high (disk I/O) latency. To ensure *bounded-memory execution* without sacrificing performance (parallelism), the local executor in GAIA employs a new mechanism for dataflow execution, called *dynamic scheduling*.

Dynamic Scheduling. For each operator, GAIA packs a segment of consecutive traversers in a stream into a single batch, and such a batch constitutes the finest data granularity for communication and computation. A *task* can be logically viewed as the combination of an operator and a batch of data to be computed. GAIA dynamically creates tasks corresponding to each operator when there is one or more batches available from all its inputs[5]. The local executor maintains all the tasks in a same scheduling queue to share resources.

We implement our own memory allocator that will report the total amount of memory used (for each query) so that the executor can watch the memory consumption. When it reaches a predefined threshold (high-watermark), the executor stops scheduling more tasks from the queue, except for those corresponding to the sink operators that will be sent to the clients. The executor resumes scheduling tasks when the memory consumption drops below another predefined threshold (low-watermark). It is possible that a single task (with a high-degree vertex) execution may produce too much output to exhaust the memory. To avoid this issue, we suspend a task when its output data exceeds a *capacity* bound, and resume it after the data has been consumed.

Data shuffling between two machines may introduce dependencies between their task scheduling. For example, a task can cause another executor to run into low memory, if it sends too much data to that executor. In this case, the sender task will be suspended until the receiver executor recovers from low memory. We implement a mechanism to send backpressure signals across network to allow cooperation of schedulers.

An execution of a dataflow graph with cyclic edges can potentially deadlock using bounded memory. In the specific context of graph traversal, this can be caused either by *infinite loops* such as traversing along a cyclic path without termination, or *inappropriate scheduling* such as buffer exhausted by a BFS-prioritized traversing (will be discussed later) that prevents downstream or sink operators from being scheduled to drain the buffered intermediate data. To address infinite loops, we apply a configurable limit N of the maximum number of iterations allowed in a loop (with a small buffer reserved for each iteration), and let the `GoTo` declare a deadlock when the limit N is reached. Once a deadlock is detected, the corresponding query is terminated with a clear error message. To handle inappropriate scheduling, we adopt a hybrid traversal strategy as described below.

Hybrid Traversal Strategy. As mentioned above, the memory crisis mainly stems from the intermediate paths, and therefore the traversal strategies can greatly impact the memory usage. There are two typical traversal strategies, namely (breadth-first-search) BFS-like traversal and (depth-like-search) DFS-like traversal. BFS-like traversal can better utilize parallelism, while it may produce data all at once that drives high the memory usage. On the contrary, DFS-like traversal tends to consume much less memory, while it may suffer from low parallelism. With this observation, we propose to allow the local executor to schedule tasks with priorities according to its topological order (i.e. the traversal depth) in the dataflow. Specifically, the executor can schedule the tasks located at the same order with higher priority for a BFS-like traversal, and prioritize those at downstream to follow a DFS-like traversal. Note that such strategy works naively for all the tasks but those in a loop context, where the traversers from different iterations may be executed in the same task. To resolve this, we let the operator's buffer reorder (and group) traversers by their iteration markers (obtained from the context identifier) before packing them into batches. This makes sure that we can prioritize tasks unambiguously even within loops. To balance the memory usage with the performance (parallelism), GAIA by default adopts a hybrid traversal strategy, that is, it uses BFS-prioritized scheduling as it has better opportunities for parallelization, and automatically switches to DFS-prioritized in case that the current operator arrives at the memory bound.

[5]The only exception is `Merge`, which is ready to run when data become available at any of its inputs.

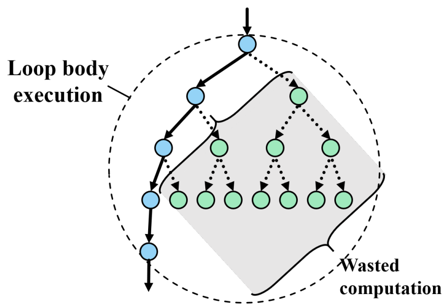


Figure 11: A loop execution with wasted computation.

5.2 Early-Stop Optimization

Traversing all candidate paths fully is often unnecessary, especially for interactive queries with dynamic conditions running on diverse input graphs. For example, in the following query Q5, only the first k results are needed.

```
Q5: g.V(2).repeat(out().simplePath())
     .times(4).path()
     .limit(k)
```

This leads to an interesting tradeoff between parallel traversal and wasted computation, as further illustrated in Figure 11. It shows an example run of query Q5 with $k = 1$. The circle denotes the traversal specified by the `repeat`-loop. Assume we have enough computation resource (CPU cores), the paths can be explored in a fully parallel fashion. However, once a 4-hop path is found, all the remaining parallel traversal will be no longer required.

For real-world queries on large graph data, such wasted computation can be hidden deeply in nested traversals (e.g., a predicate that can be evaluated early from partial inputs) and significantly impact query performance. While avoiding such wastage is straightforward in a sequential implementation, it is challenging to do so for a fully-parallel execution.

Normally, the execution of a particular context terminates when the EOS markers arrive at all the exits (from this context), including any `Exit` or `GoTo`. In the above example, an operator (e.g., `limit`) can actually terminates early after producing k outputs, before receiving any input EOS markers. GAIA further allows `Complete`$((T;e),s)$ to be called by any operators in a Scope to explicitly produce a tagged EOS marker (for current context s) to indicate the completeness of its output (after sending e downstream). However, this alone does not prevent upstream computation from continuing producing output that is no longer required and thus the corresponding computation is wasted.

To minimize such wastage, when a `Complete` is issued by an operator, it creates a *cancellation token* associated with the same context tag that is sent backward along input edges to its upstream operators within the Scope. The token serves as a signal for receiving operators to clear any unsent output data and immediately insert an EOS marker for the particular output stream. If such a token has been received from all output streams, the operator further propagates it to its own upstream operators, recursively, until it encounters the `Enter` for the same Scope. Such cancellation notification is implemented at a system level by GAIA. Due to space limit, We omit further details on propagation of cancellation tokens in any child Scope and/or through the `GoTo` to its dependent, previous contexts. We validate that such early-stop optimization can significantly improve query performance in Section 6.

6 Evaluation

6.1 Experimental Setup

Datasets. We generate 5 graph datasets as shown in Table 1 for experiments using Linked Data Benchmark Council (or LDBC) data generator [12], where G_x denotes that the graph is generated with $scale = x$. We use G_{300} as the default dataset if not otherwise specified. Note that G_{1000} is the largest data graph that LDBC can generate.

Table 1: The LDBC datasets.

Name	# vertices	# edges	Agg. Mem.
G_1	3M	17M	4GB
G_{30}	89M	541M	40GB
G_{100}	283M	1,754M	156GB
G_{300}	817M	5,269M	597GB
G_{1000}	2,687M	17,789M	1,960GB

Queries. For comparison, we consider graph queries from the Social Network Benchmark defined by LDBC [12] to model industrial use cases on a social network akin to Facebook. We choose 10 out of 14 *complex read* queries (denoted as CR-1...14) from LDBC's Interactive Workload[6].

In addition, the cycle-detection query Q6 is considered: given m (by default 10) starting nodes in V, it traverses from V via at most k (by default 4) hops, and returns those vertices among V that can form at least n (by default 10) cycles along the traversal. We modify the query based on the production query as shown in Figure 1 to align with the LDBC data. This query also shows the functionality of *prepared statement* ("Discussion", Section 4.3) enabled by the Scope abstraction, which wraps multiple starting vertices into one query.

The driver client provided by LDBC is modified to run each of the queries 20 times from a set of randomly selected parameters. Average query latency is reported.

Configurations. In the following experiments, we by default warm up all the systems to keep the computation-relevant data in memory. We do this to focus on benchmarking the computing engine instead of storage access.

All the queries have been implemented using Gremlin for all systems except Neo4j (using Cypher officially), with correctness cross-verified. The compiling time of these queries in our system is typically within 1ms, which is negligibly small compared to the query runtime, and will be ignored thereafter. We allow each query to run for at most 1 hour, and mark an OT if a query can not terminate in time. We manually configure the degree of parallelism (DOP) while running each query in GAIA. In the following, we denote DOP = $[x] \times [y]$ for running y threads in x machines.

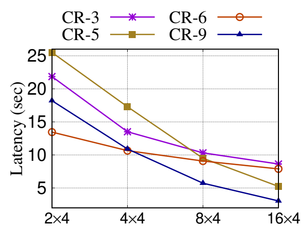
We compare GAIA with the systems in Table 2. While Neptune [1] is another popular Gremlin-enabled graph database, we do not benchmark it as it is only available in AWS, and its performance is similar to JanusGraph as shown in [42]. Timely [43] is the publicly available implementation of Naiad [27]. Plato [32] is an open-sourced implementation of Gemini [49] (Gemini does not support (de)serializing vector-like data for sending paths across network). We implement GAIA using Rust [38], and are working on open-sourcing the engine and storage.

Table 2: The evaluated systems.

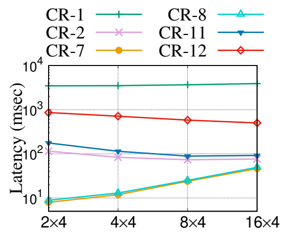
System	Version
TinkerGraph [3]	3.4.1
Neo4j-Community [29]	3.5.8
OrientDB [30]	tp3-3.0.15
JanusGraph [21]	0.4.0-hadoop2
Timely [43]	latest release in Github
Plato [32]	latest release in Github

We deploy a cluster of up to 16 machines, and each machine configures one 24-core Itel(R) Xeon(R) Platinum 8163 CPUs

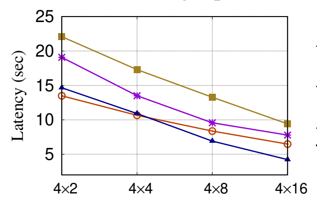
[6]The remaining queries are either too simple (such as simple point-lookup queries) or rely on user-defined logic (such as CR-4, 10, 13, 14), which is not supported by other popular TinkerPop-based systems.



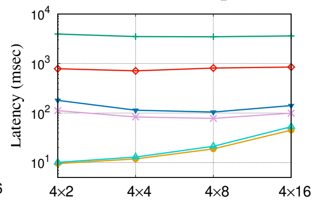
(a) Scale-out: large queries.



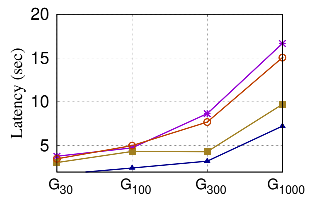
(b) Scale-out: small queries.



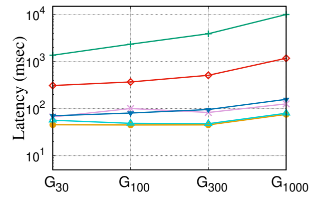
(c) Scale-up: large queries.



(d) Scale-up: small queries.



(e) Data size: large queries.



(f) Data size: small queries.

Figure 12: The scalability experiment.

at 2.5GHz and 512GB memory. The servers are connected through 25Gbps network.

6.2 Scalability

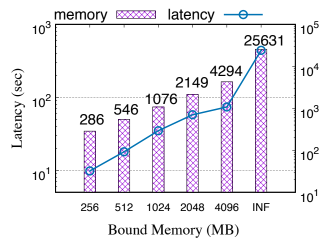
To the best of our knowledge, GAIA is the only system that can handle Gremlin queries at scale. In this experiment, we study the scalability of GAIA while running all LDBC queries. We divide these queries into two groups based on their runtime to better present the result: (1) large queries CR-3, 5, 6 and 9; (2) small queries CR-1, 2, 7, 8, 11, 12.

Scale-out. To study the scale-out performance, we fix y to 4 while varying x as 2, 4, 8, 16[7], and report the latency of each case in Figure 12a and Figure 12b. We analyze the result regarding the two query groups:

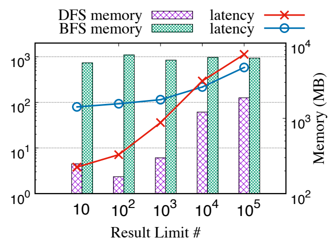
Large queries. These queries traverse large amount of data and run relatively longer, while they scale well with up to 6× performance gain from 2 machine to 16 machines. While CR-3 performs the worst to obtain only 3× performance gain, we recognize that it contains very complex nested sub-traversals that may introduce extra cost in synchronization (e.g. waiting for the EOS marker).

Small queries. Due to either effective filtering or small range of traversal, the small queries only touch a small amount of data and thus are not computation-intensive. We expect that their performance may not be improved with more parallelism, while CR-2 and CR-12 still run consistently faster as shown in Figure 12b. CR-1, as a relatively slow query in

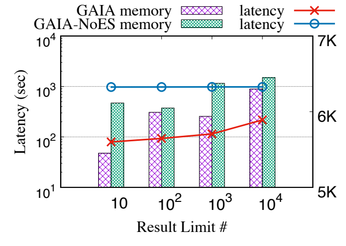
[7]G_{300} is too large to be held on one machine.



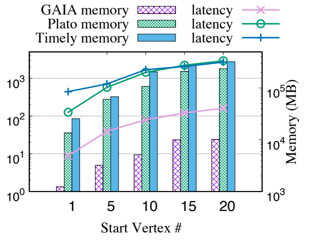
(a) Bound memory execution.



(b) Traversal strategy.



(c) Early stop.



(d) Compare with big engines.

Figure 13: The experiment of our design choices.

this group, demonstrates seemingly counter-intuitive result. The query actually asks to print out a lot of information after locating the target vertices, which constitutes a majority of the computation that cannot benefit from more parallelism.

Scale-up. We then fix x to 4, and vary y as 2, 4, 8, 16, and report the result in Figure 12c and Figure 12d. Similar to the scale-out cases, the large queries scale consistently, while small queries do not gain speedup, as more parallelism is used. It is interesting to compare the scale-out and scale-up cases with the same DOP, $[4] \times [16]$ vs. $[16] \times [4]$ as an example, we can observe $[16] \times [4]$ cases in-general perform better, even it requires more communication. The result shows that (1) communication cost is not a critical impact factor for GAIA, for which the dynamic scheduling techniques can seamlessly hide the communication cost by allowing ready tasks to get scheduled; (2) data contentions may be a more serious issue for interactive graph queries, as they are more often confronted in fewer machines.

Data Size. Finally, we fix the DOP as $[16] \times [4]$, and run the queries over the datasets of G_{30}, G_{100}, G_{300} and G_{1000}. Note that the sizes of these graphs are roughly linear to their *scale* factors. The result is in Figure 12e and Figure 12f. For the large queries, GAIA scales quite well with the growing of the data. For the small queries (except CR-1, as explained earlier), the performance stays roughly stable, as these queries only touch a small amount of data.

Discussions. The experiment demonstrates reasonable trends of scalability of GAIA: in general, the larger the query, the better the scalability. Due to the irregularity of graph data (and queries), it is challenging to derive the optimal DOP for each query, while we leave it as an interesting future work.

6.3 Our Design Choices

We study our design choices in this experiment by drilling down to the performance factors including bounded-memory execution (Section 5.1), hybrid traversal strategies (Section 5.1) and early-stop optimization (Section 5.2). We run Q6 on G_{300} using the DOP of $[16] \times [4]$, and report the query latency and peak memory usage among all machines. We use Q6 here as it includes complex nested Scopes with fine-grained dependency, and it is a real query in production. We conduct this experiment while adjusting the query parameters m (number of starting vertices), k (the hop limit) and n (the result limit) in Q6, and the system parameter of memory upper-bound of each query (default 10GB) and traversal strategies (default hybrid), and whether early stop is enabled (default enabled). We configure the following variants of GAIA, namely GAIA (default settings), GAIA-DFS (manual DFS-prioritized strategy)[8], GAIA-NoMB (without/infinity memory bound) and GAIA-NoES (without early stop).

Dynamic Scheduling. In this experiment, we study the effectiveness of dynamic scheduling. We vary the memory upper-bound as 256, 512, 1024, 2048, 4096 (MB) and infinity with $m = 10$ starting vertices, and report the result in Figure 13a. The actual memory usage (as labelled) of all cases is very close to the bounded value, and is noticeably smaller than the unbounded case, which has surged to more than 25GB. An interesting observation is that the latency increases with the memory bound. Note that graph traversal exhibits massive parallelism and all the CPU cores available can be fully utilized with just "enough" memory. Additional memory incurs overheads (in allocation, buffering, etc.) rather than benefits.

Traversal Strategy. To verify the effectiveness of the hybrid traversal strategy in GAIA, we compare GAIA with GAIA-DFS/BFS. We vary n from 10 to 10^5, and report the time cost and memory usage in Figure 13b. GAIA-DFS outperforms GAIA when $n \leq 1000$. This is because that DFS strategy will prioritize scheduling operators in the deeper order (in the dataflow), which can potentially escape earlier (thanks to early stop) as soon as n cycles have been found. As n increases, the hybrid strategy gradually catches up with, and eventually outperforms DFS, as it can compute the required number of cycles in a lower order. This experiment shows that the best traversal strategy can be query- (and data-) dependent, and the hybrid strategy is a more generic option.

Early Stop. We compare the performance of GAIA and GAIA-NoES (without early stop). We vary n from 10 to 10^4, and report the query latency and memory usage in Figure 13c. When early stop is turned off, both the query latency and memory usage remain fairly stable, as GAIA always computes all result, regardless of the limit number. When early stop is turned on, it can be observed that both the query latency and memory consumption drop noticeably, compared to the cases without early stop. In particular, the early-stop optimization enables $12\times$ improved performance and $1GB$ memory saving when the limit number is 10.

[8]Note that the BFS-prioritized strategy often causes out-of-memory, and is thus excluded from our test.

Table 3: Comparison GAIA variants with big-data engines.

	GAIA	-DFS	-NoMB	-NoES	Plato	Timely
Lat./Sec.	79	4	440	972	1431	1690
Mem./GB	5.2	0.3	25.6	6.1	108	205

Comparing with Big-Data Engines. Finally, we compare our GAIA with existing high-performance engines, Timely and Plato, in this experiment. We implement Q6 in Timely and Plato[9], which contains 105 and 95 logical lines of codes, respectively. In comparison, the Gremlin query is written in 5 lines as presented in Figure 2. The query latency and memory consumption of these engines, while varying m as 1, 5, 10, 15, 20, is shown in Figure 13d. GAIA achieves $16\times$ and $14\times$ better performance, and consumes $21\times$ and $10\times$ less memory, than Timely and Plato, respectively. To demonstrate how GAIA benefits from the proposed techniques to outperform existing engines, we further bring different variants of GAIA into the comparison, and the results of $m = 10$ are in Table 3. The performance of GAIA drops by $5.5\times$ without memory bound, and by over $12\times$ without early stop, while the latter is already in the same order as those of Plato and Timely. Note that GAIA-DFS even outperforms the default GAIA (hybrid) due to the small result limit ($n = 10$). This experiment shows that the novel design choices of GAIA, notably the Scope abstraction, and the techniques proposed on top of it, enable more convenient programming and efficient execution of the Gremlin queries over big-data engines.

6.4 Comparison with Graph Databases

Small-Scale DB. Although GAIA is designed to scale, we show that GAIA demonstrates efficiency while compared to graph databases on one single machine. Specifically, we use the small graph G_1 so that all the systems can load and process queries in reasonable time; and for each LDBC query, we choose the best query performance among the 4 systems (TinkerGraph, Neo4j, OrientDB and JanusGraph) as the *BSTI* for the query; then we vary the DOP of GAIA, and report the relative performance of GAIA to BSTI in Figure 14.

GAIA performs comparably to the BSTI in most cases except for queries CR-3 (up to $7\times$ worse) and CR-12. Neo4j performs better than any other systems on these queries. Further investigation shows that, instead of faithfully traversing the graph, Neo4j applies a *join* on some partial result to generate the output, which turns out to be more efficient in these cases. We leave better query optimization of Gremlin on GAIA as future work. As a whole, GAIA has an average relative performance of just around 1.8 using single thread, and of 0.73 using 16 threads, among all LDBC queries.

Large-scale DB. We use G_{100} in this experiment to run all LDBC queries. Note that we only compare JanusGraph, as

[9]For fair comparison, we implement cycle detection in Timely and Plato using the same algorithm as in GAIA. In addition, we exploit all possible optimizing options from both systems for the test.

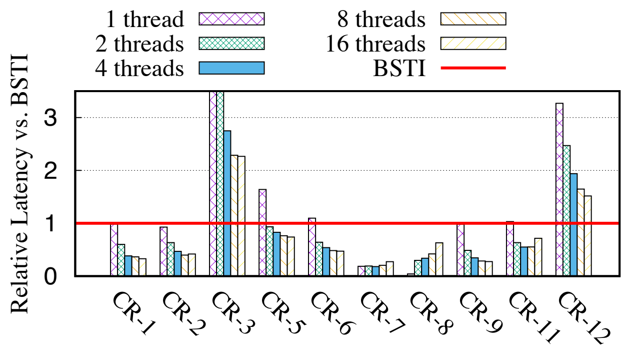


Figure 14: GAIA performance relative to the best single-threaded implementation (BSTI).

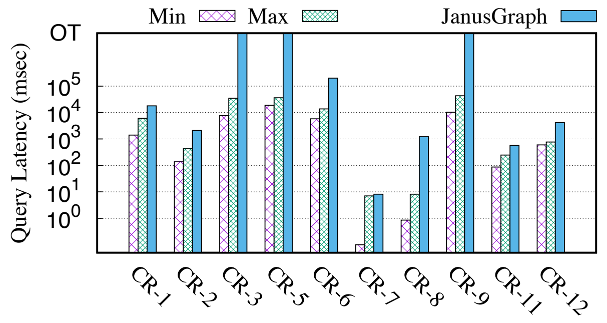


Figure 15: Compare GAIA with JanusGraph.

it is the only system that can store graph at this scale. JanusGraph cannot process query in parallel, and we run GAIA in one machine for fair comparison. The graphs are stored in 8 machines for JanusGraph, and one single machine[10] for GAIA. We run each query on GAIA with DOP varying from 1 to 16, and report its max and min latency for each query while compared to JanusGraph. The result is reported in Figure 15. JanusGraph fails to answer many queries (CR-3,5,9) due to OT. As shown, even the maximum latency (single-thread) of GAIA is much shorter than that of JanusGraph in all cases. Although GAIA is designed to scale in a cluster, it can further benefit from multi-core parallelism in a single machine to improve query performance, especially for large queries, as can be seen in Figure 15.

7 Related Work

Graph Databases. Gremlin is widely supported by many graph databases, such as Neo4j [29], OrientDB [30], JanusGraph [21], and cloud-based services including Cosmos DB [6] and Neptune [1]. However, their query processing is limited to one single process. Driven by rapidly growing needs to query large graph data, several distributed in-memory graph systems emerge, such as Trinity [40], ZipG [24], Wukong+S [48], Grasper [20], and A1 [9]. Trinity and ZipG offer their own programming models that are less flexible than Gremlin. Grasper adopts Gremlin but provides a limited subset of the language constructs (e.g., the lack of nested-loop support). Wukong+S and A1 leverage RDMA for serving *micro-second* queries with much higher concurrency, which is not the main target scenario of GAIA.

[10]Note that JanusGraph is properly warmed up to reduce the cost of pulling data from remote storage.

Graph Processing Systems. In contrast to many other systems that deal with batch-oriented iterative graph processing, such as Pregel [26], PowerGraph [15], GraphX [16], and Gemini [49], GAIA focuses on low-latency graph traversal at scale. It is hard to support graph traversal in existing graph processing systems. Firstly, their programming abstractions [22] are usually low-level, makes these systems a privilege for experienced users only [13]. Moreover, they typically adopt the bulk synchronous parallel (BSP) execution model, which is more suitable for an iterative *routine* processing over the *whole* graph, but can be inefficient for running graph traversal that visits an *arbitrary portion* of the graph.

Dataflow Engines and Dependency Tracking. A number of existing systems such as CIEL [28], Naiad [27], and TensorFlow [45] offer generic data-parallel computing infrastructures with support for dynamic control flow. While it is possible to program the logic of a Gremlin query on top of these frameworks, it is extremely challenging to do so in the pursuit of both correctness and efficiency, largely due to the fine-grained dependency in Gremlin traversal. Tracking dependency has been exploited to compute what is absolutely necessary when there are limited changes to the input (e.g., incremental computing as in Incoop [7], DryadInc [33], Nectar [19]), or frugal re-computation to repair lost state as in MadLINQ [34] and TimeStream [35].

Declarative Programming Languages. Graph queries are typically expressed using graph traversal and pattern matching. Correspondingly, Gremlin [37] and Cypher [14] are the most popular query languages. Cypher allows users to specify a graph pattern with variables. However, based on our production experience, it is often challenging to compose ad-hoc query pattern for a particular task. Therefore, we support Gremlin instead of Cypher in this work. Other notable research projects in parallel declarative languages, such as Cilk [8], can be leveraged by GAIA in theory, but they are not particularly tailored for distributed graph traversal.

8 Conclusion

GAIA has been in use by a small community of domain experts for over a year in production at Alibaba. Our overall experience is that GAIA, by combining the benefits of Gremlin with the power of distributed dataflow execution, proves to be a simple, useful and efficient programming environment for interactive analysis on big graph data.

Acknowledgments

We thank Benli Li, Pin Gao, and Donghai Yu for answering Plato related questions. We are grateful to Alibaba GraphScope team members for their support. Thanks also to the NSDI review committee, as well as our shepherd Anurag Khandelwal, for their valuable comments and suggestions.

References

[1] Amazon Neptune. https://aws.amazon.com/neptune/. [Online; accessed 2-March-2021], 2019.

[2] Renzo Angles and Claudio Gutierrez. Survey of Graph Database Models. *ACM Comput. Surv.*, 40(1), February 2008.

[3] Apache TinkerPop. http://tinkerpop.apache.org/. [Online; accessed 2-March-2021], 2019.

[4] Ching Avery. Giraph: Large-Scale Graph Processing Infrastructure on Hadoop. *Proceedings of the Hadoop Summit. Santa Clara*, 11(3):5–9, 2011.

[5] Konstantin Avrachenkov and Nelly Litvak. The Effect of New Links on Google PageRank. *Stochastic Models*, 22(2):319–331, 2006.

[6] Azure Cosmos DB. https://azure.microsoft.com/en-us/services/cosmos-db/. [Online; accessed 2-March-2021], 2019.

[7] Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A. Acar, and Rafael Pasquin. Incoop: MapReduce for Incremental Computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, New York, NY, USA, 2011. Association for Computing Machinery.

[8] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. *SIGPLAN Not.*, 30(8):207–216, August 1995.

[9] Chiranjeeb Buragohain, Knut Magne Risvik, Paul Brett, Miguel Castro, Wonhee Cho, Joshua Cowhig, Nikolas Gloy, Karthik Kalyanaraman, Richendra Khanna, John Pao, Matthew Renzelmann, Alex Shamis, Timothy Tan, and Shuheng Zheng. A1: A Distributed In-memory Graph Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 329–344, New York, NY, USA, 2020. Association for Computing Machinery.

[10] Ding-Kai Chen, Hong-Men Su, and Pen-Chung Yew. The Impact of Synchronization and Granularity on Parallel Systems. *SIGARCH Comput. Archit. News*, 18(2SI):239–248, May 1990.

[11] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: Taking the Pulse of a Fast-Changing and Connected World. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 85–98, New York, NY, USA, 2012. ACM.

[12] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. The LDBC Social Network Benchmark: Interactive Workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 619–630, New York, NY, USA, 2015. ACM.

[13] Wenfei Fan, Jingbo Xu, Yinghui Wu, Wenyuan Yu, and Jiaxin Jiang. Grape: Parallelizing Sequential Graph Computations. *Proceedings of the VLDB Endowment*, 10(12):1889–1892, 2017.

[14] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An Evolving Query Language for Property Graphs. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1433–1445, New York, NY, USA, 2018. ACM.

[15] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.

[16] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. GraphX: Graph Processing in a Distributed Dataflow Framework. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 599–613, 2014.

[17] Gremlin Console. http://tinkerpop.apache.org/docs/3.4.3/reference/#gremlin-console. [Online; accessed 2-March-2021], 2019.

[18] Gremlin Server. http://tinkerpop.apache.org/docs/3.4.3/reference/#connecting-gremlin-server. [Online; accessed 2-March-2021], 2019.

[19] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A Thekkath, Yuan Yu, and Li Zhuang. Nectar: Automatic Management of Data and Computation in Datacenters. In *OSDI*, volume 10, pages 1–8, 2010.

[20] Chen Hongzhi, Li Changji, Fang Juncheng, Huang Chenghuan, Cheng James, Zhang Jian, Hou Yifan, and Yan Xiao. Grasper: A High Performance Distributed System for OLAP on Property Graphs. In *ACM Symposium on Cloud Computing 2019*, Socc'19, 2019.

[21] JanusGraph. http://janusgraph.org/. [Online; accessed 2-March-2021], 2019.

[22] Vasiliki Kalavri, Vladimir Vlassov, and Seif Haridi. High-Level Programming Abstractions for Distributed Graph Processing. *CoRR*, abs/1607.02646, 2016.

[23] U Kang, Mary McGlohon, Leman Akoglu, and Christos Faloutsos. Patterns on the Connected Components of Terabyte-Scale Graphs. In *2010 IEEE International Conference on Data Mining*, pages 875–880. IEEE, 2010.

[24] Anurag Khandelwal, Zongheng Yang, Evan Ye, Rachit Agarwal, and Ion Stoica. ZipG: A Memory-Efficient Graph Store for Interactive Queries. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 1149–1164, New York, NY, USA, 2017. Association for Computing Machinery.

[25] Andrea Lattuada, Frank McSherry, and Zaheer Chothia. Faucet: A User-level, Modular Technique for Flow Control in Dataflow Engines. In *Proceedings of the 3rd ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*, BeyondMR '16, pages 2:1–2:4, New York, NY, USA, 2016. ACM.

[26] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[27] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A Timely Dataflow System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA, 2013. ACM.

[28] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. CIEL: A Universal Execution Engine for Distributed Data-Flow computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, page 113–126, USA, 2011. USENIX Association.

[29] Neo4j. `https://neo4j.com/`. [Online; accessed 2-March-2021], 2019.

[30] OrientDB. `https://orientdb.com/`. [Online; accessed 2-March-2021], 2019.

[31] Thomas M Parks. Bounded Scheduling of Process Networks. Technical report, CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCES, 1995.

[32] Plato: A Framework for Distributed Graph Computation. `https://github.com/Tencent/plato`. [Online; accessed 2-March-2021], 2020.

[33] Lucian Popa, Mihai Budiu, Yuan Yu, and Michael Isard. DryadInc: Reusing Work in Large-Scale Computations. *HotCloud*, 9:2–6, 2009.

[34] Zhengping Qian, Xiuwei Chen, Nanxi Kang, Mingcheng Chen, Yuan Yu, Thomas Moscibroda, and Zheng Zhang. MadLINQ: Large-Scale Distributed Matrix Computation for the Cloud. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 197–210, 2012.

[35] Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. TimeStream: Reliable Stream Computation in the Cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 1–14, 2013.

[36] Rodrigo Caetano Rocha and Bhalchandra D Thatte. Distributed Cycle Detection in Large-Scale Sparse Graphs. *Proceedings of Simpósio Brasileiro de Pesquisa Operacional (SBPO'15)*, pages 1–11, 2015.

[37] Marko A. Rodriguez. The Gremlin Graph Traversal Machine and Language (Invited Talk). In *Proceedings of the 15th Symposium on Database Programming Languages*, DBPL 2015, pages 1–10, New York, NY, USA, 2015. ACM.

[38] Rust Programming Language. `https://www.rust-lang.org/`. [Online; accessed 2-March-2021], 2020.

[39] Semih Salihoglu and Jennifer Widom. GPS: A Graph Processing System. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, pages 1–12, 2013.

[40] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A Distributed Graph Engine on a Memory Cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, page 505–516, New York, NY, USA, 2013. Association for Computing Machinery.

[41] The HIVE project. `http://hadoop.apache.org/hive/`. [Online; accessed 2-March-2021], 2020.

[42] TigerGraph. `https://www.tigergraph.com/benchmark/`. [Online; accessed 2-March-2021], 2018.

[43] Timely Dataflow. https://github.com/TimelyDataflow/timely-dataflow. [Online; accessed 2-March-2021], 2019.

[44] Leslie G Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, 1990.

[45] Yuan Yu, Martín Abadi, Paul Barham, Eugene Brevdo, Mike Burrows, Andy Davis, Jeff Dean, Sanjay Ghemawat, Tim Harley, Peter Hawkins, Michael Isard, Manjunath Kudlur, Rajat Monga, Derek Murray, and Xiaoqiang Zheng. Dynamic Control Flow in Large-Scale Machine Learning. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 18:1–18:15, New York, NY, USA, 2018. ACM.

[46] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: A System for General-Purpose Distributed Data-parallel Computing Using a High-Level Language. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

[47] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for Inmemory Cluster Computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA, April 2012. USENIX Association.

[48] Yunhao Zhang, Rong Chen, and Haibo Chen. Submillisecond Stateful Stream Querying over Fast-Evolving Linked Data. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 614–630, New York, NY, USA, 2017. ACM.

[49] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A Computation-Centric Distributed Graph Processing System. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 301–316, 2016.

TEGRA: Efficient Ad-Hoc Analytics on Evolving Graphs

Anand Padmanabha Iyer[⋆†], Qifan Pu[◇], Kishan Patel[‡], Joseph E. Gonzalez[†], Ion Stoica[†]
[⋆]*Microsoft Research* [◇]*Google* [‡]*Two Sigma* [†]*University of California, Berkeley*

Abstract

Several emerging *evolving graph* application workloads demand support for efficient ad-hoc analytics—the ability to perform ad-hoc queries on arbitrary time windows of the graph. We present TEGRA, a system that enables efficient ad-hoc window operations on evolving graphs. TEGRA allows efficient access to the state of the graph at arbitrary windows, and significantly accelerates ad-hoc window queries by using a compact in-memory representation for both graph and intermediate computation state. For this, it leverages persistent data structures to build a versioned, distributed graph state store, and couples it with an incremental computation model which can leverage these compact states. For users, it exposes these compact states using Timelapse, a natural abstraction. We evaluate TEGRA against existing evolving graph analysis techniques, and show that it significantly outperforms state-of-the-art systems (by up to 30×) for ad-hoc window operation workloads.

1 Introduction

Graph-structured data is on the rise, in size, complexity and dynamism [1, 61]. This growth has spurred the development of a large number of graph processing systems [16, 17, 19, 26, 27, 30, 33, 39, 42, 51, 54, 57, 59, 60, 68] in both academia and the open-source community. By leveraging specialized abstractions and careful optimizations, these systems have the ability to analyze large, *static* graphs, some even in the order of a trillion edges [20].

However, real-world graphs are seldom static. Consider, the familiar example of social network graphs such as in Facebook and Twitter. In these networks, "friends" relations and tweets with "mentions" are created continuously resulting in the graph's constant evolution. Similarly, each new call in a cellular network connects devices with receivers and can be used in real-time for network monitoring and diagnostics [31]. Additionally, emerging applications such as connected cars [13], real-time fraud detection [9], and disease analysis [23] all produce such graph data. Analyzing these *time-evolving* graphs can be useful, from scientific and commercial perspectives, and is often desired [61].

In this paper, we focus on the problem of *efficient ad-hoc window operations* on evolving graphs—the ability to perform ad-hoc queries on arbitrary time windows (i.e., segments in time) either in the past or in real-time. To illustrate the need for such operations, consider two examples. In the first, a financial expert wishes to improve her fraud-detection algorithm. For this, she retrieves the *complete states of the graph at different segments in time* to train and test variants of her algorithm. In the second example, a network administrator wishes to diagnose a transient failure. To do so, she retrieves a *series of snapshots*[1] *of the graph* before and after the failure, and runs a handful of queries on them. She iteratively refines the queries until she comes up with a hypothesis. In such scenarios, neither the queries nor the windows on which the queries would be run are predetermined.

To efficiently perform ad-hoc window operations, a graph processing system should provide two key capabilities. First, it must be able to quickly retrieve arbitrary size windows starting at arbitrary points in time. There are two approaches to provide this functionality. The first is to store a *snapshot* every time the graph is updated, i.e., a vertex or edge is added or deleted. While this allows one to efficiently retrieve the state of the graph at *any* point in the past, it can result in prohibitive overhead. An alternative is to store only the changes to the graph and reconstruct a snapshot on demand. This approach is space efficient, but can incur high latency, as it needs to re-apply all updates to reconstruct the requested snapshot(s). Thus, there is a fundamental trade-off between in-memory storage and retrieval time.

Second, we must be able to efficiently execute queries (e.g., connected components) not only on a single window, but also across multiple related windows of the graph. Existing systems, such as Chronos [30] allows executing queries on a single window, while Differential Dataflow [54] and GraphBolt [45] support continuously updating queries over sliding windows. However, none of the systems support efficient execution of queries across multiple windows, as they do not have the ability to share the computation state across windows and computations. This fundamental limitation of existing systems arises from their inability to efficiently store intermediate state from within a query for later reuse.

We present TEGRA[2], a system that enables efficient ad-hoc window operations on time-evolving graphs. TEGRA is based on two key insights about such real-world evolving graph workloads: *(1) during ad-hoc analysis graphs change slowly over time relative to their size , and (2) queries are frequently applied to multiple windows relatively close by in time.*

[1] A snapshot is a full copy of the graph, and can be viewed as a window of size zero. Non-zero windows have several snapshots.

[2] for **T**ime **E**volving **GR**aph **A**nalytics.

Leveraging these insights TEGRA is able to significantly accelerate window queries by reusing both storage and computation across queries on related windows. TEGRA solves the storage problem through a highly efficient, distributed, versioned *graph state store* which compactly represents graph snapshots in-memory as logically separate versions that are efficient for arbitrary retrieval. We design this store using persistent (functional) data-structures that lets us heavily share common parts of the graph thereby reducing the storage requirements by several orders of magnitude (§5). Second, to improve the performance of ad-hoc queries, we introduce an efficient in-memory representation of intermediate state that can be stored in our graph state store and enables non-monotonic[3] incremental computations. This technique leverages the computation pattern of the familiar graph-parallel models to create compact intermediate state that can be used to eliminate redundant computations across queries. (§4).

TEGRA exposes these compact persistent snapshots of the graph and computation state using a logical abstraction named *Timelapse*, which hides the intricacies of state management and sharing from the developer. At a high level, a timelapse is formed by a sequence of graph snapshots, starting from the original graph. Viewing the time-evolving graph as consisting of a sequence of independent static *snapshots* of the entire graph makes it easy for the developer to express a variety of computation patterns naturally, while letting the system optimize computations on those snapshots with much more efficient incremental computations (§3). Finally, since Timelapse is backed by our persistent graph store, users and computations always work on independent *versions* of the graph, without having to worry about consistency issues. Using these, TEGRA outperforms state-of-the-art systems significantly on ad-hoc window operation workloads (§7).

In summary, we make the following contributions:

- We present TEGRA, a time-evolving graph processing system that enables efficient ad-hoc window operations on both historic and live data. To achieve this, TEGRA shares storage, computation and communication across queries by compactly representing the evolving graph and intermediate computation state in-memory.
- We propose *Timelapse*, a new abstraction for time-evolving graph processing. TEGRA exposes timelapse to the developer using a simple API that can encompass many time-evolving graph operations. (§3)
- We design *DGSI*, an efficient distributed, versioned property graph store that enables timelapse APIs to perform efficient operations. (§5)
- Leveraging timelapse and DGSI, we present an incremental graph computation model which supports non-monotonic computations across (non-contiguous) windows. (§4)

[3] Allows vertex/edge deletions, additions and modifications on any graph algorithm implemented in a graph-parallel fashion.

2 Background & Challenges

We begin with a brief background on graph-parallel systems (§2.1) and then describe various types of time-evolving graph workloads (§2.2). The limitations of existing systems are discussed (§2.3) before we layout the challenges in enabling efficient ad-hoc analytics on evolving graphs (§2.4).

2.1 Graph-Parallel Systems

Most general purpose graph systems provide a *graph-parallel* abstraction for performing computations. In the graph-parallel abstraction, a user-defined program is run (in parallel) on every entity in the graph, who then change their state depending on the neighborhood. This process is iteratively done until convergence. Thus, the graph-parallel abstraction lets the end-developer view distributed graph computations as simpler entity centric computations, leaving the burden of orchestration to the system. The simplest of the graph-parallel abstraction is a vertex-centric model [46], where every vertex independently runs the user program. Several other forms have been proposed, such as the graph centric models [66], edge centric models [59] and the more recent subgraph centric models [57]. In all these models, the basic form of computation is implemented as message exchanges between the entities and their corresponding state changes. Communication is enabled either via shared memory model [63] or message passing interface [43]. PowerGraph [27] introduced the **Gather-Apply-Scatter (GAS)** model, a popular vertex-centric model adopted by many open-source graph processing systems, where a vertex program is represented as three conceptual phases: **gather** phase that collects information about adjacent vertices and edges and applies a function on them, **apply** phase that uses the function's output to update the vertex, and **scatter** phase that uses the new vertex value to update adjacent edges. To perform a graph algorithm computation, the system iteratively applies these phases until convergence. TEGRA focuses on the GAS model. (§6).

2.2 Time-evolving Graph Workloads

Time-evolving graph workloads, an important graph workload [61], can be classified into three categories:

Temporal Queries: Here, an analyst is querying the graph at different points in the past and evaluates how the result changes over time. Examples are *"How many friends did Alice have in 2017?"* or *"How did Alice's friend circle change in the last three years?"*. Such queries may have time windows of the form $[T-\delta, T]$ and are performed on offline data, executed in a batch fashion.

Streaming/Online Queries: These workloads are aimed at keeping the result of a graph computation up-to-date as new data arrives (i.e., $[Now-\delta, Now]$). For example, the analyst may ask *"What are the trending topics now?"*, or use a moving window (e.g., *"What are the trending topics in the last 10 minutes?"*). These focus on the most recent data, thus streaming systems operate on the live graph.

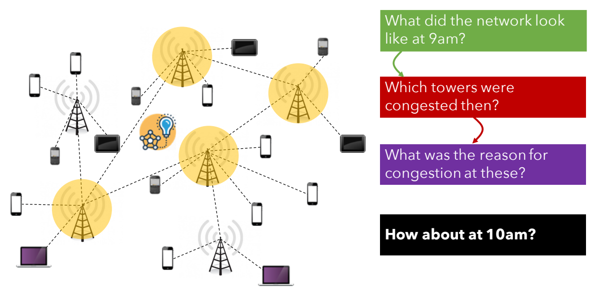

What did the network look like at 9am?
Which towers were congested then?
What was the reason for congestion at these?
How about at 10am?

Figure 1: Carol, a network administrator, diagnoses issues by performing ad-hoc queries on *snapshots* of the network at disjoint but *close-by* points in time.

Ad-hoc Queries: In these workloads, an analyst is likely to explore the graph by performing ad-hoc queries on arbitrary windows. Below, we illustrate a real-world use case. We omit some details and use fictitious names for anonymity.

Carol is a network administrator at a large cellular network operator in the United States [58]. Her job is to manage several thousands of wireless base stations, deployed across a large geographic region. When problems occur, Carol is tasked with finding the reason for the issue and fix them. For instance, she may be trying to answer *"What is the reason for poor download throughput for (several) users at 9:00am?"*.

Cellular network operators collect extensive data (several terabytes per day) from their network, and previous studies have shown the benefits of viewing cellular network analytics as a time-evolving graph problem [31]. Carol might start by asking *"What did the network look like at 9am?"*, when the problem was reported. The query returns a *graph view* of the network, depicted in fig. 1, where there are several base stations (towers) serving many users. Carol doubts congestion as the cause for low throughput, so the next query is *"Which towers were congested at this point?"* that returns a subset of towers from the original answer. Based on her extensive domain knowledge, she knows that temporary congestion in some towers do not cause persistent poor throughput, so she is interested in learning if *clusters* of towers were congested. To do so, she runs a *connected component* algorithm on the network graph. Then, to confirm her hypothesis, she asks *"How about at 10am?"* meaning to repeat the entire analysis again, but now on a different subset of the data. By the time Carol finishes her investigation, she has retrieved *100s of different subsets of data*, each depicting a *snapshot of the network* at some *disjoint, random* point in time around the failure, conducted exploratory analysis to test several of her hypothesis including one where she runs connected components.

Thus, in ad-hoc workloads, not only does the analyst need to access arbitrary windows, but also the queries and the windows on which they are executed are determined just-in-time (i.e., not predetermined). Further, the analyst applies the same query to multiple (close-by, discontinuous) windows.

2.3 State of the Art & Limitations

Recent work in graph systems has made considerable progress in the area of evolving graph processing. (§8)

Temporal analysis engines (e.g., **Chronos** [30], **ImmortalGraph** [50]) operate on *offline data* and focus on executing queries on one or a sequence of snapshots in the graph's history. Upon execution of a query, these systems load the relevant history of the graph and utilize a pre-processing step to create an in-memory layout that is efficient for analysis. Such preprocessing can often dominate the algorithm execution time [44]. As a result, these systems are tuned for operating on a large number of snapshots in each query (e.g., temporal changes over months or year), and are efficient in such cases. Fundamentally, the in-memory representation in these systems cannot support updates. Additionally, these systems do not allow updating the results of a query.

Proposals such as GraphOne [35, 36] and Aspen [21] focus on providing *efficient storage for streaming computations*. These (typically single-machine) systems allow only storing a few recent versions of the graph and do not support storing intermediate state, or updating the results of a previous query. **GraphOne** [35, 36] combines circular edge logs and adjacency store to allow storing a few recent versions of the graph and dual versioning to decouple computation from storage. However, the use of *chaining* (with compaction) in the adjacency store to enable versioning introduces an ordering constraint among the versions, and traversing (and applying operations e.g., deletions) is necessary to retrieve a specific snapshot. For ad-hoc analysis where arbitrary changes maybe applied on a version, this may be expensive and fundamentally difficult. **Aspen** [21] leverages functional/persistent datastructures to preserve previous version of the graph upon mutation and presents C-trees, a highly compressed functional tree that can store graphs efficiently (they do not allow property graphs). This is similar in spirit to our use of persistent datastructures in designing DGSI (§5). However, C-trees are tuned for streaming workloads where there is one (or a few) previous version(s) and thus employ aggressive garbage collection for efficiency. When large number of versions are required, main memory becomes a bottleneck and thus it is necessary to have a persistent storage based hybrid store (e.g., DGSI).

Streaming systems (e.g., Kineograph [19], Differential-Dataflow [49], Kickstarter [67], GraphBolt [45]) operate on *live data* and allow query results to be updated *incrementally* (rather than doing a full computation) when *new* data arrives. These systems only allow queries on the live graph, and do not support ad-hoc retrieval of previous state. Additionally, the incremental computation is tied to the live state of the graph, and cannot be utilized over multiple windows.

Differential Dataflow (DD) [49] is a distributed system that allows general, non-monotonic incremental computations using special versions of operators. Each operator stores "differences" to its input and produces the corresponding differences in output (hence the full output is not materialized), automatically incrementalizing algorithms written using them. While this technique is very efficient for real-time streaming queries, incorporating ad-hoc window operations in it is fundamentally hard. Since the computation model is based on the operators maintaining state (*differences* to their input and output) indexed by data (rather than time), accessing a particular snapshot can require a full scan of the maintained state. Further, since every operator needs to maintain state, the system accumulates large state over time which must be compacted (at the expense of forgoing the ability to retrieve the past). Finally, intermediate state of a query is cleared once completed and storing these efficiently for reuse is an open question.

GraphBolt [45], a *single-machine* streaming system, presents a dependency driven "refinement" based computation model for (non-monotonic) incremental computations that tracks dependency information as aggregation values at vertices thus reducing the state requirements to O|V| (in contrast to DD's O|E|). Users can implement incremental algorithms by defining user-defined refinement functions (e.g., `repropagate`, `retract` and `propagate`) whose implementations are algorithm specific (e.g., Algorithm 3 in [45] for PageRank), and iteratively refines aggregation values. GraphBolt only stores the value aggregations for the current snapshot of the graph and does not present a solution for storing multiple versions of aggregations or efficiently using/operating on them. Thus, GraphBolt does not support ad-hoc analysis. Building ad-hoc support requires building a state store, similar to the solution we present (DGSI), tuned for GraphBolt's computation model and exposing the right APIs.

2.4 Challenges

Several challenges stand in the way of enabling efficient ad-hoc analytics on evolving graphs. First is the ability to efficiently *store* and *retrieve* snapshots of the graph at arbitrary time windows. Second, we must be able to *compactly represent large amounts of computation state and use it to accelerate future queries, across multiple windows* using a computation model that can leverage the state efficiently. Finally, the system should be able to provide users a natural and intuitive way to operate on evolving graphs. Based on this, our solution, TEGRA, consists of three components:

Timelapse Abstraction (§3): In TEGRA, users interact with time-evolving graphs using the *timelapse* abstraction, which logically represents the evolving graph as a sequence of *static*, *immutable* graph snapshots (fig. 2). TEGRA exposes this abstraction via a simple API that allows users to save/retrieve/query the materialized *state* of the graph at any point.

Computational Model (§4): TEGRA proposes a computation model that allows *ad-hoc queries across windows to share computation and communication*. The model stores compact intermediate state as a timelapse, and uses it to perform general, *non-monotonic* incremental computations.

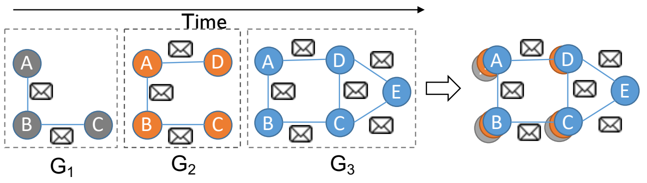


Figure 2: A timelapse of graph G consisting of three snapshots. For temporal analytics, instead of applying graph-parallel operations independently on each snapshot (left), timelapse enables them to be applied to all snapshots in parallel (right).

Distributed Graph Snapshot Index (§5): TEGRA stores evolving graphs, intermediate computation state and results in DGSI, an efficient indexed, distributed, versioned property graph store which *shares storage* between versions of the graph. Such decoupling of state from queries and operators allow TEGRA to share it across queries and users.

3 Timelapse Abstraction & API

TEGRA introduces *Timelapse* as a new abstraction for time-evolving graph processing that enables efficient ad-hoc analytics. The goal of timelapse is to provide the end-user with a simple, natural interface to run queries on time-evolving graphs, while giving the system opportunities for efficiently executing those queries. In timelapse, TEGRA *logically* represents a time-evolving graph as a sequence of immutable, static graphs (fig. 2), each of which we refer to as *snapshot* in the rest of this paper. A snapshot depicts a consistent state of the graph at a particular instance in time. TEGRA uses the popular property graph model [26], where vertices and edges in the graph are associated with arbitrary properties, to represent each snapshot in the timelapse. For the end-user, timelapse provides the abstraction of having access to a *materialized* snapshot at any point in the history of the graph. This enables the usage of the familiar static graph processing model in evolving graphs (e.g., queries on arbitrary snapshot).

Timelapses are created in TEGRA in two ways—by the system and by the users. When a new graph is introduced to the system, a timelapse is created for it that contains a single snapshot of the graph. Then, as the graph evolves, more snapshots are added to the timelapse. Similarly, users may create timelapses while performing analytics. Because snapshots in a timelapse are immutable, any operation on them creates new snapshots as a result (e.g., a query on a snapshot results in another snapshot as a result). Such newly created snapshots during an analytics session may be added to an existing timelapse, or create a new one depending on the kind of operations performed. For instance, for an analyst performing what-if analysis by introducing artificial changes to the graph, it is logical to create a new timelapse. Meanwhile, snapshots created as a result of updating a query result should ideally be added to the same timelapse. The system does not impose restrictions on how users want to book-keep timelapses. Instead, it simply tracks their lineage and allows

`save(id): id`	Save the state of the graph as a snapshot in its timelapse. ID can be autogenerated. Returns the id of the saved snapshot.
`retrieve(id): snapshot`	Return one or more snapshots from the timelapse. Allows simple matching on the id.
`diff(snapshot, snapshot): delta`	Difference between two snapshots in the timelapse. (§4)
`expand(candidates): subgraph`	Given a list of candidate vertices, expand the computation scope by marking their 1-hop neighbors. Used for implementing incremental computations (§4)
`merge(snapshot, snapshot,func): snapshot`	Create a new snapshot using the union of vertices and edges of two snapshots. For common vertices, run func to compute their value. Used for implementing incremental computations (§4)

Table 1: TEGRA exposes Timelapse via simple APIs.

users to efficiently operate on them. We describe how TEGRA implements timelapses in §5.3.

Since timelapse logically represents a sequence of related graph snapshots, it is intuitive to expose the abstraction using the same semantics as that of static graph. In TEGRA, users interact with timelapses using a language integrated API. It extends the familiar `Graph` interface, common in static graph processing systems, with a simple set of additional operations, listed in table 1. This enables users to continue using existing static graph operations on any snapshot in the timelapse obtained using the `retrieve` API. **(§6.2)**

3.1 Evolving Graph Analytics Using Timelapse

In addition to providing ad-hoc access to any snapshot, timelapse is also useful in enabling efficient time-evolving graph analysis. The natural way to do graph computations over the time dimension is to iterate over a sequence of snapshots. For instance, an analyst interested in executing the `degrees` query on three snapshots, G_1, G_2 and G_3 depicted in fig. 2 can do:

```
for(id <- Array(G1,G2,G3))
  result = G.retrieve(id).degrees
```

However, applying the same operation on multiple snapshots of a time-evolving graph independently is inefficient. In graph-parallel systems (§2), `degrees()` computation is typically implemented using a user-defined program where every vertex sends a message with value 1 to their neighbors, and all vertices adding up their incoming message values. Such message exchange accounts for a non-trivial portion of the analysis time [62]. In the earlier example, sequentially applying the query to each snapshot results in 11 messages of which 5 are duplicates (fig. 2).

To avoid such inefficiencies, timelapse allows access to the *lineage* of graph entities. That is, it provides efficient

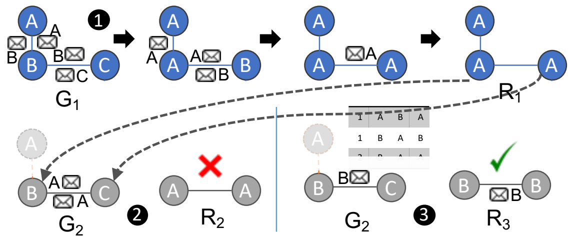


Figure 3: ❶ Connected components by label propagation on snapshot G_1 produces R_1. ❷ Vertex A and edge $A-B$ is deleted in G_2. Using the last result to bootstrap computation results in incorrect answer R_2. ❸ A strawman approach of storing all messages during the initial execution and replaying it produces correct results, but needs to store large amounts of state.

retrieval of the state of graph entities in any snapshot. Using this, graph-parallel phases can operate on the evolution of an entity (vertex or edge) as opposed to a single (at a given snapshot) value. In simple terms, each processing phase is able to see the history of the node's property changes. This allows *temporal* queries (§2.2) involving multiple snapshots, such as the degree computation, to be expressed as:

```
  results = G.degrees(Array(G1,G2,G3))
```

where `degrees` implementation takes advantage of timelapse by combining the phases in graph-parallel computation for these snapshots. That is, the user-defined vertex program is provided with state in all the snapshots. Thus, we are able to eliminate redundant messages and computation.

4 Computation Model

To improve interactivity, TEGRA must be able to efficiently execute queries by effectively reusing previous query results to reduce or eliminate redundant computations, commonly referred to as performing *incremental computation*. Here, we describe TEGRA's incremental computation model.

4.1 Incremental Graph Computations

Supporting incremental computation requires the system to manage *state*. The simplest form of state is the previous computation result. However, many graph algorithms are iterative in nature, where the graph-parallel stages are repeatedly applied in sequence until a fixed point. Here, simply restarting the computations from previous results do not lead to correct answers. To illustrate this, consider a connected components algorithm using label propagation on a graph snapshot, G_1 as shown in ❶ in fig. 3 which entails result R_1 after three iterations. When the query is to be repeated on G_2, restarting the computation from R_1 as shown in ❷ computes incorrect result. In general, correctness in such techniques depend on the properties of the algorithm (e.g., abelian group) and the monotonicity of updates (e.g., the graph only grows). Hence, supporting general non-monotonic iterative computations requires maintaining *intermediate* state.

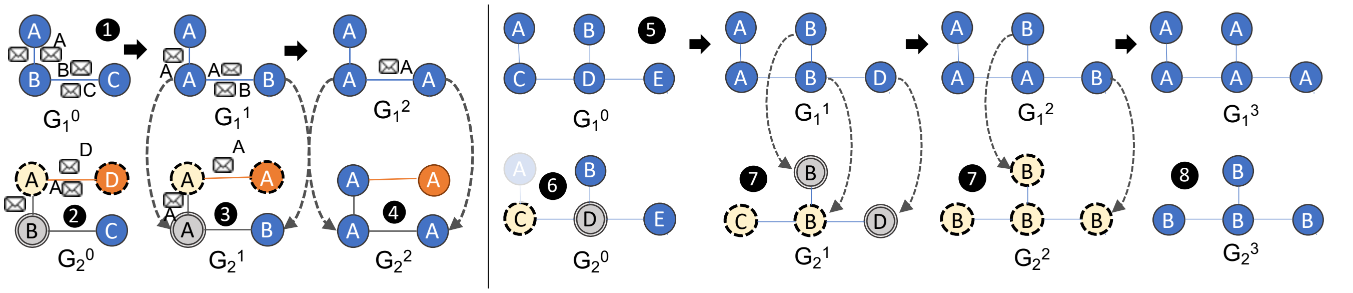


Figure 4: Two examples that depict how ICE works. Dotted circles indicate vertices that recompute, and double circles indicate vertices that need to be present in the subgraph to compute the correct answer, but do not recompute state themselves. ❶❺ Iterations of initial execution is stored in the timelapse. ❷❻ ICE bootstraps computation on a new snapshot, by finding the subgraph consisting of affected vertices and their dependencies (neighbors). In ❻, C is affected by the deletion of $A-C$. To recompute state it needs D (yields subgraph $C-D$). ❸❼ At every iteration, after execution of the computation on the subgraph, ICE copies state for entities that did not recompute. Then finds the new subgraph to compute by comparing the previous subgraph to the timelapse snapshot. In ❼, though C recomputes the same value as in previous iteration, its state is different from the snapshot in timelapse and hence needs to be propagated. ❹❽ ICE terminates when the subgraph converges and no entity in the graph needs the state copied from stored snapshots in the timelapse.

TEGRA proposes a general, incremental iterative graph-parallel computation model that is *algorithm independent*. It leverages the fact that graph-parallel computations proceed by making iterative changes to the original graph. Thus, *iterations of a graph-parallel computation can be seen as a time-evolving graph*, where the snapshots are the materialized state of the graph at the end of each iteration. In each of the snapshots, the intermediate state can be stored as vertex and edge properties. Since timelapse can efficiently store and retrieve these snapshots, we can perform incremental computations in a generic fashion by invoking graph-parallel computations on *affected neighborhood*. We call this model *Incremental Computation by entity Expansion* (ICE).

4.2 ICE Computation Model

ICE executes computations only on the subgraph that is affected by the updates *at each iteration*. To do so, it needs to find the relevant entities that should participate in computation at any given iteration. For this, it uses the state stored as timelapse, and the computation proceeds in four phases:

Initial execution: When an algorithm is executed for the first time, ICE stores the state (using the `save` API) of the vertices (and edges if the algorithm demands it) as properties in the graph. At the end of every iteration, a snapshot of the graph is added to the timelapse. The ID is generated using a combination of the graph's unique ID, an algorithm identifier and the iteration number. As depicted in ❶ and ❺ in fig. 4, the timelapse contains three and four snapshots, respectively.

Bootstrap: When the computation is to be executed on a new snapshot, ICE needs to bootstrap the incremental computation. Intuitively, the subgraph that must participate in the computation at bootstrap consists of the updates to the graph, and the *entities affected by the updates*. For instance, any newly added or changed vertices should be included. Similarly, edge modifications would result in the source and/or destination vertices to be included in the computation. However, the affected entities alone are not sufficient to ensure correctness of the results. This is because in graph-parallel execution, the state of a graph entity is dependent on the collective input from its neighbors. Thus, ICE must also include the one-hop neighbors of *affected entities*, and so the bootstrapped subgraph consists of the affected entities and their one-hop neighbors. ICE uses the `expand` API for this purpose. The graph computation is run on this subgraph. ❷ in fig. 4 shows how ICE bootstraps when a new vertex D and a new edge $A-D$ is added. D and A should recompute state, but for A to compute the correct state, it must involve its one-hop neighbor B, yielding subgraph $D-A-B$.

Iterations: At each iteration, ICE needs to find the right subgraph to perform computations. ICE exploits the fact that the nature of the graph-parallel abstraction restricts the propagation distance of updates in an iteration. Intuitively, the graph entities that might possibly have a different state at any iteration will be contained in the subgraph that ICE has already executed computation on from the last iteration. Thus, after the initial bootstrap, ICE can find the new subgraph at a given iteration by examining the changes to the subgraph from the previous iteration (using `diff`) and expanding to the one-hop neighborhood of affected entities (using `expand`). For the vertices/edges that did not recompute the state, ICE simply copies the state from the timelapse (using `merge`). For instance, in ❸ in fig. 4, though A and D recomputed, only D changed state and needs to be propagated to its neighbor A which needs B.

Termination: It is possible that modifications to the graph may result in more (or less) number of iterations compared to the initial execution. Unlike normal graph-parallel computations, ICE does not necessarily stop when the subgraph converges. If there are more iterations stored in the timelapse for the initial execution, ICE needs to check if the unchanged parts of the graph must be copied over. Conversely, if the subgraph has not converged and there are no more corresponding iterations, ICE needs to continue. To do so, it simply switches to normal (non-incremental) computation from that point. Thus, ICE converges only when the subgraph converges and no entity needs their state to be copied from the stored snapshot in the timelapse. (❹ and ❽ in fig. 4)

By construction, ICE generates the exact same intermediate states for all edges and vertices at all iterations, as compared to running a full execution on the entire graph. Thus, not only does ICE guarantee correctness of the incremental execution, but also enables *any* algorithm implemented in a graph-parallel fashion to be made incremental.

4.3 Improving ICE Model

Sharing State Across Different Queries Many graph algorithms consist of several stages of computations, some of which are common across different algorithms. For example, variants of connected components and pagerank algorithms both require the computation of vertex degree as one of the steps. Since ICE decouples state, such common computations can be stored as separate state that is shared across different queries. Thus, ICE enables developers to generate and *compose* modular states. This reduces the need to duplicate common state across queries which results in reduced memory consumption and better performance.

Incremental Computations Can Be Inefficient Incremental computation is not useful in all cases. For instance, in graphs with high degree vertices, a small change may result in a domino effect in terms of computation—that is, during later iterations, a large number of graph entities might need to participate in computation (e.g., Example 2 in fig. 4). To perform incremental computation, ICE needs to spend computations cycles to identify the set of vertices that should recompute (using `diff`) and copy the state of vertices that did not do computations (using `merge`). As a result, the total work done by the system may exceed that of completely executing the computation from scratch [24, 67]. Fortunately, the design of ICE lets us overcome this inefficiency. Since ICE generates the same intermediate states at every iteration as full re-execution, it can switch to full re-execution at any point.

A Simple Learning Based Model for Switching A key requirement for avoiding the inefficiencies with incremental execution detailed previously is to determine *when* to switch to full re-execution. This can be done at two places: at the start of the incremental execution, or at iteration boundaries (i.e., at the beginning of an iteration during the execution). In TEGRA, we picked the latter. A strawman approach is to use a simple threshold—for instance, *active* vertices in an iteration—to determine when to switch. Unfortunately, such approaches did not perform well in our evaluation, as we found that the optimal point for the switch depends on a number of factors, including the query, the properties of the graph and the nature of the modifications. Thus, we use a simple learning based approach to determining when TEGRA makes the switch.

In our approach, we train a simple random forest classifier [14] to predict, at the beginning of an iteration, if switching to full re-execution from that point would be faster compared to continuing with incremental execution. We do the training in an offline phase, where we use several runs of queries both in a full incremental and full re-execution mode as the input, ensuring enough runs in both cases to avoid class sensitivity. For each run, in every iteration, we record the following fields that we use as *features* for the learning: number of vertices that participate in the computation, the average degree of the active vertices, the number of partitions active, the number of messages generated per vertex, the number of messages received per vertex, the amount of data transferred over the network and the time taken for the iteration to complete. To make the learning general, we also use a few graph-specific characteristics such as the average degree of vertices, the average diameter and clustering coefficient. The label indicates whether switching to full recomputation in the next iteration resulted in faster execution.

While simple, we found that this approach works well as we show in fig. 11. Examination of the model revealed that it tries to learn the significance of vertices that participate in the computation (in terms of average degrees), the layout (how they are partitioned) and graph characteristics (in terms of diameter and clustering coefficient) in relation to the execution time. We plan to explore ways to improve our technique (e.g., better/robust models) as part of our future work.

5 Distributed Graph Snapshot Index (DGSI)

To make timelapse abstraction and ICE computation model practical, TEGRA needs to back them with a storage that satisfies the following three requirements: (1) enable ingestion of updates in real-time, and make it available for analysis in the minimum time possible, (2) support space-efficient storage of snapshots and intermediate computation state in a timelapse, and (3) enable fast retrieval and efficient operations on stored timelapses. These requirements, crucial for efficiently supporting ad-hoc analytics on time-evolving graphs, pose several challenges. For instance, they prohibit the use of preprocessing, typically employed by many graph processing systems, to compactly represent graphs and to make computations efficient. In this section, we describe how TEGRA achieves this by building DGSI. It addresses (1) and (2) by leveraging persistent data structures to build a graph store (§5.1, §5.2) that enables efficient operations (§5.3) while managing memory over time (§5.4).

5.1 Leveraging Persistent Data Structures

In TEGRA, we leverage persistent data structures [22] to build a distributed, versioned graph state store. The key idea in persistent data structures is to maintain the previous versions of data when modified, thus allowing access to earlier ver-

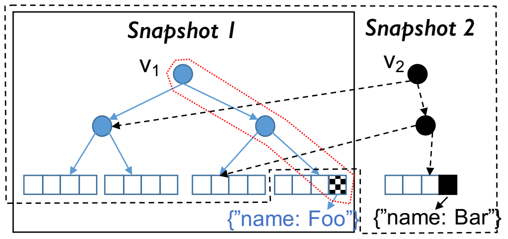


Figure 5: DGSI has one pART for vertices and one for edges in each partition. Version 1 (v_1) of a vertex pART stores properties of vertices in its leaves. Vertex id traverses the tree to its properties (e.g., *name*). Changes to vertices (e.g., property *name* changed from *Foo* to *Bar*) generates a new version v_2. The snapshot of the vertex pART before (Snapshot 1, shown in solid box) and after (Snapshot 2, shown in dotted box) the change share common leaves. (§5.2)

sions. DGSI uses a persistent version of the Adaptive Radix Tree [38] as its data structure. ART provides several properties useful for graph storage such as efficient updates and range scans. Persistent Adaptive Radix Tree (PART) [5] adds persistence to ART by simple path-copying. For the purpose of building DGSI, we reimplemented PART (hereafter pART) in Scala and made several modifications to optimize it for graph state storage. We also heavily engineered our implementation to avoid performance issues, such as providing fast iterators, avoiding unnecessary small object creation and optimizing path copying under heavy writes.

5.2 Graph Storage & Partitioning

TEGRA stores graphs using two pART data structures: a *vertex* tree and an *edge* tree. The vertices are identified by a 64-bit integer key. For edges, we allow arbitrary keys stored as byte arrays. By default, the edge keys are generated from their source and destination vertices and an additional short field for supporting multiple edges between vertex pairs. pART supports prefix matching, so using matching on this key enables retreiving all the destination edges of a given vertex. The leaves in the tree store pointers to arbitrary properties.We create specialized versions of pART to avoid (un)boxing costs when properties are primitive types.

TEGRA supports several graph partitioning schemes, similar to GraphX [26], to balance load and reduce communication. To distribute the graph across machines in the cluster, vertices are hash partitioned and edges are partitioned using one of many schemes (e.g., 2D partitioning). We do not partition the pART structures, instead TEGRA partitions the graph and creates *separate* pART structures locally in each partition. Hence logically, in each partition, the vertex and edge trees store a subgraph (fig. 5). By using local trees, we further amortize the (already low) cost[4] associated with modifying the tree upon graph updates.

To consume updates, TEGRA needs to send the updates to the right partition. Here, we impose the *same* partitioning as the original graph on the vertices/edges in the update.

[4]Modifications to nodes in ART trees only affect the $O(\log_{256} n)$ ancestors

5.3 Version Management

DGSI is a versioned graph state store. Every "version" corresponds to a root in the vertex and edge tree in the partitions—traversing the trees from the root pair materializes the graph snapshot. For version management, DGSI stores a mapping between a root and the corresponding "version id" in every partition. The version id is simply a byte array.

For operating on versions, DGSI exposes two low level primitives inspired by existing version management systems: `branch` and `commit`. A `branch` operation creates a new working version of the graph by creating a new (transient) root that points to the original root's children. Users operate on this newly created graph without worrying about conflicts because the root is exclusive to them and not visible in the system. Upon completing operations, a `commit` finalizes the version by adding the new root to version management and makes the new version available for other users in the system. Once a `commit` is done on a version, modifications to it can only be done by "branching" that version. Any timelapse based modifications cause `branch` to be called, and the timelapse `save` API invokes `commit`.

TEGRA can interface with external graph stores, such as Neo4J [4] or Titan [6] for importing and exporting graphs. While importing new graphs, DGSI automatically assigns an integer id (if not provided) and commits the version when the loading is complete. We create a version by batching updates. The batch size is user-defined. In order to be able to retrieve the state of the graph in between snapshots, TEGRA stores the updates between snapshots in a simple log file, and adds a pointer to this file to the root.

The simplest retrieval is by using its id. In every partition, DGSI then gets a handle to the root element mapped to this id, thus enabling operations on the version (e.g., branching, materialization). By design, versions in DGSI have no global ordering because branches can be created from any version at any time. However, in some operations, it may be desirable to have ordered access to versions, such as in incremental computations where the system needs access to the consecutive iterations. For this purpose, we enable suffix, prefix and simple ranges matching primitives on version id.

5.3.1 Implementing Timelapses

TEGRA implements timelapses using DGSI with its version ids and the matching primitives it provides. Recall that each timelapse logically represents a sequence of graph snapshots. Hence, every snapshot stored in DGSI is part of one or more timelapses. As a simple example, a user intending to track the Twitter graph by time could create snapshots by appending UNIX epoch to a unique ID for the graph (e.g., `TWTR`). In this scheme, a snapshot created at 9:00AM on 01/01/2020 may be given ID `TWTR_1577869200`. Prefix matching `TWTR` provides the entire timelapse for this graph. When this snapshot is chosen for a query, say PageRank, TEGRA can automatically append an algorithm ID (e.g., `PR`) and an iteration number to gener-

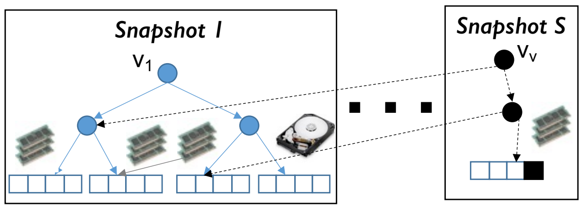


Figure 6: DGSI has fine-grained control over leaves (where data is stored). Here DGSI has 1000s of snapshots. All snapshots except *S* are on disk, their parents just hold pointers to files on disk. Parents are also dynamically written to disk if all of their children are on disk. Data structure uses adaptive leaf sizes for efficiency.

ate version IDs such as `TWTR_1577869200_PR_1`. Here, prefix matching on `TWTR_1577869200_PR` provides the timelapse of the execution of the page rank algorithm.

Currently, TEGRA only manages automatic ID generation for time-based snapshots and iterations of query execution. For instance, creating snapshots every hour is automated by doing a `branch` on the last snapshot in the `TWTR` timelapse and `commit`ing the new changes with new timestamp appended (e.g., `TWTR_1577872800`). Similarly, saving iterations of a query is as simple as keeping track and `branch` ing the last snapshot in the corresponding timelapse, and saving the new iteration with incremented iteration number (e.g., `TWTR_1577869200_PR_2`). However, since TEGRA stores IDs as byte arrays, users are free to choose any ID generation schemes; for instance it is possible to come up with complex hierarchical IDs that enable sophisticated retrievals using the matching capabilities on IDs provided by DGSI.

5.4 Memory Management

Over time, DGSI stores several versions of a graph, and hence TEGRA needs to manage these versions efficiently. We employ several ways to do this. Between `branch` and `commit` operations, it is likely that many transient child nodes are formed. We aggressively remove them during the `commit` operation. In addition, we enable in-place updates when the operations are local, such as after a branch and before a commit. Further, during ad-hoc analysis, analysts are likely to create versions that are never committed. We periodically mark such orphans and adjust the reference counting in our trees to make sure that they are garbage collected.

For managing stored versions, we leverage a simple Least Recently Used (LRU) eviction policy. Each time a version is accessed, we annotate the version and all its children with a timestamp. The system then employs a thread for periodically removing versions that were not accessed in a long time. The eviction is done by saving the version to local disk (or distributed file system). We do this in the following way. Since every version in DGSI is a branch, we write each subtree in that branch to a separate file and then point its root to the file identifier (e.g., in fig. 5, we can store v_2's leaf that is different from v_1 in disk as a file and point the parent node to this file). By writing subtrees to separate files, we ensure that different versions sharing tree nodes in memory can share tree nodes written to files. Due to this technique, we can ensure that leaf nodes (which are most memory consuming) that are specific to a version (not shared with any other version) are always written to disk if the version is evicted. As depicted in fig. 6, a large number of versions can be flushed to disk over time while still being retrievable when necessary. Thus, only active snapshots are fully materialized in memory, thereby allowing TEGRA to store several snapshots.

6 Implementation

TEGRA is a drop-in replacement for GraphX [26]. It uses the popular Gather-Apply-Scatter (GAS) [27] graph parallel model. We utilize the barrier execution mode to implement direct communication between tasks to avoid most Spark overheads. Spark provides fault tolerance by checkpointing inputs and operations for reconstructing the state. TEGRA provides coarse-grained fault tolerance by leveraging Spark's `rdd.checkpoint` semantics. Users can explicitly run checkpoint operation, upon which TEGRA flushes the contents in DGSI to persistent storage. We currently do not support fine-grained lineage-based fault tolerance provided by Spark.

6.1 ICE on GAS Model

As described in §4.2, the `diff()` API marks the candidates that must perform graph-parallel computation in a given iteration. In GAS decomposition [27], the `scatter()` function, invoked on `scatter_nbrs`, determines the set of active vertices which must perform computation. Starting with an initial candidate set (e.g., at bootstrap the changes to the graph, and at any iteration the candidates from the previous iteration) the `diff()` API uses `scatter_nbrs` (`EdgeDirection` in GraphX) in the user-defined vertex program to mark all necessary vertices for computation. We mark all `scatter_nbrs` of a vertex if its state differs from the previous iteration, or from the previous execution stored in the timelapse. For instance, a vertex addition must inspect all its neighbors (as defined by `scatter_nbrs`) and include them for computation.

The vertices in GAS parallel model perform computation using the user defined `gather()`, `sum()` and `apply()` functions, where `gather_nbrs` determine the set of neighbors to gather state from. The `expand` API enables correct `gather()` operations on the candidates marked for recomputation by also marking the `gather_nbrs` of the candidates. After the `diff` and `expand`, TEGRA has the complete subgraph on which the graph-parallel computation can be performed.

6.2 Using TEGRA as a Developer

TEGRA provides feature compatibility with GraphX, and expands the existing APIs in GraphX to provide ad-hoc analysis support on evolving graphs. It extends all the operators to operate on user-specified snapshot(s) (e.g., `Graph.vertices(id)` retrieves vertices at a given snapshot id, and `Graph.mapV([ids])` can apply a map function on vertices of

```
def IncPregel(g: Graph[V, E],
     prevResult: Graph[V, E],
     vprog: (Id, V, M) => V,
     sendMsg: (Triplet) => M,
     gather: (M, M) => M): Graph[V, E] = {
 iter = 0
 // Loop until no active vertices and nothing to copy
 // from previous results in timelapse.
 while (!converged) {
   // Restrict to vertices that should recompute
   val msgs: Collection[(Id, M)] =
     g.expand(g.diff(prevResult.retrieve(iter))).
      .aggregateMessages(sendMsg, gather)
   iter += 1
   // Receive messages and copy previous results
   g = g.leftJoinV(msgs).mapV(vprog)
        .merge(prevResult.retrieve(iter)).save(iter) }
 return g }
```

Listing 1: Implementation of incremental Pregel using TEGRA APIs.

the graph on a set of snapshots). Graph-parallel computation is enabled in GraphX using the `Graph.aggregateMessages()` (previously `mrTriplets()`) API. To use TEGRA, users incorporate the TEGRA API in table 1 in their normal, static (non-incremental) versions of the algorithm at places where graph's state is mutated. These are places where GraphX's `Graph.aggregateMessages()` is used.

GraphX further offers iterative graph-parallel computation support through a Pregel API which captures the GAS decomposition using repeated invocation of the `aggregateMessages` and `joinVertices` until a fixed point. Listing 1 shows how a user might use TEGRA APIs to implement an incremental version of Pregel. The code is reproduced from GraphX [26], with minimal changes to incorporate TEGRA APIs to store and retrieve state. In general, a developer can write incremental versions of any iterative graph parallel algorithm by using the TEGRA APIs along with `aggregateMessages`.

7 Evaluation

We have evaluated TEGRA through a series of experiments.

Comparisons: We compare TEGRA against many state-of-the-art systems (§2.3). For streaming system, we use GraphBolt [45] and the Rust implementation of Differential Dataflow (DD) [3]. Since we were unable to obtain an open source implementation of a temporal engine, we developed our version of Chronos [30] in GraphX [26], which we call Chlonos (Clone of Chronos) in this section. This implementation emulates the array based in-memory layout of snapshots and the incremental computation model in Chronos. We note that while Chronos supports updates to graphs by storing the temporal changes on disk, it uses a pre-processing step to create an in-memory layout which is used for every query. This in-memory layout does not support updates (§2.3) and needs to be recreated every time. We compare DGSI against GraphOne [35] and Aspen [21].

Dataset	Vertices / Edges
twitter [11]	41.6 M / 1.47 B
uk-2007 [12]	105.9 M / 3.74 B
Facebook Synthetic Data [2]	Varies / 5, 10, 50 B

Table 2: Datasets in our evaluation. M = Millions, B = Billions.

Evaluation Setup: All of our experiments were conducted on 16 commodity machines available as Amazon EC2 instances, each with 8 virtual CPU cores, 61GB memory, and 160GB SSDs. The cluster runs a recent 64-bit version of Linux. We use Differential Dataflow v0.10.0 and Apache Spark v2.4.4. We warm up the JVM before measurements. For single machine systems, we use a x1.32xlarge instance with 128 virtual CPUs and 2 TB memory to be comparable with our cluster.

Dataset & Workloads: We evaluate TEGRA on a number of real-world graphs depicted in table 2, with up to 50 billion edges. TEGRA creates default properties at vertices and edges to allow queries that compute on them (our comparisons do not support *arbitrary* properties). We use three standard, well understood, iterative graph algorithms with varying computation and state requirements, commonly used to evaluate graph processing systems as queries: Connected Components (CC), Page Rank (PR) [55] and Belief Propagation (BP) [74]. We run PR until a specific convergence or 20 iterations, whichever is lower. Note that while the queries in this section do not access the vertex and edge properties explicitly (i.e., queries do not ask for them), TEGRA depends on them extensively to store intermediate state (§4).

Caveats. While perusing the evaluation results, we wish to remind the reader a few caveats. Though many of the graphs we use fit in the memory of a modern machine, TEGRA is focused on ad-hoc analytics which requires storage of multiple snapshots (and computation state) of the graph. Further, ad-hoc analytics requires the use of property graphs. TEGRA supports edge and vertex properties and creates a default value, which increases the graph size by several magnitudes and also affects performance. Finally, DD's connected component uses the union-find based implementation (hard to fit in a vertex centric model) which is superior to TEGRA's label propagation based implementation.

7.1 Microbenchmarks

We first present experiments that highlight the effectiveness of DGSI. GraphBolt is excluded as it doesn't allow storing multiple versions or intermediate state. (§2.3).

Snapshot Retrieval Latency: We generate 1000 snapshots of the Twitter and UK graphs by randomly modifying (adding and removing equal number) 1% of the edges (no computations are performed) to emulate the evolution of the graph. Table 3 shows the average latency for 10 random retrievals with varying number of snapshots in the system.

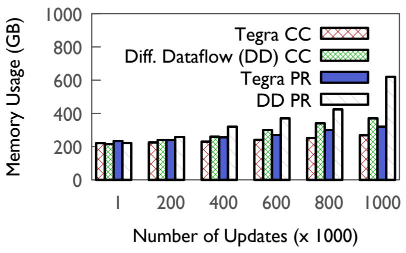


Figure 7: Differential dataflow generates state at every operator, while TEGRA's state is proportional to the number of vertices.

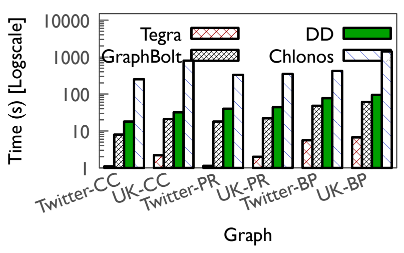


Figure 8: On ad-hoc queries on snapshots, TEGRA is able to significantly outperform due to state reuse.

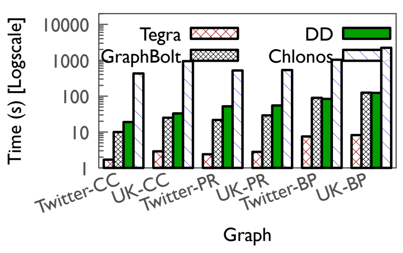


Figure 9: TEGRA's performance is superior on ad-hoc window operations even with materialization of results.

Graph	System	# Snapshots in memory				
		200	400	600	800	1000
Twitter	DD	30.2	44.8	92.4	131.6	180.2
	GraphOne	32.3	50.1	105.5	141.2	201.2
	Aspen	0.43	0.54	0.53	0.57	0.65
	TEGRA	1.34	1.5	1.25	1.78	2.1
UK	DD	63.1	86.4	220	271.5	283.2
	GraphOne	74.3	102.2	263.5	321.4	332.5
	Aspen	0.86	1.04	1.12	1.25	1.31
	TEGRA	2.2	2.5	2.6	2.25	2.3

Table 3: Snapshot retrieval latency (in seconds). DD and GraphOne require reconstruction, while TEGRA and Aspen can simply traverse the data structure from a version's root.

We see that TEGRA is able to return the queried snapshot within seconds due to DGSI which stores and retrieves materialized snapshots efficiently. In contrast, DD needs to reconstruct the graph from indexed differences and takes several minutes. It also exhibits high variance in retrieval time based on the amount of reconstruction required. GraphOne faces similar challenge with its `get-prior-edges()` API which needs to reconstruct the historic view from the durable edge log. While not shown, Chlonos too exhibits similar characteristics as DD but is significantly slower, in some cases up to an order of magnitude. This is because it stores updates on disk and needs an intensive preprocessing step to create the in-memory layout. DD exhausts the memory available in our cluster ($\approx$1TB) in this experiment which limited the number of snapshots we could store to $\approx$1000. One solution is to store the updates in a persistent storage, but this incurs significant performance degradation while retrieving (like Chlonos). Aspen performs similar to TEGRA (slightly faster since it is able to compress the graph significantly better) due to its use of persistent data structures, hence it only needs to traverse the tree from a specific root to retrieve a version (like DGSI). However, it neither supports intermediate state storage nor includes an incremental computation model.(§2.3)

Computation State Storage Overhead: To measure the memory overhead due to computation state, we perform PR and CC computation on the Twitter graph in an incremental fashion, where we add and delete 1000 edges to create a snapshot. We note the memory usage by each system after every 200 such computations until 1000 computations (for a total of 1 million edge updates). Figure 7 shows this experiment's results. When the number of updates is small, both TEGRA and DD use comparable amount of memory to store the state, even with DD's highly compact layout (native arrays compared to TEGRA's property graph). However, DD's state size increases rapidly as it does more computation and takes up to 2$\times$ that of TEGRA. TEGRA's memory requirement also increases over time, but much more gracefully. This is due to the combined effect of TEGRA's compact state representation (proportional to the number of vertices) and the ability of DGSI to manage memory efficiently (§5.4), while DD needs to keep state (proportional to the number of edges) at every operator. The amount of increase also depends on the algorithm. For instance, page rank generates the same amount of state in every iteration while connected component's state requirement reduces over iterations. Note that DD uses compaction in this experiment which is automatically done by the system. While GraphBolt also reduces the state requirement to be proportional to the number of vertices, it does not allow storing computational state for later reuse.

7.2 Ad-hoc Window Operations

Here, we present evaluations that focus on TEGRA's main goal. In these experiments, we emulate an analyst performing ad-hoc analytics. We load the graph, and apply a large number of sequential updates to it, where each update modifies 0.1% of the edges (adds and removes equal number) to adhere to our assumption that during ad-hoc analysis the graph doesn't change much and it is possible to leverage incremental computation (we show results with large changes in fig. 11). We then retrieve 100 random windows of the graph that are close-by, and apply queries in each. We assume that *some* earlier results are available so that the system could do incremental computations. We do not consider the window retrieval time in this experiment for any system. We present the average time taken to compute the query result.

Single Snapshot Operations: In the first experiment, we set the window size to zero so that every window retrieval returns a single snapshot. The results are depicted in fig. 8. DD,

Graph	5B			10B			50B		
	PR	CC	BP	PR	CC	BP	PR	CC	BP
DD	1m	8s	1.5m	2m	34s	-	-	-	-
GraphBolt	29s	21s	1.1m	1.2m	28s	2.2m	5.3m	54s	12m
TEGRA	10s	5s	6.5s	19s	7s	9.3s	1.5m	18s	2.4m

Table 4: Ad-hoc analytics on big graphs, with 5 billion, 10 billion and 50 billion edges. A '-' indicates the system failed to run the workload. TEGRA can handle big graphs and large amounts of state due to its efficient memory management (§5.4)

GraphBolt and Chlonos do not allow reusing computation across queries, so they compute from scratch for every retrieval. In contrast, TEGRA is able to leverage the compact computation state stored in its DGSI from earlier queries to do incremental computation. In this case, most of the snapshots incur no computation overhead because of the small amount of changes between them, and TEGRA is able to produce an answer within a few seconds. DD and GraphBolt take a few 10s of seconds, while Chlonos requires 100s of seconds. TEGRA's benefits range from 18-30× compared to DD and 8-18× compared to GraphBolt.

Window Operations: Here we set the window size to be 10 snapshots. GraphBolt, Chlonos and DD are able to apply incremental computations once the query has been computed on the first snapshot. Figure 9 shows the results. We see that DD is fast once the first result has been computed. This is due to the combination of its extremely efficient streaming computation model (no materialization) and recent optimizations such as shared arrangements [48]. Chlonos incurs a penalty initially because it uses the first result to bootstrap the rest using its LABS model. TEGRA's performance remains consistent. This is due to two reasons. First, since TEGRA separates state from computation, it can reuse the state across multiple snapshots. Second, due to the use of persistent data structures, snapshot can be independently and concurrently operated on (§3.1). Since GraphBolt does not support concurrent processing, it does sequential computation (in an incremental fashion) on the snapshots. For simple queries (e.g., CC), the penalty is unnoticeable. It becomes pronounced in BP which is more computationally heavy. TEGRA is still 9-17× (5-23×) faster compared to DD (GraphBolt).

Large Graphs & Large Amounts of State: Here we answer two questions: (1) can TEGRA support ad-hoc analysis on large graphs, and (2) can TEGRA efficiently manage memory when large amounts of state need to be stored? We use synthetic graphs provided by Facebook [2] modeled using the social network's properties. We execute the queries once on the original graph, then modify the graph by a tiny percentage (0.01%) randomly 1000 times to create 1000 snapshots. We then pick a snapshot, run the queries on it and provide the average of 100 such runs in table 4. DD works well when both the graph and the updates (and the generated state) are small. However, as the graph becomes larger, DD needs to push a large number of updates through the computation, and state becomes a bottleneck in its performance. On the largest graph, we were unable to get DD to work as it failed due to excessive memory usage during initial execution. GraphBolt doesn't store any previous state, and hence is unable to do incremental computations. It also required several optimizations to support the largest graph. In contrast, TEGRA is not only able to efficiently use memory and disk (§5.4) and scale to large graphs and snapshots, but also provide significant benefits by using previous computation state.

		Twitter			UK		
		1K	**10K**	**100K**	**1K**	**10K**	**100K**
CF	GraphBolt	15.1	15.3	15.0	21.5	21.8	21.7
	TEGRA	1.2	1.3	1.5	1.4	1.4	1.6
CoEM	GraphBolt	17.1	17.6	17.8	28.1	28.6	28.9
	TEGRA	1.7	1.8	2.1	1.8	1.8	2.3
LP	GraphBolt	22.1	22.4	22.6	30.1	30.2	32.4
	TEGRA	1.8	1.9	1.9	2.0	2.2	2.3
TC	GraphBolt	68.1	68.5	69.2	5.2	5.4	5.7
	TEGRA	0.10	0.12	0.14	0.11	0.15	0.25
BFS	GraphBolt	0.6	0.7	0.8	0.8	0.8	0.9
	TEGRA	0.15	0.16	0.16	0.21	0.21	0.25
k-hop (4-hop)	GraphBolt	1.1	1.1	1.2	1.2	1.2	1.3
	TEGRA	0.6	0.6	0.55	0.7	0.7	0.8

Table 5: Running time (in seconds) for TEGRA and GraphBolt when doing ad-hoc analysis with different batch sizes and algorithms.

Effect of Batch Size & Additional Algorithms: In this experiment, we evaluate the effect of batch size on the ad-hoc analysis capability of TEGRA. For this, we fix the batch size to a specific number in each run, and use several other algorithms. Specifically, we use Label Propagation (LP), Collaborative Filtering (CF) and Triangle Count (TC) and Co-Training Expectation Maximization (CoEM) as used in GraphBolt [45]. For CoEM, we use the Latent Dirichlet Allocation (LDA) implementation in GraphX which uses EM. We also provide results on k-hop, which computes the set of vertices that are k hops away, and Breadth First Search (BFS). For the k-hop algorithm, we set k to 4 for all batch sizes.

In each run, we execute the algorithm first. We then generate several snapshots using varying batches of equal edge additions and deletions. We choose three fixed numbers: 1K, 10K and 100K. We pick a random snapshot and repeat the same algorithm on it. The results are shown in table 5.TEGRA is able to perform incremental computation using previous results, while GraphBolt does not support ad-hoc analysis and hence need to execute the algorithm fully. We also notice that varying batch size doesn't affect TEGRA much, and that it is able to provide results efficiently.

We note a few caveats here. In TC, the incremental computations are simple (edge additions and deletions do not result in multiple iterations) and involves just updating a count based on the edges added or deleted. Similarly, BFS and 4-hop algorithms are light weight and result in only a very small part of the graph to be active, especially during incremental computation. Due to this reason, for these algorithms, we only measure the actual computation time and ignore the scheduling overhead in TEGRA. Hence, the times we report for these

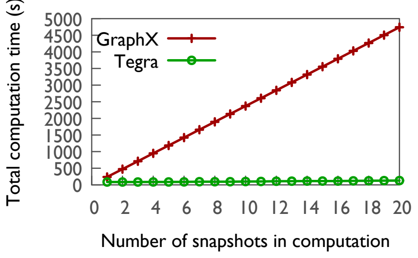


Figure 10: Timelapse lets queries to be executed simultaneously on a sequence of snapshots, enabling efficient temporal analysis.

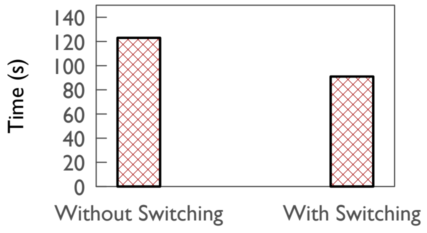


Figure 11: TEGRA can switch to full re-execution when incremental computations are not useful.

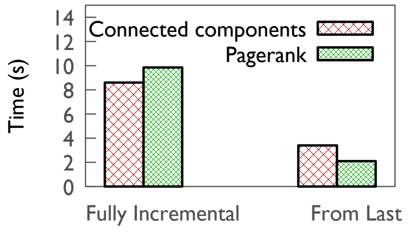


Figure 12: Monotonicity of updates (additions only) can be leveraged to speed up computations by starting from the last answer.

three algorithms are not end-to-end, but just the time it takes for the computation task to complete.

7.3 Timelapse & ICE

Parallel computations. Here we evaluate the ability of Timelapse to do temporal queries, where a query is applied to a sequence of snapshots in parallel (§3.1). We create 20 snapshots of the Twitter graph by starting with 80% of the edges and adding 1% to it repeatedly. We run the connected components algorithm, each time varying the number of snapshots on which the computation is run. In each run, we measure the time taken to obtain the results on all the snapshots considered. For comparison, we use GraphX and apply the algorithm to each snapshot in a serial fashion. The results are depicted in fig. 10. We see that TEGRA significantly outperforms GraphX. The improvement for a single snapshot is due to TEGRA's optimizations and use of barrier execution mode in Spark. Further, we see a linear trend with increasing number of snapshots. By sharing computation and communication, TEGRA is able to achieve up to 36× speedup.

ICE's switching capability: To test ICE's switching capability when incremental computations are not useful (§4.3), we run the CC algorithm on the Twitter graph. Next, we introduce a batch of deletions in the largest components so that incremental computation executes on a large portion of the graph. We then make TEGRA recompute with and without the switching enabled and average the results over 10 such runs. The results are shown in fig. 11. We see that without the switching, TEGRA incurs a penalty—the incremental execution takes more time than complete re-execution of the algorithm. With switching, TEGRA is easily able to identify that it needs to switch and hence does not incur this penalty.

ICE's versatility: Since ICE differs from streaming engines, it can also provide flexibility in how it uses state. For instance, if updates are monotonic (only additions), then ICE can simply restart from the last answer instead of using full incremental computations. Figure 12 shows this on two algorithms on the UK graph. PR and CC can benefit, but PR is faster since it only needs to converge within a given tolerance.

Sharing state across queries: To evaluate how much benefits sharing state between different queries provides, we run an experiment with CC and PR. For these queries, the degree computation can be shared. We run the algorithms with and without sharing enabled on the Twitter graph, and average the results of 10 runs of incremental computations on random snapshots. The results in fig. 13 show 20% and 30% reduction in memory usage and runtime.

7.4 TEGRA Shortcomings

Finally, we ask "What does TEGRA **not** do well?".

Purely Streaming Analysis: We consider an online query (§2) of CC. To emulate a streaming graph, we first perform CC computation on the graph. Then we continuously change 0.01% of the graph by adding and deleting equal number of edges. After fixed number of changes (every 200), we show the average runtime of 10 runs in fig. 14. We see that DD and GraphBolt are significantly better than TEGRA for such workloads. This is due to a combination of DD and GraphBolt optimized for online queries (pushing small updates really fast through computation) and their Rust/C++ implementation. We remind the reader of two caveats here. First, DD uses a much superior union-find approach to CC while TEGRA and GraphBolt use an iterative approach. Second, TEGRA only executes queries when it is asked to, whereas DD and GraphBolt executes queries for every batch of updates (thus TEGRA accumulates more updates when executing queries). While TEGRA can theoretically process each small update separately, the computation engine it builds on (Spark) is tuned for batched updates.

Purely Temporal Analysis: We assume that the queries and the window are known, and the system has optimized the data layout. We run a query on a window size of 10 and compare TEGRA and Chlonos on the incremental processing time (we discard the time for full execution). Excluding processing time, fig. 15 shows that TEGRA incurs a 15% performance hit due to its use of tree structure.

COST Analysis: The COST metric [47] is not designed for incremental systems, but we note that TEGRA is able to match the performance of an optimized single threaded implementation using 4 machines, each with 8 cores and has a COST of 32 cores. However, TEGRA uses property graphs while the optimized implementation does not.

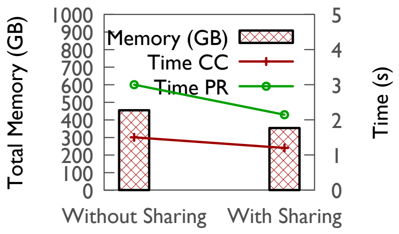


Figure 13: Sharing state across queries leads to reduction in memory usage and improvement in performance.

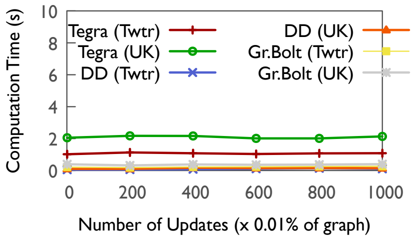


Figure 14: Both DD and GraphBolt significantly outperform TEGRA for purely streaming analysis.

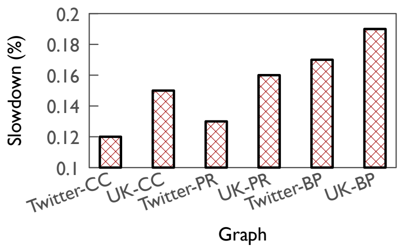


Figure 15: Tree based data structures in TEGRA result in a 15% slowdown compared to Chlonos for purely temporal analysis.

8 Related Work

Analytics on Static Graphs: A large number of graph processing systems [7, 8, 16, 18, 25–28, 37, 39–41, 56, 57, 59, 60, 65, 68–73, 76–81] focus on static graph processing, some of which are single machine systems and some are distributed. These systems do not consider evolving graph workloads.

(Transactional) Graph Stores: The problem of managing time-evolving graph has been studied in the context of graph stores [15,52,53,56]. These focus on optimizing point queries which retrieves graph entities and do not support storing multiple snapshots. This yields a different set of challenges compared to iterative graph analytics.

Managing Graph Snapshots: DeltaGraph [33] proposes a hierarchical index that can manage multiple snapshots of a graph using deltas and event lists for efficient retrievals, but lacks the ability to do windowed iterative analytics. TAF [34] fixes this, but it is a specialized framework that does not provide a generalized incremental model or ad-hoc operations. LLAMA [42] uses a multi-version array to support incremental ingestion. It is a single machine system, and it is unclear how the multi-version array can be extended to support data parallel operations required for iterative analytics. Version Traveler [32] achieves switching between snapshots of a graph by loading the common subgraph in the compressed-sparse-row format and extending it with deltas. It does not support incremental computation. GraphOne [35, 36] uses dual-versioning to provide access to recent snapshots. It doesn't support ad-hoc analysis or efficient retrieval of arbitrary snapshots. Aspen [21] leverages functional data structures to build a compressed streaming graph engine, but doesn't support incremental computations. Chronos [30] and ImmortalGraph [50] optimizes for efficient computation across a series of snapshots. They propose an efficient model for processing temporal queries, and support snapshot storage of the graph on-disk using a hybrid model. While their technique reduces redundant computations in a given query, they cannot store and reuse intermediate computation results. Their in-memory layout of snapshots requires preprocessing and cannot support updates. None of these systems support storing computation state for later reuse.

Incremental Maintenance on Evolving Graphs: Kineograph [19] supports constructing consistent snapshots of an evolving graph for streaming computations but does not allow ad-hoc analysis. WSP [75] focuses on streaming RDF queries. GraphInc [17] supports incremental graph processing using memoization of the messages in graph parallel computation, but does not support snapshot generation or maintenance. Kickstarter [67] and GraphBolt [45] support non-monotonic computations, but do not support ad-hoc analysis or compactly storing graph and state. Differential Dataflow [48, 49, 51, 54] leverages indexed differences of data in its computation model to do non-monotonic incremental computations. However, it is challenging to do ad-hoc window operations using indexed differences (§2.3). As we demonstrate in our evaluation, compactly representing graph and computation state is the key to efficient ad-hoc window operations on evolving graphs.

Incremental View Maintenance (IVM): In databases, IVM algorithms [10, 29] maintain a consistent view of the database by reuse of computed results. However, they are tuned for different kinds of queries and not iterative graph computations. Further, they generate large intermediate state and hence require significant storage and computation cost [49].

Versioned File Systems (e.g., [64]) allow several versions of a file to exist at a time. However, they are focused on disk based files in contrast to in-memory efficiency.

9 Conclusion

In this paper, we present TEGRA, a system that enables efficient ad-hoc window operations on evolving graphs. The key to TEGRA's superior performance in such workloads is a compact, in-memory representation of both graph and intermediate computation state, and a computation model that can utilize it efficiently. For this, TEGRA leverages persistent data structures and builds DGSI, a versioned, distributed graph state store. It further proposes ICE, a general, non-monotonic iterative incremental computation model for graph algorithms. Finally, it enables users to access these states via a natural abstraction called Timelapse. Our evaluation shows that TEGRA is able to outperform existing temporal and streaming graph systems significantly on ad-hoc window operations.

Acknowledgements

We thank all our reviewers and our shepherd, Christopher Rossbach for the valuable feedback. In addition to NSF CISE Expeditions Award CCF-1730628, this research is supported by gifts from Amazon Web Services, Ant Group, CapitalOne, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk and VMware.

References

[1] Graph dbms increased their popularity by 500 http://db-engines.com/en/blog_post//43, 2015 (accessed March 2021).

[2] A comparison of state-of-the-art graph processing systems. https://code.facebook.com/posts/319004238457019/a-comparison-of-state-of-the-art-graph-processing-systems/, 2016 (accessed March 2021).

[3] Differential dataflow rust implementation. https://github.com/TimelyDataflow/differential-dataflow, (accessed March 2021).

[4] Neo4j. http://www.neo4j.com, (accessed March 2021).

[5] Persistent adaptive radix tree. https://github.com/ankurdave/part, (accessed March 2021).

[6] Titan distributed graph database. http://thinkaurelius.github.io/titan/, (accessed March 2021).

[7] Zhiyuan Ai, Mingxing Zhang, Yongwei Wu, Xuehai Qian, Kang Chen, and Weimin Zheng. Squeezing out all the value of loaded data: An out-of-core graph processing system with reduced disk i/o. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 125–137, Santa Clara, CA, 2017. USENIX Association.

[8] Zhiyuan Ai, Mingxing Zhang, Yongwei Wu, Xuehai Qian, Kang Chen, and Weimin Zheng. Squeezing out all the value of loaded data: An out-of-core graph processing system with reduced disk i/o. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 125–137, Santa Clara, CA, 2017. USENIX Association.

[9] Leman Akoglu, Hanghang Tong, and Danai Koutra. Graph based anomaly detection and description: a survey. *Data Mining and Knowledge Discovery*, 29(3):626–688, 2015.

[10] Jose A. Blakeley, Per-Ake Larson, and Frank Wm Tompa. Efficiently updating materialized views. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, SIGMOD '86, pages 61–71, New York, NY, USA, 1986. ACM.

[11] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World Wide Web*. ACM Press, 2011.

[12] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[13] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, pages 13–16, New York, NY, USA, 2012. ACM.

[14] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[15] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. Tao: Facebook's distributed data store for the social graph. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60, San Jose, CA, 2013. USENIX.

[16] Aydin Buluç and John R. Gilbert. The combinatorial BLAS: design, implementation, and applications. *IJHPCA*, 25(4):496–509, 2011.

[17] Zhuhua Cai, Dionysios Logothetis, and Georgos Siganos. Facilitating real-time graph mining. In *Proceedings of the Fourth International Workshop on Cloud Data Management*, CloudDB '12, pages 1–8, New York, NY, USA, 2012. ACM.

[18] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 1:1–1:15, New York, NY, USA, 2015. ACM.

[19] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: Taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 85–98, New York, NY, USA, 2012. ACM.

[20] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proc. VLDB Endow.*, 8(12):1804–1815, August 2015.

[21] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. Low-latency graph streaming using compressed purely-functional trees. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 918–934, New York, NY, USA, 2019. Association for Computing Machinery.

[22] James R Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. Making data structures persistent. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 109–121. ACM, 1986.

[23] Stephen Eubank, Hasan Guclu, VS Anil Kumar, Madhav V Marathe, et al. Modelling disease outbreaks in realistic urban social networks. *Nature*, 429(6988):180, 2004.

[24] Wenfei Fan, Chunming Hu, and Chao Tian. Incremental graph computations: Doable and undoable. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 155–169, New York, NY, USA, 2017. ACM.

[25] P. Gao, M. Zhang, K. Chen, Y. Wu, and W. Zheng. High performance graph processing with locality oriented design. *IEEE Transactions on Computers*, 66(7):1261–1267, July 2017.

[26] Joseph Gonzalez, Reynold Xin, Ankur Dave, Daniel Crankshaw, and Ion Franklin, Stoica. Graphx: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, Broomfield, CO, October 2014. USENIX Association.

[27] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.

[28] Samuel Grossman, Heiner Litz, and Christos Kozyrakis. Making pull-based graph processing performant. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, pages 246–260, New York, NY, USA, 2018. ACM.

[29] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, pages 157–166, New York, NY, USA, 1993. ACM.

[30] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. Chronos: A graph engine for temporal graph analysis. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 1:1–1:14, New York, NY, USA, 2014. ACM.

[31] Anand Iyer, Li Erran Li, and Ion Stoica. Celliq : Real-time cellular network analytics at scale. In *Proceedings of the 12th USENIX conference on Networked Systems Design and Implementation*, NSDI'15, Berkeley, CA, USA, 2015. USENIX Association.

[32] Xiaoen Ju, Dan Williams, Hani Jamjoom, and Kang G. Shin. Version traveler: Fast and memory-efficient version switching in graph processing systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 523–536, Denver, CO, 2016. USENIX Association.

[33] U. Khurana and A. Deshpande. Efficient snapshot retrieval over historical graph data. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 997–1008, April 2013.

[34] Udayan Khurana and Amol Deshpande. Storing and analyzing historical graph data at scale. *CoRR*, abs/1509.08960, 2015.

[35] Pradeep Kumar and H. Howie Huang. Graphone: A data store for real-time analytics on evolving graphs. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 249–263, Boston, MA, February 2019. USENIX Association.

[36] Pradeep Kumar and H. Howie Huang. Graphone: A data store for real-time analytics on evolving graphs. *ACM Trans. Storage*, 15(4), January 2020.

[37] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 31–46, Hollywood, CA, 2012. USENIX.

[38] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, pages 38–49, Washington, DC, USA, 2013. IEEE Computer Society.

[39] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new framework for parallel machine learning. In Peter Grünwald and Peter Spirtes, editors, *UAI*, pages 340–349. AUAI Press, 2010.

[40] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woon-Hak Kang, Mohan Kumar, and Taesoo Kim. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, pages 527–543, 2017.

[41] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 527–543, New York, NY, USA, 2017. ACM.

[42] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer. Llama: Efficient graph analytics using large multiversioned arrays. In *2015 IEEE 31st International Conference on Data Engineering*, pages 363–374, April 2015.

[43] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[44] Jasmina Malicevic, Baptiste Lepers, and Willy Zwaenepoel. Everything you always wanted to know about multicore graph processing but were afraid to ask. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 631–643, Santa Clara, CA, 2017. USENIX Association.

[45] Mugilan Mariappan and Keval Vora. Graphbolt: Dependency-driven synchronous processing of streaming graphs. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 25:1–25:16, New York, NY, USA, 2019. ACM.

[46] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Comput. Surv.*, 48(2):25:1–25:39, October 2015.

[47] Frank McSherry, Michael Isard, and Derek G. Murray. Scalability! but at what cost? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, May 2015. USENIX Association.

[48] Frank McSherry, Andrea Lattuada, Malte Schwarzkopf, and Timothy Roscoe. Shared arrangements: practical inter-query sharing for streaming dataflows. VLDB, 2020.

[49] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow. In *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*, 2013.

[50] Youshan Miao, Wentao Han, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Enhong Chen, and Wenguang Chen. Immortalgraph: A system for storage and analysis of temporal graphs. *Trans. Storage*, 11(3):14:1–14:34, July 2015.

[51] Microsoft Naiad Team. GraphLINQ: A graph library for naiad. http://bigdataatsvc.wordpress.com/2014/05/08/graphlinq-a-graph-library-for-naiad/, 2014.

[52] Jayanta Mondal and Amol Deshpande. Eagr: Supporting continuous ego-centric aggregate queries over large dynamic graphs. *CoRR*, abs/1404.6570, 2014.

[53] Jayanta Mondal and Amol Deshpande. Stream querying and reasoning on social data. In *Encyclopedia of Social Network Analysis and Mining*, pages 2063–2075, 2014.

[54] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA, 2013. ACM.

[55] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[56] Vijayan Prabhakaran, Ming Wu, Xuetian Weng, Frank McSherry, Lidong Zhou, and Maya Haradasan. Managing large graphs on multi-cores with graph awareness. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 41–52, Boston, MA, 2012. USENIX.

[57] Abdul Quamar, Amol Deshpande, and Jimmy Lin. Nscale: Neighborhood-centric large-scale graph analytics in the cloud. *The VLDB Journal*, 25(2):125–150, April 2016.

[58] Real use case at Redacted, tier-1 cellular provider in USA. Company name redacted due to authors NDA with the company.

[59] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 410–424, New York, NY, USA, 2015. ACM.

[60] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 472–488, New York, NY, USA, 2013. ACM.

[61] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. The ubiquity of large graphs and surprising challenges of graph processing. *Proc. VLDB Endow.*, 11(4):420–431, December 2017.

[62] Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, M. Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey. Navigating the maze of graph analytics frameworks using massive graph datasets. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 979–990, New York, NY, USA, 2014. ACM.

[63] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 505–516, New York, NY, USA, 2013. ACM.

[64] Craig AN Soules, Garth R Goodson, John D Strunk, and Gregory R Ganger. Metadata efficiency in a comprehensive versioning file system. 2002.

[65] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. Arabesque: A system for distributed graph mining - extended version. *CoRR*, abs/1510.04233, 2015.

[66] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From "think like a vertex" to "think like a graph". *Proc. VLDB Endow.*, 7(3):193–204, November 2013.

[67] Keval Vora, Rajiv Gupta, and Guoqing Xu. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 237–251, New York, NY, USA, 2017. ACM.

[68] Guozhang Wang, Wenlei Xie, Alan J Demers, and Johannes Gehrke. Asynchronous large-scale graph processing made easy. In *CIDR*, 2013.

[69] Guozhang Wang, Wenlei Xie, Alan J. Demers, and Johannes Gehrke. Asynchronous large-scale graph processing made easy. In *CIDR*. www.cidrdb.org, 2013.

[70] Ming Wu and Rong Jin. A graph-based framework for relation propagation and its application to multi-label learning. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '06, pages 717–718, New York, NY, USA, 2006. ACM.

[71] Ming Wu, Fan Yang, Jilong Xue, Wencong Xiao, Youshan Miao, Lan Wei, Haoxiang Lin, Yafei Dai, and Lidong Zhou. Gram: Scaling graph computation to the trillions. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 408–421, New York, NY, USA, 2015. ACM.

[72] Wencong Xiao, Jilong Xue, Youshan Miao, Zhen Li, Cheng Chen, Ming Wu, Wei Li, and Lidong Zhou. Tux^2: Distributed graph computation for machine learning. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 669–682, Boston, MA, 2017. USENIX Association.

[73] Wenlei Xie, Guozhang Wang, David Bindel, Alan Demers, and Johannes Gehrke. Fast iterative graph computation with block updates. *Proc. VLDB Endow.*, 6(14):2014–2025, September 2013.

[74] Jonathan S Yedidia, William T Freeman, and Yair Weiss. Generalized belief propagation. In *Advances in neural information processing systems*, pages 689–695, 2001.

[75] Haibo Chen Yunhao Zhang, Rong Chen. Sub-millisecond stateful stream querying over fast-evolving linked data. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17. ACM, 2017.

[76] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian. Graphp: Reducing communication for pim-based graph processing with efficient data partition. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 544–557, Feb 2018.

[77] Mingxing Zhang, Yongwei Wu, Kang Chen, Xuehai Qian, Xue Li, and Weimin Zheng. Exploring the hidden dimension in graph processing. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 285–300, GA, 2016. USENIX Association.

[78] Mingxing Zhang, Yongwei Wu, Kang Chen, Xuehai Qian, Xue Li, and Weimin Zheng. Exploring the hidden dimension in graph processing. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 285–300, Savannah, GA, 2016. USENIX Association.

[79] Mingxing Zhang, Yongwei Wu, Youwei Zhuo, Xuehai Qian, Chengying Huan, and Kang Chen. Wonderland: A novel abstraction-based out-of-core graph processing system. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 608–621, New York, NY, USA, 2018. ACM.

[80] Y. Zhang, V. Kiriansky, C. Mendis, S. Amarasinghe, and M. Zaharia. Making caches work for graph analytics. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 293–302, Dec 2017.

[81] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 301–316, GA, 2016. USENIX Association.

Unifying Timestamp with Transaction Ordering for MVCC with Decentralized Scalar Timestamp

Xingda Wei, Rong Chen, Haibo Chen, Zhaoguo Wang, Zhenhan Gong, Binyu Zang
Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University

Abstract

This paper presents DST, a *decentralized scalar timestamp* scheme to scale distributed transactions using multi-version concurrency control (MVCC). DST is efficient in storage and network by being a scalar timestamp but requiring no centralized timestamp service for coordination, which may become a scalability bottleneck. The key observation is that concurrency control (CC) protocols like OCC and 2PL already imply a serializable order among concurrent read-write transactions through conflicting database tuples. To this end, DST piggybacks on CC protocols to maintain the timestamp ordering with low cost and no new scalability bottleneck for read-write transactions. DST further provides snapshot reads with bounded staleness by using a hybrid scalar timestamp (physical clock and logical counter).

To demonstrate the generality of DST, we provide a general guideline for the integration of DST and further show the effectiveness by using three representative transactional systems (i.e., DrTM+R, MySQL cluster, and ROCOCO) with different CC protocols. Experimental results show that DST can achieve more than 95% of optimal performance (using Read Committed) without compromising correctness. With DST, DrTM+R achieves up to 1.8X higher peak throughput for `TPC-E` and outperforms other timestamp schemes by 6.3X for `TPC-C`. DST also leads up to 1.9X and 2.1X speedup on `TPC-C` for MySQL cluster and ROCOCO, respectively.

1 Introduction

Many large-scale applications like Web services, stock exchange, and e-commerce require accessing scalable sharded data stores in a consistent way. Among such accesses, a large fraction requires consistently scanning data over many shards despite concurrent updates on the fly. For example, an examination of `TPC-E` [57], a sophisticated online transaction processing benchmark that models stock exchange, uncovers that 79% of transactions are read-only ones at run time. It was also reported that 99.8% of accesses to Facebook's distributed data store TAO are reads [16], which need strong consistency along with transactional writes [7].

However, it is costly to provide transactional isolation to read-only transactions [39] because a user read request may result in thousands of sub-queries [7]. Pessimistically executing a read-only transaction may cause unnecessary blocking to itself and concurrent read-write transactions, while optimistically executing it is likely to cause excessive aborts. For instance, as shown in Fig. 1(a), there is a notable performance gap between using optimistic concurrency control (OCC) [29] and read committed (RC) protocol for `TPC-E`.[1]

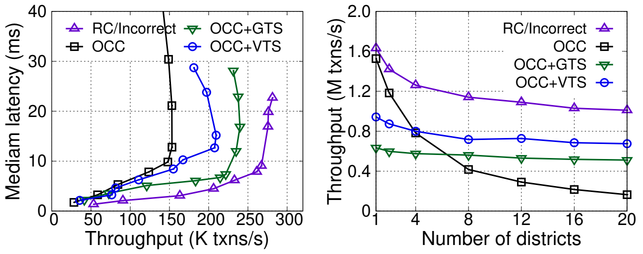


Fig. 1: *Performance of (a) `TPC-E` and (b) `TPC-C` on a local 16-node cluster using different CC protocols and TS schemes (see §5 for details). GTS and VTS stand for using OCC protocol, while the read-only transactions read the snapshots delimitated by GTS and VTS. RC/Incorrect stands for using RC protocol, which can provide optimal performance, but at the expense of correctness. One machine is dedicated for timestamp oracle, even OCC and RC have no need.*

A common approach is to leverage multi-version concurrency control (MVCC) [13, 65] for transactional systems, which has been widely adopted by nearly every commercial database like PostgreSQL [3], Oracle [4], MySQL/InnoDB [2], Hekaton [21], and SAP HANA [50]. MVCC simultaneously maintains multiple database snapshots by using timestamps to delimitate them. Thus, readers may read tuples from a stale snapshot while writers can write the tuples concurrently. It essentially unleashes the parallelism between concurrent readers and writers.

While MVCC extracts more concurrency for transactions (especially for read-only transactions), it does not necessarily approach optimal performance and/or scalability improvement (see Fig. 1), due to the overhead of maintaining timestamp ordering at scale (§2.3). More specifically, a centralized sequencer (timestamp oracle) is usually used to provide snapshot timestamp to transactions, which reflects a total order among transactions (i.e., global timestamp (GTS)). However, such a mechanism not only adds more communications but also causes overly-constrained concurrency control for read-write transactions, leading to performance degradation and scalability bottlenecks [14]. Vector timestamp (VTS), which leverages a clock per worker or machine, only mitigates the scalability bottleneck of centralized timestamp schemes but causes more network traffic, which grows lin-

[1] We evaluate different timestamp schemes on DrTM+R [18]. RC cannot provide correct results as it completely disregards the conflicts between read-write and read-only transactions. Detailed setup can be found in §5.

early with the increase of workers or machines in the system. Although recent work ameliorates the performance of centralized and/or vectorized timestamp schemes (e.g., batching requests [46, 27], timestamp compression [70], and dedicated fetch thread [70]), the fundamental performance and scalability bottlenecks remain.

In this paper, we propose a new timestamp scheme, namely, *decentralized scalar timestamp* (DST), which enables MVCC without a centralized sequencer or vector timestamps. DST is motivated by a key observation: *transaction ordering provided by existing CC protocols already implies serializable ordering among transactions, which can be reused to maintain timestamp ordering in a lightweight and scalable way.* This is because any pair of conflicting transactions must have conflicting accesses to a particular tuple. Thus, the *later* transaction should see the timestamp of the *former* transaction from the conflicting tuple and have a *larger* timestamp.

DST piggybacks on CC protocols to derive a scalable timestamp, in contrast to providing a separate timestamp scheme. Specifically, DST starts with a scalar timestamp for each transaction from a local clock and dynamically refines the tentative timestamp through transaction execution with the largest one from tuples in the read/write set. Upon commit, a transaction will also install the refined timestamp to the read/write set so that any transactions serialized after this transaction will have a larger timestamp.

One key challenge is how to derive a *consistent* yet *fresh* snapshot. DST leverages a decentralized design for read-only transactions, which introduces a *hybrid* scalar timestamp to provide snapshot reads with *bounded staleness*. Specifically, the read-only transaction can read fresh tuples whose timestamp is within two times the maximum physical clock drift under loosely synchronized clocks.[2] The fresh and consistent snapshot is obtained by attempting to read tuples using the latest hybrid timestamp while detecting and reordering any concurrent conflicting read-write transactions. In the hybrid timestamp, the physical part (a loosely synchronized clock) ensures the read-only transaction can read a fresh snapshot, and the logical part (a monotonically increasing counter) avoids possible overflow of the physical part.

To demonstrate the effectiveness and generality of DST, we have implemented DST on three representative transactional systems with different CC protocols, namely DrTM+R [18] (OCC), MySQL cluster [1] (2PL), and ROCOCO [43]. We also implemented two centralized timestamp schemes (GTS and VTS) on DrTM+R by following the state-of-the-art [46, 70]. The experimental results on three clusters show that DST can achieve more than 95% of optimal performance (using RC protocol) without compromising correctness. With DST, DrTM+R achieves up to 1.8X and 6.1X performance improvements for `TPC-E` and `TPC-C`.

[2]The clock drift (aka clock skew) can be obtained using a network time protocol like the precision time protocol (PTP), which only affects the freshness of reads in DST rather than correctness.

A comparison with other timestamp schemes shows DST is up to 1.7X and 6.3X faster than best of them for `TPC-E` and `TPC-C`, respectively. Further, DST also leads up to 1.9X and 2.1X speedup on `TPC-C` for MySQL cluster and ROCOCO.

DST shares some similarities with decentralized timestamps proposed in prior work [68, 37], which optimize a specific CC protocol for multi-core databases. For instance, TicToc [68] uses a data-driven timestamp scheme to reduce transaction aborts for OCC. Differently, DST is a general timestamp scheme for various CC protocols (§4.2) and can piggyback on each one efficiently in a distributed setting.

In summary, the contributions of this paper are:

- A *decentralized scalar* timestamp scheme called DST for MVCC that enables efficient read-only transactions with little impact on read-write transactions (§3.1 and §3.2), as well as an intuitive proof of correctness (§3.3).
- A *consistent* yet *fresh* snapshot-read approach based on a hybrid timestamp that provides bounded staleness (§3.4).
- To demonstrate the generality, DST is integrated into three representative transactional systems with different CC protocols, including OCC, 2PL, and ROCOCO (§4).
- A set of evaluations on three clusters with both microbenchmarks and applications (e.g., `TPC-E`, `TPC-C`, and `SmallBank`) confirms the performance gains of DST (§5).

The source code of three transactional systems with DST, including all benchmarks and experimental results, are available at `https://github.com/SJTU-IPADS/dst`.

2 Background and Motivation

2.1 Target Systems

DST is designed for general distributed transactions over database data partitioned to multiple storage nodes. The client's transaction request is handled by a coordinator, which interacts with storage nodes for executing the transaction. During the transaction's execution, the coordinator may send read/write requests to read/write data from the storage nodes; or send transactional requests (e.g., lock or unlock) according to the database's concurrency control protocol. It batches requests (e.g., write and unlock) to avoid extra network roundtrip.

Our goal is to support serializable read-only transaction that never aborts, and does not interfere with read-write transaction. Further, it is desirable to execute reads in the read-only transaction in one-roundtrip, i.e., the coordinator can retrieve a consistent view of the data from the storage nodes in one request.

2.2 MVCC and Timestamps

A common approach to support serializable read-only transaction without interfering with read-write transaction is through multi-version concurrency control (MVCC). There are two major design considerations for an efficient MVCC system compared with single-version mechanisms [65]. The first is *how to cheaply allocate a globally-ordered version*

2PL with global timestamp (GTS)

```
At Oracle:                          ▸ timestamp server
+ GlobalTS                  ▸ monotonic global timestamp
+ StableGTS                  ▸ snapshot global timestamp
+ Queue                       ▸ pending global timestamp

  INSTALL(gts):
+1  add gts to Queue

  STABILIZE( ):                    ▸ run asynchronously
+1  for each gts in Queue do
+2    if gts is ready then
+3      dequeue gts and gts → StableGTS

At Worker_i:                  ▸ i denotes the worker number
  WRITE(tx, id, data)
 1  acquire lock
 2  add ⟨id, data⟩ to tx.wset

  READ(tx, id)
 1  acquire lock and get latest ⟨data⟩
 2  add ⟨id, data⟩ to tx.rset
 3  return ⟨data⟩

  COMMIT(tx)
+1  tx.TS ← Oracle.GlobalTS        ▸ network round trip
 2  for each w in tx.wset do
:3    update ⟨w.data, tx.TS⟩ and release lock
 3  for each r in tx.rset do
 4    release lock
+5  Oracle.INSTALL(tx.TS)          ▸ network round trip

  ROTX(tx)                              ▸ snapshot read
+1  tx.TS ← Oracle.StableGTS
+2  for each r in tx.rset do
+3    get ⟨r.data⟩ up to tx.TS
```

Fig. 2: *Using GTS (i.e., blue code lines) to enable consistent snapshots for read-only transactions with 2PL.* +N *and* :N *denote new and modified lines of code respectively.*

for updating tuples transactionally. Deciding the version installed with tuples should have minimal impacts on read-write transactions. The second is *how to efficiently allocate a freshly-stable version for reading tuples consistently*. Read-only transactions should have access to consistent snapshots with low latency and high freshness. MVCC schemes typically adopt the concept of *timestamps* for tuple versions. However, it is non-trivial to design a general timestamp scheme that supports *efficient* snapshot reads while incurring *minimal overhead* for broad CC protocols.

To motivate the design of DST, we start by briefly reviewing how existing timestamp schemes are applied to two-phase locking (2PL) for MVCC and snapshot reads [45, 65].

Global timestamp (GTS). This approach leverages a timestamp service, namely *timestamp oracle*, to manage globally ordered timestamps [46, 15, 21]. It provides two functions for MVCC systems, as shown in Fig. 2. First, the read-write transaction contacts the oracle for a commit timestamp (GlobalTS) at the commit phase. Upon a successful commit, this transaction creates a new version denoted by the commit timestamp for each tuple in the write set (*line:3* of COMMIT) and sends back the committed timestamp to the oracle (*line:5*). Second, the read-only transaction contacts the ora-

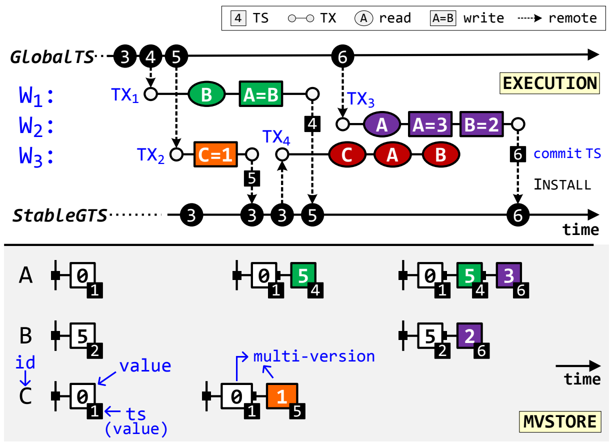


Fig. 3: *A sample case of using GTS, where four transactions (TX_1-TX_4) operate on three tuples (A, B, and C).*

cle for a read timestamp (StableGTS) and retrieves tuples in the read set with versions no larger than the read timestamp (*line:2-3* of ROTX).

Given the specification of extensions to 2PL with GTS in Fig. 2, we analyze the transaction behavior in the case shown in Fig. 3 to explain the design of GTS. There are four transactions (TX_1–TX_4), which operate on three tuples (A, B, and C). Note that non-conflicting transactions TX_1 (green) and TX_2 (orange) are both forced to acquire GlobalTS according to the specification. This operation is necessary to maintain the global timestamp ordering, yet results in overly-constrained concurrency control and an extra network round trip compared to the vanilla 2PL.

The necessity of the oracle to maintain StableGTS can be revealed with the conflict between the timestamp order and the commit order concerning TX_1 and TX_2. In this case, TX_2 acquires a larger GlobalTS but commits before TX_1. When read-only transaction TX_4 (red) starts, it cannot simply use the latest committed timestamp (GlobalTS=5) for snapshot reads. The snapshot would be inconsistent if the read-only transaction observes TX_2 before TX_1 commits. Thus, transactions must install commit timestamps so that the oracle can determine the read timestamp (StableGTS=3) for TX_4.

Vector timestamp (VTS). To reduce the overhead of acquiring GlobalTS in the critical path of read-write transactions, VTS replaces the global timestamp counter with a vector of local timestamps. The vector contains a slot for each worker, which records the per-worker timestamp. In each worker, a local counter (LocalTS) is used to assign the commit timestamp for transactions, hence reducing one network round trip compared to GTS. However, the oracle is retained in VTS to maintain the StableVTS with similar reasons as GTS. Fig. 4 shows the specification of extensions to 2PL with VTS.

Fig. 5 presents a concrete case of using VTS. Each worker maintains its local counter (W_1:3, W_2:2, and W_3:5). The version of a tuple is represented as $\langle i : ts \rangle$, where i is the worker ID, and ts is the commit timestamp of the transaction that writes the tuple. The initial StableVTS is $(3, 2, 5)$, which means that tuples with versions less than $\langle 1 : 3 \rangle$, $\langle 2 : 2 \rangle$, and

```
2PL with vector timestamp (VTS)

At Oracle:                              ▸ timestamp server
+ StableVTS                      ▸ snapshot global timestamp
+ Queues                          ▸ pending global timestamp

  INSTALL(⟨i:ts⟩, deps)
+1  add ⟨⟨i:ts⟩, deps⟩ to a Queues[i]

  STABILIZE( )                       ▸ run asynchronously
+1  for each queue in Queues do
+2    for each ⟨⟨i:ts⟩, deps⟩ in queue do
+3      if deps is ready then        ▸ stability protocol
+4        dequeue ⟨⟨i:ts⟩, deps⟩
+5        ⟨i:ts⟩ → StableVTS

At Worker_i:                   ▸ i denotes the worker number
+ LocalTS                         ▸ monotonic local timestamp

  WRITE(tx, id, data)
:1  acquire lock and get latest ⟨i:ts⟩
 2  add ⟨id, data⟩ to tx.wset
+3  add ⟨i:ts⟩ to tx.deps

  READ(tx, id)
:1  acquire lock and get latest ⟨data, ⟨i:ts⟩⟩
 2  add ⟨id, data⟩ to tx.rset
+3  add ⟨i:ts⟩ to tx.deps
 4  return ⟨data⟩

  COMMIT(tx)
+1  tx.TS ← LocalTS
 2  for each w in tx.wset do
:3    update ⟨w.data, ⟨i:tx.TS⟩⟩ and release lock
 4  for each r in tx.rset do
 5    release lock
+6  Oracle.INSTALL(⟨i:tx.TS⟩, tx.deps)  ▸ NT round trip

  ROTX(tx)                                ▸ snapshot read
+1  tx.TS ← Oracle.StableVTS
+2  for each r in tx.rset do
+3    get ⟨r.data⟩ up to tx.TS
```

Fig. 4: *Using VTS (i.e., blue code lines) to enable consistent snapshots for read-only transactions with 2PL. +N and :N denote new and modified lines of code respectively.*

$\langle 3:5 \rangle$ can be consistently read by read-only transactions.

Maintaining the stable timestamp (StableVTS) becomes more complex in VTS because the per-worker timestamps are not directly comparable [70, 8]. To convey the ordering of transactions to the oracle for deciding StableVTS, workers collect observed timestamps of accessed tuples from other workers (e.g., $\langle 1:2 \rangle$ of C for TX_2). Note that when read-write transactions (TX_1, TX_2, and TX_3) commit, they must send all observed timestamps (*deps*) to the oracle (INSTALL in Fig. 4). Moreover, read-only transactions (TX_4) must request the whole vector timestamp (StableVTS) from the oracle to start a snapshot read.

2.3 Analysis of Network Overhead

We present an in-depth analysis of centralized timestamp schemes[3] and attribute performance overhead and scalability bottleneck to three main aspects:

[3]For brevity, we avoid prior sophisticated optimizations (incl. batching requests [46, 27], timestamp compression and dedicated fetch thread [70]) for timestamps in here, but enable all of them in the evaluation (§5).

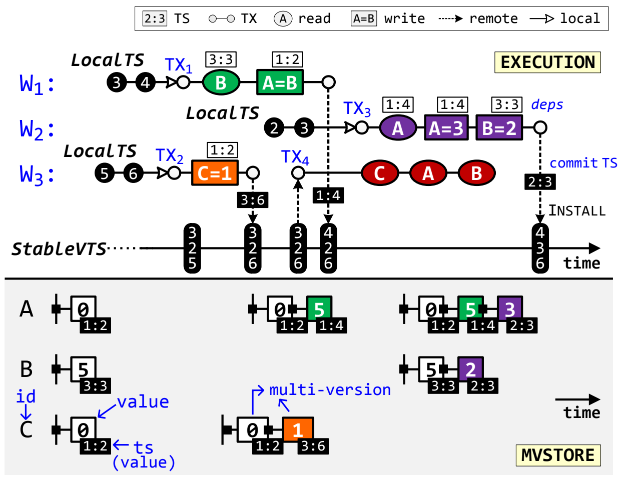


Fig. 5: *A sample case of using VTS, where four transactions (TX_1-TX_4) operate on three tuples (A, B, and C).*

Non-scalable timestamp oracle. Prior work [46, 61, 11, 59, 67, 37] has shown that a centralized timestamp oracle will become the scalability bottleneck of MVCC systems. The throughput of schemes using a shared counter with atomic operations (GTS) [31, 21, 25] is limited to less than 10 M ops/s (GlobalCNT in Fig. 6(a)). The throughput will further decrease due to maintaining the stable timestamp for read-only transactions (+StableTS). VTS mitigates the scalability issue by using a local counter for read-write transactions. Besides, prior work [70] avoids the mechanism for the stable timestamp (reaching close to 40 M ops/s) at the expense of increasing transaction aborts. However, the network will first become the bottleneck for both GTS and VTS (Network). Consequently, the throughput of timestamp oracle (TSOracle) can only reach 1.26 M and 2.39 M ops/s for GTS and VTS respectively, which may be enough for TPC-E (281 K txns/s) but far not enough for TPC-C (1.64 M txns/s) and SmallBank (80 M txns/s) even only scaling out to 16 machines.

Using fast networks can boost the throughput of timestamp oracle, while the performance of transactional systems will also increase much [23, 64, 26, 70], and CPU may first become the bottleneck [59]. Moreover, batching requests [46, 27] or dedicated fetch thread [70] can alleviate the timestamp-related load on the network[4], while these techniques also amplify the staleness of the data retrieved by read-only transactions, and increase the abort rate and the end-to-end latency of read-write transactions (see §5.1).

Costly timestamp allocation. A centralized timestamp scheme will inevitably cause extra network communication overhead for each read-write transaction. GTS demands two network round trips, one for obtaining the commit timestamp and one for installing it. VTS uses per-worker local counters to assign the commit timestamp, but still demands one network round trip to install the timestamp. Given that most transactions operate on tuples in local partitions [56, 54, 55], especially for read-write transactions, additional network round trips will notably lengthen the critical section of trans-

[4]We enabled these optimizations for GTS and VTS in our evaluation (§5).

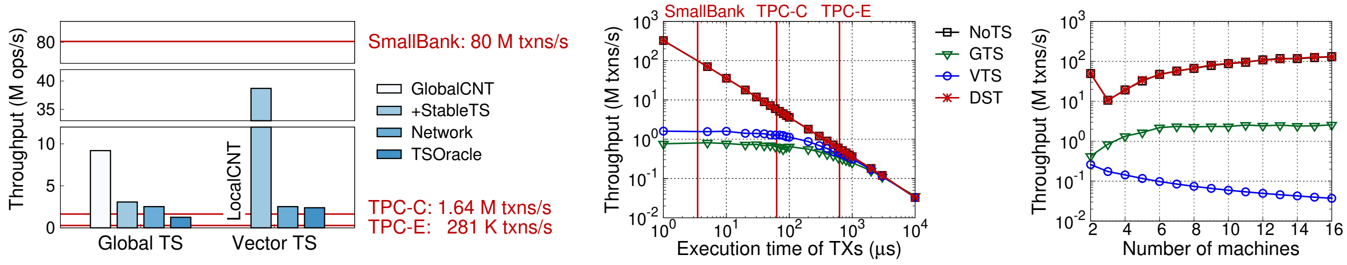


Fig. 6: *(a) Analysis of peak performance and bottleneck of timestamp oracle for GTS and VTS using a 24core machine with 10GbE. (b) The performance of `read-write` transactions with the increase of execution time for different timestamp schemes. The weighted average median latency of read-write transactions in `TPC-E`, `TPC-C` and `SmallBank` are labeled by red lines. (c) The performance of `read-only` transactions with the increase of machines for different timestamp schemes. All experiments are conducted on a local 16-node cluster with 10GbE network (§5). One machine is dedicated for timestamp oracle, even NoTS has no need. Each machine spawns 24 server workers.*

actions and further increase the chance of conflicts, causing extra transaction aborts or blocking time. Thus, it is non-trivial to hide the network round trips without sacrificing the latency of transactions (e.g., batching requests [46, 27]).

The overhead of timestamp allocation highly depends on the execution time of transactions. Hence, we implement a microbenchmark only consisting of read-write transactions, which do not access any tuples and just spin in a loop for a given time. As shown in Fig. 6(b), the overhead of VTS is moderate (from 10% to 30%) compared to not using timestamp schemes (NoTS), when the execution time is close to that of read-write transactions in `TPC-E` (from 1,400μs to 470μs). The throughput will significantly drop more than 80% when transactions execute in about 50μs, which is similar to that of read-write transactions in `TPC-C`. Further, GTS can only achieve half of VTS throughput, since it demands one more round trip to obtain the commit timestamp.

Large traffic size. VTS mitigates the timestamp overhead by using per-worker local counters as the commit timestamp for read-write transactions. However, a critical downside is that a whole vector of per-worker timestamps must be obtained as the read timestamp first, and then be transferred to every tuple for performing consistent snapshot reads. In contrast to the scalar timestamp (e.g., GTS), this overhead grows linearly with the increase of workers or machines in the system. For most transactional workloads [54, 55, 56, 57], the size of the vector timestamp can become orders of magnitude larger than the tuple size, even in a moderate-sized cluster. Using the per-machine counter in VTS (i.e., all workers on one machine share one timestamp slot) can reduce traffic size [8]. However, these workers have to share a local counter by using *atomic* operations (e.g., CAS), which will incur additional overhead on read-write transactions [70].

To demonstrate the impact of traffic size, we implement a microbenchmark only consisting of read-only transactions, which read ten 8-byte tuples with 90% of which being local accesses. In Fig. 6(c), the performance collapse of VTS is due to the increase of timestamp vector obtained from the oracle and transferred to remote tuples. Note that GTS is still one order of magnitude slower due to extra one round-trip to fetch the read timestamp (even scalar), compared to NoTS.

3 Decentralized Scalar Timestamp (DST)

Managing globally ordered timestamps in a centralized service inevitably results in the problem of maintaining the consistency between timestamp ordering and transaction ordering. More importantly, without a holistic decentralized design, the timestamp scheme cannot achieve good scalability. This observation can be backed by the aforementioned performance bottlenecks due to acquiring commit/read timestamps and installing committed timestamps. Such operations add significant overhead to the execution of CC protocols.

To fundamentally overcome the above drawbacks of traditional timestamp schemes, we propose DST, a *decentralized scalar timestamp* that facilitates the multi-version concurrency control (MVCC) implementation for broad CC protocols with efficient snapshot read support and minimal overhead. The intuition behind our design is that *the timestamp scheme can piggyback on concurrency control protocols to maintain the timestamp ordering with low cost and no new scalability bottleneck to read-write transactions*.

In this section, we first use two-phase locking (2PL) as an example to explain the basic protocol of DST for read-write and read-only transactions (§3.1 and §3.2). We then prove the serializability of read-only transactions with DST (§3.3) and introduce a hybrid scalar timestamp to provide snapshot reads with bounded staleness (§3.4). Finally, we discuss the impact of DST on the fault-tolerance scheme (§3.5).

3.1 Timestamps in Read-write Transaction

DST is a fully decentralized timestamp without a centralized sequencer (timestamp oracle) to provide total order timestamps for read-write transactions and stable timestamps for read-only transactions. Therefore, DST must ensure that the derived timestamps for read-write transactions always match the transaction ordering.

The CC protocol is used to ensure the serializable transaction ordering and provide the following three properties, where Transaction A (TX_A) commits before Transaction B (TX_B), and both of them access a conflicting tuple O.

PROPERTY 1: Write-Write. TX_B's write ($W_B(O)$) should overwrite TX_A's write ($W_A(O)$) or generate a newer version.

```
Read-write Transaction: 2PL with DST
At Worker_i:              ▸ i denotes the worker number
+  LocalTS                 ▸ monotonic local timestamp

   START(x)
+1    x.TS ← LocalTS

   WRITE(x, id, data)
:1    acquire lock and get ts
 2    add ⟨id, data⟩ to x.wset
+3    x.TS ← max(x.TS, ts+1)

   READ(x, id)
:1    acquire lock and get latest ⟨data, ts⟩
 2    add ⟨id, data⟩ to x.rset
+3    x.TS ← max(x.TS, ts+1)
 4    return data

   COMMIT(x)
 1    for each w in x.wset do
:2      update ⟨w.data, x.TS⟩ and release lock
 3    for each r in x.rset do
:4      update ⟨x.TS⟩ and release lock
+5    LocalTS ← max(LocalTS, x.TS)
```
Fig. 7: *Specification of* `read-write` *transaction for 2PL with DST. +N and* `:N` *denote new and modified lines of code respectively.*

PROPERTY 2: Write-Read. TX_B's read ($R_B(O)$) should retrieve TX_A's write ($W_A(O)$).

PROPERTY 3: Read-Write. TX_A's read ($R_A(O)$) should not retrieve TX_B's write ($W_B(O)$).

To match the transaction ordering, DST should ensure TX_B's commit timestamp (TS_B) is larger than TX_A's commit timestamp (TS_A) under the above case. The general idea is to piggyback over the CC protocol to derive a commit timestamp from conflicting tuples. Fig. 7 presents how DST is integrated with two-phase locking (2PL), and Fig. 8 also illustrates the execution of sample transactions with DST. DST leverages conflicting tuples and above three properties to transmit commit timestamps between dependent transactions. The additional codes for DST in WRITE, READ, and COMMIT (see Fig. 7) are commented on corresponding operations in the following explanations.

Write-Write property. Transaction TX_A installs value (V_A) with commit timestamp (TS_A) into the tuple O.

$$\begin{aligned} \langle V_A, TS_A\rangle &\rightarrow O \\ TS_A &\rightarrow O.ts && \triangleright\ \textit{line:2} \text{ of } \textsc{Commit} \end{aligned}$$

Transaction TX_B reads the timestamp of tuple O (O.ts) and installs new value (V_B) with a *larger* commit timestamp (TS_B) into the tuple O.

$$\begin{aligned} O.ts &\rightarrow ts && \triangleright\ \textit{line:1} \text{ of } \textsc{Write} \\ max(ts{+}1, TS_B) &\rightarrow TS_B && \triangleright\ \textit{line:3} \text{ of } \textsc{Write} \\ \langle V_B, TS_B\rangle &\rightarrow O \\ TS_B &\rightarrow O.ts && \triangleright\ \textit{line:2} \text{ of } \textsc{Commit} \end{aligned}$$

In Fig. 8, TX_1 (green) commits before TX_3 (purple), and both of them write tuple A. Therefore, the commit timestamp of TX_3 should be larger than that of TX_1. Using DST, TX_1 installs its value (5) with its commit timestamp (TS_1=4) into tuple A. After that, TX_3 should derive a larger timestamp (TS_3=5) from the timestamp of tuple A (A.ts=4) and use it to install new value (3) into tuple A. Note that the write operations will update both the tuple's timestamp and the value's timestamp (as a tuple may have multiple values with different versions).

Write-Read property. Transaction TX_A installs value V_A with its commit timestamp TS_A into tuple O.

$$\begin{aligned} \langle V_A, TS_A\rangle &\rightarrow O \\ TS_A &\rightarrow O.ts && \triangleright\ \textit{line:2} \text{ of } \textsc{Commit} \end{aligned}$$

Transaction TX_B reads value V_A of tuple O with timestamp O.ts and installs a larger commit timestamp TS_B.

$$\begin{aligned} O &\rightarrow \langle V_A, ts\rangle && \triangleright\ \textit{line:1} \text{ of } \textsc{Read} \\ max(ts{+}1, TS_B) &\rightarrow TS_B && \triangleright\ \textit{line:3} \text{ of } \textsc{Read} \\ TS_B &\rightarrow O.ts && \triangleright\ \textit{line:4} \text{ of } \textsc{Commit} \end{aligned}$$

In Fig. 8, TX_1 (green) commits before TX_3 (purple), and TX_1 writes tuple A before TX_3 reads it. Therefore, the commit timestamp of TX_3 should be larger than that of TX_1. Using DST, TX_1 installs its value 5 with its commit timestamp (TS_1=4) into tuple A. After that, TX_3 reads the timestamp of tuple A (A.ts=4) and derives a larger timestamp (TS_3=5).

Read-Write property. Transaction TX_A installs commit timestamp TS_A into tuple O since it has read the value of tuple O.

$$TS_A \rightarrow O.ts \qquad \triangleright\ \textit{line:4} \text{ of } \textsc{Commit}$$

TX_B reads timestamp of tuple O (O.ts) and installs new value V_B with a larger timestamp TS_B into tuple O (O.ts).

$$\begin{aligned} O.ts &\rightarrow ts && \triangleright\ \textit{line:1} \text{ of } \textsc{Write} \\ max(ts{+}1, TS_B) &\rightarrow TS_B && \triangleright\ \textit{line:3} \text{ of } \textsc{Write} \\ \langle V_B, TS_B\rangle &\rightarrow O \\ TS_B &\rightarrow O.ts && \triangleright\ \textit{line:2} \text{ of } \textsc{Commit} \end{aligned}$$

In Fig. 8, TX_1 (green) commits before TX_3 (purple), and TX_1 reads tuple B before TX_3 writes it. Therefore, the commit timestamp of TX_3 should be larger than that of TX_1. Using DST, TX_1 reads an old value (5) of tuple B and installs its commit timestamp (TS_1=4) into tuple B. After that, TX_3 will derive a larger timestamp (TS_3=5) and use it to install new value (2) into tuple B.

3.2 Timestamps in Read-only Transaction

DST ensures that the order of derived commit timestamps for read-write transactions always matches the transaction ordering. Therefore, read-only transactions can directly pick any timestamp (TS_{RO}) to read a consistent snapshot by comparing its read timestamp with the timestamps of tuples.

Since the (snapshot) read-only transaction does not follow the CC protocol (e.g., lock/unlock tuples before/after reading values), the read-only transaction may read a part of updates of a concurrent read-write transaction. For example, in Fig. 8, the read-only transaction TX_4 (red) and the read-write transaction TX_3 (purple) are concurrently executed. If TX_3 commits between the read operations to tuple A and tuple B in TX_4, and then TX_4 will read an old version of tuple A (5) and a new version of tuple B (2).

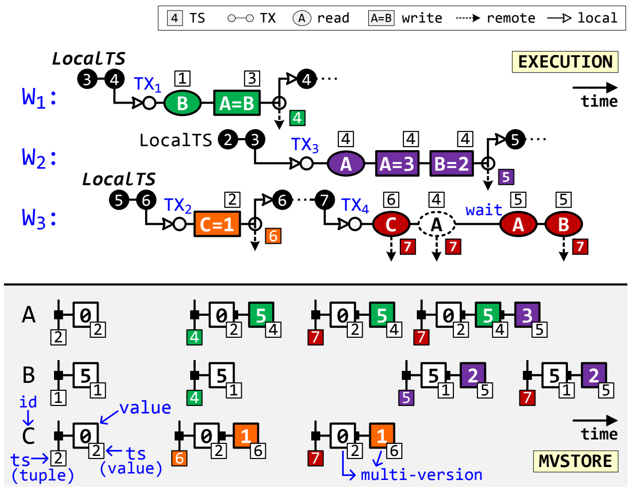


Fig. 8: *A sample case of using DST, where four transactions (TX_1-TX_4) operate on three tuples (A, B, and C).*

To ensure the serializability of read-only transactions, DST asks the read-only transaction to claim its operations actively before reading the tuple. It first installs its read timestamp (TS_{RO}) into the tuple and waits until the conflicting read-write transaction commits (e.g., the tuple is not locked), if the timestamp of the tuple is not larger than the read timestamp (DEP_READ in Fig. 9). Note that the read-only transaction will only wait for at most one conflicting read-write transaction because if the concurrent read-write transaction starts after the claim, it will definitely see the read timestamp through accessing the tuple and derive a larger commit timestamp. Consequently, the read-only transaction will skip all of the updates from this transaction. If the concurrent read-write transaction starts before the claim, it will hold the lock of the tuple. The read-only transaction will wait until the read-write transaction commits. No matter the commit timestamp is larger or smaller than the read timestamp, a read-only transaction can always read a consistent snapshot by ignoring or reading all of the updates from conflicting transactions. Note that CC protocols ensure the atomicity of read-write transaction's updates.

As shown in Fig. 8, the read-only transaction TX_4 will install its read timestamp (TS_4=7) into tuples with smaller tuple timestamps (*line:1* of DEP_READ in Fig. 9). For unlocked tuple C, TX_4 will directly read the value up to the timestamp (1). For locked tuple A and B, TX_4 will wait until the concurrent read-write transaction TX_3 commits. In this example, since TX_3 does not see the read timestamp of TX_4, the commit timestamp of TX_3 is still smaller than the read timestamp of TX_4 (5 vs. 7). Hence, TX_4 can read all updates from TX_3 (A=3 and B=2).

3.3 Proof of Correctness

Theorem (Serializability). *DST implements serializable read-only transactions, which always read a consistent snapshot generated by serializable read-write transactions.*

Proof sketch. The intuition of the proof is that if a read-only transaction can be serialized with read-write transactions, then it reads a consistent snapshot. We provide a proof sketch by contradiction based on this intuition: i.e., if a read-only transaction cannot be serialized with read-write transactions, then it leads to a contradiction. Before giving the proof, we need to prove following two lemmas first:

```
Read-only Transaction: 2PL with DST
At Worker_i:                    ▸ i denotes the worker number
+  LocalTS                      ▸ monotonic local timestamp

   ROTX(x)                      ▸ snapshot read
+1    x.TS ← LocalTS
+2    for each r in x.rset do
+3       DEP_READ(x, r)

   DEP_READ(x, r)
+1    if r.ts <= x.TS then
+2       r.ts ← x.TS             ▸ atomic (CAS)
+3       wait until r not Locked ▸ if conflict
+4    get ⟨r.data⟩ up to x.TS
```

Fig. 9: *Specification of* `read-only` *transaction for 2PL with DST.*

Lemma 1. *Given two dependent read-write transactions TX_1 and TX_2, if TX_2 depends on TX_1, then TX_2's timestamp (TS_2) is larger than TX_1's timestamp (TS_1).*

Proof. If TX_2 directly depends on TX_1[5], this lemma follows directly from the algorithm (see §3.1) that TX_2 always calculates TS_2 based TS_1. If TX_2 transitively depends on TX_1, in a proof by contradiction we assume TS_1 is not smaller than TS_2, then in the partial dependent graph denoted by $TX_1 \rightarrow \ldots \rightarrow TX_i \rightarrow TX_j \ldots \rightarrow TX_2$[6], there exists TX_i and TX_j that TX_j directly depends on TX_i, but its timestamp is not larger than TX_i's, which is a contradiction with the first case. □

Lemma 2. *Given a read-only transaction TX_{RO} and a read-write transaction TX_{RW}, TX_{RO} observes TX_{RW}'s update on tuple O[7], if and only if TX_{RO}'s timestamp (TS_{RO}) is not smaller than TX_{RW}'s timestamp (TS_{RW}).*

Proof. First, if TX_{RO} observes TX_{RW}'s update on O, then TS_{RO} is not smaller than TS_{RW}. Because TX_{RW} updates O with TS_{RW} and content atomically (e.g., 2PL), TX_{RO} waits for TX_{RW}'s commit. Second, if TS_{RO} is not smaller than TS_{RW}, then TX_{RO} eventually observes TX_{RW}'s update on O. Assume TX_{RO} does not observe TX_{RW}'s update, then TX_{RO} reads O before TX_{RW} commits its update. One situation is TX_{RO} reads O before TX_{RW}'s request arrives, it leads a contradiction that TX_{RO} update O's timestamp to be TS_{RO} before the read. Another situation is TX_{RO} reads O after TX_{RW} calculates TS_{RW}, but before committing its update. This leads to the contradiction that TX_{RO} always waits for the concurrent TX_{RW} to commit (e.g., 2PL). □

Proof of the Theorem. TX_1 updates A, TX_2 updates B, and TX_2 depends on TX_1. Assume read-only transaction TX_{RO} only observes TX_2's update on B, but does not observe TX_1's

[5] TX_2 is conflicting with TX_1, and TX_2 accesses the conflicting tuples immediately after TX_1.

[6] The symbol $\rightarrow$ indicates the happen-before relation.

[7] It means TX_{RO}'s read on O happens after TX_{RW}'s update.

update on A (i.e., inconsistent reads).[8] From LEMMA 2, we have $\mathsf{TS_{RO}}$ is not smaller than $\mathsf{TS_2}$, while $\mathsf{TS_1}$ is larger than $\mathsf{TS_{RO}}$. Therefore, we have $\mathsf{TS_1}$ is larger than $\mathsf{TS_2}$, which is contradictory to LEMMA 1. □

3.4 Hybrid Timestamp and Bounded Staleness

Hybrid timestamp. The commit timestamp of a read-write transaction is derived from the timestamps of tuples in its read/write set, and the read timestamp of a read-only transaction can be any timestamp in the past, at present, or even in the future. Therefore, the local timestamp (LocalTS) is not essential for the correctness of DST. However, the read-only transaction may suffer from either staleness or performance issues if using an improper read timestamp. If the read timestamp is too small (past), the read-only transaction may read an excessively stale snapshot. If the read timestamp is too large (future), the read-only transaction will frequently install its read timestamp into tuples and wait until conflicting read-write transactions commit (DEP_READ in Fig. 9).

DST adopts a combination of *physical* clock and *logic* counter as a hybrid timestamp. The 64-bit timestamp consists of the 48-bit physical part (high-order bits) and the 16-bit logic part (low-order bits). DST uses a loosely synchronized clock as the physical part and uses a monotonically increasing counter as the logical part. At the beginning of the transaction, it will acquire a local hybrid timestamp composed of the current physical clock and zero-initialized logic counter (START in Fig. 7 and *line:1* of ROTX in Fig. 9). The logical part of the hybrid timestamp is used to avoid possible overflow of the physical part since the timestamp will be incremented when calculating the maximum timestamp (e.g., *line:3* of WRITE in Fig. 7). On the other hand, the physical part of the hybrid timestamp is used to ensure the read-only transaction can read a fresh snapshot.

Bounded staleness. Based on the hybrid timestamp, DST can provide snapshot reads with bounded staleness.

THEOREM (BOUNDED STALENESS). *The updates of read-write transactions can be observed in at most Δ, where Δ is the maximal duration any machine needs to make its local clock increased by $2 \times \varepsilon$, and ε is the maximal clock drift between any two machines in the cluster.*

PROOF SKETCH. First, we prove the following two lemmas:

LEMMA 1. *Given a read-write transaction $\mathsf{TX_{RW}}$, its commit timestamp ($\mathsf{TS_{RW}}$) is not larger than $t_m + \varepsilon$, where t_m is the local machine time on $\mathsf{TX_{RW}}$ commits.*

PROOF. If $\mathsf{TS_{RW}}$ is larger than $t_m + \varepsilon$, then there is a $\mathsf{TX_i}$ which accesses a tuple before $\mathsf{TX_{RW}}$, and $\mathsf{TS_i}$ is larger than $t_m+\varepsilon$. As the timestamp is calculated from its local machine time or the tuples it accessed, we can inductively find a transaction $\mathsf{TX_j}$ whose timestamp is larger than $t_m + \varepsilon$, and it is calculated from its local machine time. It is a contradiction to the maximal clock drift between any two nodes is ε. □

[8]The proof is also correct for $\mathsf{TX_1}$ and $\mathsf{TX_2}$ are the same transaction.

LEMMA 2. *For any read-only transaction $\mathsf{TX_{RO}}$ starts after $\mathsf{TX_{RW}}$ commits, its read timestamp $\mathsf{TS_{RO}}$ is larger than $t_m - \varepsilon$.*

PROOF. This follows that $\mathsf{TX_{RO}}$ calculates its timestamp based on local machine time and the clock drift between any two nodes cannot be larger than ε. □

PROOF OF THE THEOREM. With LEMMA 1 and 2, we can have a fact that, if $\mathsf{TX_{RO}}$ starts after $\mathsf{TX_{RW}}$, then $\mathsf{TS_{RO}}$ cannot be smaller than $\mathsf{TS_{RW}}-2 \times \varepsilon$. Since any machine is able to increase its local machine time by $2 \times \varepsilon$ in Δ, we can conclude that the updates of $\mathsf{TX_{RW}}$ will be visible in the duration of Δ. □

3.5 Failure and Recovery

The CC protocol should provide a proper fault-tolerance scheme to recover the transactional system from various failures. For example, the primary-backup replication [30] is widely used to provide high availability in prior work [23, 18, 26]. The fault-tolerance schemes can usually work with various timestamp schemes by replicating tuples together with the commit timestamps of read-write transactions. However, the fully decentralized design of DST has two sides. The advantage of this approach is to avoid handling the failure of centralized timestamp oracle, which may cause a stop-the-world recovery [70]. The disadvantage is the potential cost to maintain the consistency of decentralized timestamps before and after some failure occurs.

An obvious, but costly solution is to replicate the read timestamps of read-only transactions together with tuples, as the commit timestamps of read-write transactions. Because the missing read timestamp may cause a new conflicting read-write transaction to use a smaller commit timestamp to write tuples; the read-only transaction may read some tuples with an old version and other tuples with a new version before and after the failure occurs, respectively.

To avoid replicating or persisting read timestamps, DST provides two alternative solutions that can be selected according to the behavior of workloads or the CC protocol associated. More specifically, after recovery, DST can selectively abort and re-execute either the remaining read-only transactions that read tuples on crashed machines or the remaining read-write transactions that write tuples on crashed machines. Consequently, there is no additional overhead and modification associated with the normal execution of transactions, regardless of which approach is selected.

4 Generality of DST

DST is a general timestamp scheme to enable efficient read-only transactions with little impact on read-write transactions. Hence, it is easy to integrate DST with various CC protocols, and DST can also cooperate with many optimizations [37, 43] on CC protocols. In this section, we first lay out a general guideline for piggybacking DST on various CC protocols, and then demonstrate the efficacy of this guideline by applying it to three representative transactional systems

(DrTM+R, MySQL cluster, and ROCOCO) with different CC protocols (OCC, 2PL, and ROCOCO).

4.1 A Guideline for Integrating DST

Read-write transaction. DST should allocate a commit timestamp for the read-write transaction that is larger than any dependent transactions' timestamp. Thus, two following tasks (`RW1` and `RW2`) should piggyback on CC protocols.

1. *select a commit timestamp larger than both the current local timestamp and the timestamps of tuples in the read/write set.* (`RW1`)
2. *install the commit timestamp to tuples in the read/write set before the transaction commits.* (`RW2`)

Read-only transaction. DST should guarantee the read-only transaction can read the value of tuples up to the read timestamp. Thus, two following tasks (`RO1` and `RO2`) should piggyback on CC protocols.

1. *select an appropriate read timestamp according to the current local timestamp.* (`RO1`)
2. *ensure the tuple has an equal or larger timestamp before reading its value up to the read timestamp.* (`RO2`)

4.2 Case Study

The description below focuses on the general comments about integrating DST; we omit a few details and corner cases due to space limitations.

DrTM+R. Optimistic concurrency control (OCC) is widely adopted by modern transactional systems [21, 10, 59, 62, 22, 23, 18, 26, 63]. The read-only transaction in OCC will take two or more rounds of reads for consistent results without MVCC and timestamp schemes, due to conflicting read-write transactions. We use DrTM+R [18] to demonstrate how DST piggybacks on OCC.[9]

For the read-write transaction, we can obtain the timestamp of tuples in the read and write set when validating and locking them respectively and then derive a larger commit timestamp (`RW1`). Before committing, we should install the commit timestamp to the tuples in the read and write set (`RW2`). Note that there is no need to lock tuples in the read set since dummy timestamps from aborted transactions are benign. For the read-only transaction, all CC protocols share (almost) the same implementation (see Fig. 9). The only difference is how to wait for conflicting transactions (*line:3* of DEP_READ). For OCC, the conflicting read-write transaction will lock the tuple when installing its timestamp for updates. Therefore, similar to 2PL, the read-only transaction will confirm that the tuple is not locked before reading the value up to its read timestamp.

MySQL cluster. Two-phase locking (2PL) is another classic CC protocol used by many transactional systems [1, 19, 36]. The read-only transaction in 2PL will be blocked without MVCC and timestamp schemes, due to conflicting read-write transactions. We use MySQL cluster [1] (v7.6.8) to show the integration of 2PL and DST, mainly following the specification in Fig. 7 and 9.[10] To support the read-write lock in MySQL cluster, the transaction only needs to install timestamp into tuples in the read set atomically (i.e., compare-and-swap) and avoids overwriting a larger timestamp. Further, we leverage the lock queue mechanism in MySQL cluster to wait for conflicting transactions (*line:3* of DEP_READ in Fig. 9), which avoids spinning on the tuple.

[9]We use the version of DrTM+R [63] without HTM, which also enables coroutine and supports various networks (e.g., TCP/IP and RDMA).

Table 1: *Measurement clusters.*

Name	#	Hardware
AWS	32	r4.2xlarge (8x vCPU, 61GB DRAM, up to 10GbE)
VAL	16	2x Intel Xeon E5-2650 v4 (12 cores), 128GB DRAM, 1x Intel I350 10GbE
VLR	16	2x Intel Xeon E5-2650 v4 (12 cores), 128GB DRAM, 2x Mellanox ConnectX-4 100Gbps InfiniBand RNICs

ROCOCO. ROCOCO [43] is a research CC protocol that outperforms traditional protocols under high contention workloads by reordering conflicting read-write transactions instead of aborting them. The read-write transaction is chopped into pieces by an offline checker and uses a two-phase mechanism. The *start* phase explores a dependency graph, and then the *commit* phase executes conflicting transactions with a serializable order according to the dependency graph. The read-only transaction in ROCOCO is blocked until the completion of conflicting transactions and uses multiple rounds for reading consistent results.

To extend ROCOCO[11] with DST, the general idea is to use the dependency graph to collect timestamps of dependent tuples and derive a larger commit timestamp for the read-write transaction in the start phase (`RW1`). Then the commit timestamp can be installed to tuples in the commit phase (`RW2`). For the read-only transaction, the blocking mechanism in ROCOCO is reused to wait for conflicting transactions (*line:3* of DEP_READ in Fig. 9).

5 Evaluation

We have integrated DST with three representative transactional systems, namely DrTM+R, MySQL cluster, and ROCOCO, with different CC protocols, and also implemented two centralized timestamp schemes (GTS and VTS) by following the state-of-the-art [46, 70][12] with many carefully tuned optimizations (e.g., batching requests [46, 27], cooperative multitasking [26], timestamp compression and dedicated fetch thread [70]). These optimizations have significant performance improvements on GTS and VTS. For example, cooperative multitasking improves the peak per-machine throughput of GTS on DrTM+R by 3.04X, and timestamp compression improves VTS by 2.7X on a 16-node cluster.

Testbed and setup. To study the impact of hardware plat-

[10]Although MySQL cluster uses read committed (RC) protocol by default, it also provides serializability by using per-row 2PL.

[11]Source code: `https://github.com/shuaimu/rococo`.

[12]Different than Percolator [46], we use the stabilization process to avoid holding locks when acquiring write timestamp, since it will significantly increase transaction abort rate.

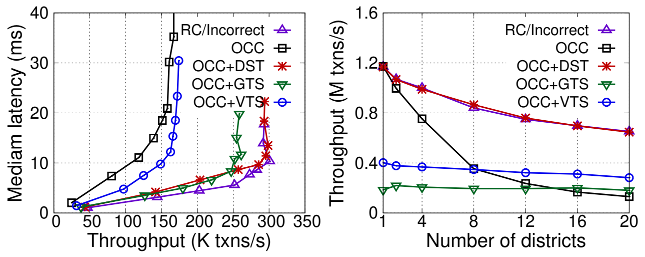


Fig. 10: *Performance of (a)* `TPC-E` *and (b)* `TPC-C` *on AWS.*

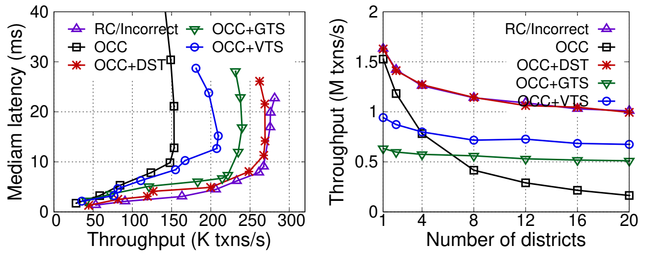


Fig. 11: *Performance of (a)* `TPC-E` *and (b)* `TPC-C` *on VAL.*

forms on DST, we use three clusters with different networks and CPU processing power (Table 1). Without explicit mention, one machine in each cluster is dedicated to the timestamp oracle, even only GTS and VTS need. Other machines serve as both database nodes and clients. We use these machines in a *symmetric* setting [23], namely each machine both executes transactions and store database data.

Benchmarks and performance overview. As the performance benefit of using MVCC and snapshot reads is sensitive to characteristics of read-only transactions in OLTP workloads, we chose three different benchmarks, namely `TPC-E`, `TPC-C`, and `SmallBank`, to show the benefits of DST comprehensively. `TPC-E` [57] presents the workload of a brokerage firm with a high proportion of read-only transactions (79% of the standard mix) and complicated operations (massive range queries and distributed accesses). DST is expected to improve the performance much compared to the vanilla CC protocols for this target workload, with *a relaxed consistency level from strict serializability to serializability*. `TPC-C` [56] simulates a warehouse-centric order processing application with a few read-only transactions (8% of the standard mix). DST is expected to show gradual improvement with the increase of execution time in read-only transactions (not affect proportion). We increase the number of districts (one district by default) accessed by the read-only `stock-level` transactions (4%). `SmallBank` [54] models a simple banking application where transactions perform very simple read and write operations (less than four) on user accounts. DST is expected not to incur perceptible overhead and show order-of-magnitude speedup compared to centralized timestamp schemes (GTS and VTS). In all benchmarks, DST should achieve close to optimal performance using RC (5%) but without compromising correctness, which can be backed by the experimental results of DST on motivating microbenchmarks (see Fig. 6).

5.1 DrTM+R

We deploy one server at each machine and co-locate clients to saturate the performance of servers as prior work [58, 60, 23, 64, 26]. Due to space limitations, we do not report the experimental results on `SmallBank`, which are as expected.

TPC-E. Fig. 10(a) shows the results of `TPC-E` on AWS. `TPC-E` has a high proportion of read-only transactions, and most of them are distributed. Compared to using snapshot reads (GTS, VTS, and DST), the vanilla OCC protocol provides *strict serializability* and requires an additional round to validate tuples in the read set. Thus, many read-only transactions will abort under heavy workloads. As a reference, RC can outperform OCC by 1.79X (yet with incorrect results), since it simply skips the validation phase. DST achieves almost the same performance as RC, as it also avoids the validation phase and never aborts read-only transactions. Differently, DST ensures the read-only transaction can read a consistent yet fresh snapshot. Moreover, compared to GTS and VTS with the same consistency level (*serializability*), DST can outperform the throughput of them by 1.16X and 1.72X, respectively. Because DST omits the communication to the timestamp oracle and avoids large traffic size due to using a fully decentralized design and scalar timestamps (see §2.3).

We further evaluate `TPC-E` on VAL. As shown in Fig. 11(a), DST can still achieve similar performance as RC and provides 1.13X and 1.29X speedup compared to GTS and VTS, respectively. VTS performs slightly better on VAL due to using a relatively smaller vector timestamp.

TPC-C. Fig. 10(b) and Fig. 11(b) show the peak throughput of `TPC-C` on AWS and VAL with the increase of districts accessed by the read-only `stock-level` transactions. Note that the default setting in `TPC-C` accesses one district (the first data point of every line). Besides, we retain all default settings, like the proportion of `stock-level` transactions (4%).

As shown in Fig. 10(b), when accessing one district, DST has a very close performance compared to RC. These results indicate that DST has little overhead to read-write transactions. In comparison to DST, GTS and VTS are 6.29X and 2.93X slower than RC, due to the significant cost for maintaining centralized and/or vectorized timestamps (see §2.3).

OCC performs well on the original TPC-C due to the limited read-only transactions in the standard-mix (8%). On the other hand, when increasing the execution time of read-only `stock-level` transactions (by accessing more districts), the performance difference between RC and OCC is more evident because OCC has more overheads for validating the read-set of the `stock-level`. DST still performs close to RC and is 4.94X faster than vanilla OCC (accessing 20 districts) with a relaxed consistency level. Finally, DST still outperforms VTS and GTS by 2.29X and 3.56X when accessing 20 districts, respectively.

In Fig. 11(b), the performance of DST is also very close to RC for `TPC-C` on VAL. On the other hand, the overhead of GTS and VTS still incurs up to 2.57X (from 1.95X) and 1.73X (from 1.47X) slowdown, compared with DST. Differ-

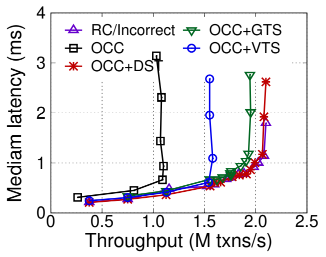


Fig. 12: *Performance of (a)* `TPC-E` *and (b)* `TPC-C` *on VLR.*

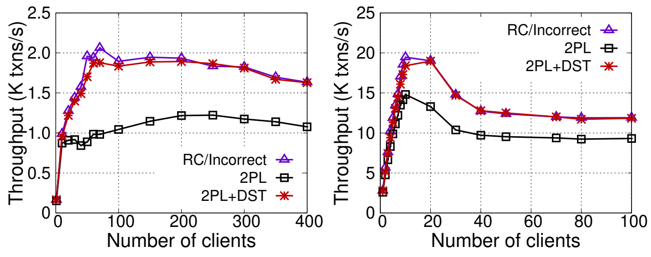


Fig. 13: *Performance of (a)* `TPC-C` *and (b)* `SmallBank` *for MySQL cluster with different CC protocols on VAL.*

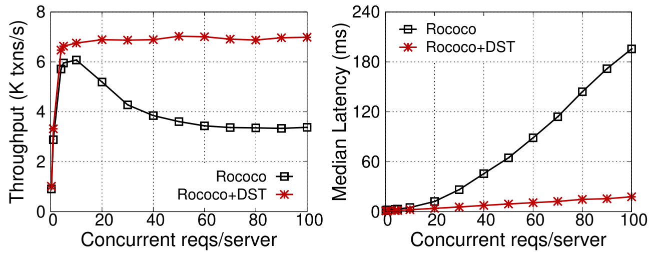


Fig. 14: *(a) Throughput of* `new-order` *transactions and (b) median latency of* `stock-level` *transactions in* `TPC-C` *mixed workload.*

ent than AWS, the lower latency of network round-trip on VAL (90μs) is beneficial for centralized timestamp schemes, but the effect is quite limited.

Using fast network (i.e., RDMA). Readers might be interested in how the performance of networks impacts the performance of timestamp schemes, especially using RDMA. DrTM+R naturally supports RDMA, and we adopt FaSST-RPC [26] to implement the timestamp oracle for GTS and VTS. By using 100Gbps RDMA, the CPU may become the bottleneck in the timestamp oracle for GTS, about 3.0 M ops/s (see §2.3). For VTS, the timestamp oracle will not limit the performance of `TPC-E` and `TPC-C` with only 16 machines, while the increase of transaction abort rate (due to optimizations [70]) and large traffic size still incur non-trivial costs, compared to the decentralized scalar timestamp (like DST).

As shown in Fig. 12, the fast network (RDMA) in VLR has a significant positive impact on all of the settings, as expected. For `TPC-E`, DST still outperforms GTS and VTS by 1.07X, and 1.32X, respectively. RDMA reduces the overhead of centralized timestamp allocation for GTS, while the impact of traffic size in VTS remains. For `TPC-C`, DST is still 4.49X (from 1.19X) and 1.76X (from 1.15X) faster than GTS and VTS.

5.2 MySQL cluster

We evaluate MySQL cluster with DST by using `TPC-C` and `SmallBank` on VAL. We increase the number of clients until the throughput is saturated. As shown in Fig. 13, with DST, MySQL cluster achieves up to 1.91X (from 1.09X) and 1.28X (from 1.07X) higher throughput for `TPC-C` and `SmallBank`, respectively. The main reason is due to enabling snapshot reads to avoid blocking for the read-only transactions. It also mitigates the contention in the read-write transactions. DST is more effective in `TPC-C` since it is more sensitive to blocking time from conflicting transactions due to relatively longer execution time compared to `SmallBank`. On the other hand, DST can provide comparable performance to RC but still guarantee serializability for correctness.

5.3 ROCOCO

We follow the methodology (benchmarks and settings) in prior work [43, 39] to evaluate ROCOCO on VAL.[13] Fig. 14 shows the performance of ROCOCO by increasing the number of concurrent requests per server. In Fig. 14(a), using DST on ROCOCO can improve the throughput of `new-order` transactions by 2.09X with 100 concurrent requests per server, due to reducing transaction aborts and skipping the validation process in read-only transactions. For example, less than 4% of `stock-level` transactions can be committed when there are more than 50 concurrent requests per server. Thus, the server CPU is wasted on retrying and validating read-only transactions. Further, as shown in Fig. 14(b), ROCOCO+DST has a much lower median latency of (read-only) `stock-level` transactions, thanks to reading a consistent snapshot by one round of execution without validation.

[13]We try our best to compare with ROCOCO-SNOW [39], which also optimizes the read-only transaction of ROCOCO. Unfortunately, it failed to run on our testbed.

5.4 A Study of DST Cost

To study the overhead from blocking and additional timestamp updates in DST, we use two workloads that share most characteristics with `TPC-C`. We tuned the workload behavior to better reflect these overheads.

Blocking overhead. One read-only transaction accesses 10 tuples, while another write-only transaction continuously updates these tuples with locking. This is considered as the worst-case scenario for DST, since the read tuples are locked most of the time. Fig. 15(a) shows the impact on the median latency of read-only transactions when varying the staleness of read timestamps. When using the current time (staleness=0ms) as the read timestamp, 10% of the reads are blocked by concurrent writes, which incur 83% overhead of the median latency (1.72ms vs. 0.96ms). With the increase of staleness (smaller timestamp), fewer reads are blocked since the tuples have been updated with larger commit timestamps. The blocking overhead becomes trivial when staleness exceeds 100ms. Note that this is an extreme case for blocking: reads always touch the locked tuples. In reality, we only observe about 160 and 200 blocks per second at each machine

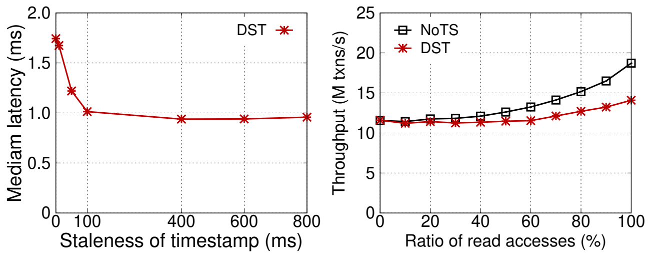


Fig. 15: *(a) The impact of latency for read-only transactions by using stale timestamp. (b) The overhead of DST with the ratio increase of read-only accesses in read-write transactions.*

under peak throughput for `TPC-E` and `TPC-C`, respectively.

Timestamp update overhead. Fig. 15(b) presents the overhead of DST to read-write transactions. Each transaction accesses 10 tuples, while some tuples are made read-only. We can see that when all tuples are updated, there is no overhead for DST, since the timestamp update will piggyback on the unlock operation. With the increase of read(-only) ratio, DST adds up to 25% overhead to the overall performance. Because DST will update the timestamp of tuples even just reading them, which requires additional synchronizations using atomic operations. Fortunately, most of the read and write sets are overlapping in OLTP workloads.

6 Discussion

Performance overhead. Compared to traditional centralized timestamp schemes, DST needs to update the timestamps of tuples in the read set for read-write transactions, which may incur additional costs. However, these operations can easily piggyback on original operations in CC protocols (see Fig. 7), like the locking and the validating in 2PL and OCC, respectively. Moreover, the read-only transaction may also update the timestamps of tuples, while it only happens as the read timestamp is larger (DEP_READ in Fig. 9). Thus, using a hybrid timestamp can effectively mitigate it. To study the potential performance overhead for DST, we designed two microbenchmarks to model the worst-case scenarios (see §5.4), and the experimental results show limited cost.

Range scans and phantom reads. DST relies on the CC protocol to detect conflicts, including range scans and phantom reads, and also needs to assign timestamps to certain "guard" (e.g., index structures) [32, 48]. For example, the next-key locking mechanism [42] is widely used by 2PL to support range scans. The CC protocol acquires such locks, and DST assigns timestamps to them. For OCC, DST assigns timestamps to the internal nodes in the index structure as the versions during the validation phase.

The SNOW theorem. The SNOW Theorem [39] describes the fact that strict serializability (`S`), non-blocking read-only transactions (`N`), one-response from each tuple (`O`), and compatible with conflicting write transactions (`W`) cannot be satisfied at the same time. Yet, SNOW-optimal and latency-optimal read-only transactions can achieve three of the above properties (i.e., `N+O+W`) without strict serializability (`S`). DST also relaxes `S` to serializability for read-only transactions, and satisfies `O` and `W` apparently. DST can simply satisfy `N` by letting reads return a relatively stale data. However, it may be not reasonable; thus, DST chooses to provide bounded staleness with much fewer blocking operations (see §5.4).

Session strict serializability. DST only ensures *serializability* to read-only transactions rather than *strict serializability*, while it is equal to or better than most snapshot-based systems [39, 19, 6, 1]. Further, DST can provide session guarantees [53, 8] (i.e., read-my-write [52] and read-after-write [40] consistency), such that read-only transactions can always observe the latest updates of read-write transactions within the same session (e.g., issued from the same client or handled by the same server). DST returns the commit timestamp to the session manager (e.g., client or server) after the transaction commits. The session manager will always use the largest observed commit timestamp as the read timestamp for successive read-only transactions.

7 Related Work

Using timestamp for snapshot reads. A centralized timestamp is the most straightforward way to support MVCC for snapshot reads, which is widely adopted by centralized systems [21, 25, 31, 28, 44, 66, 33]. Many distributed systems also use timestamps to provide MVCC [46, 17, 51, 19, 6, 14, 70], while most of them only support weaker isolation guarantees (e.g., Snapshot Isolation) [46, 6, 14, 70]. For example, Percolator [46] uses a global timestamp oracle, and NAM-DB [70] uses vectorized centralized timestamps. Spanner [19] is based on a combination of 2PL and MVCC developed in previous decades [45]. Spanner relies on TrueTime API to provide scalable timestamps for strict serializable read-only transactions and snapshot reads, which requires specific hardware (GPS and atomic clocks) to ensure clocks with bounded uncertainty. Further, the read-write transactions still require blocking to ensure the match of timestamp and transaction ordering. DST chooses to support serializable read-only transactions with bounded staleness. It requires no external timestamp service and does not block read-write transactions. RAMP [9] introduces Read Atomic isolation and uses timestamps to identify and retry inconsistent reads. TxCache [47] provides a distributed transactional cache that always returns a consistent snapshot by lazily selecting the timestamps for transactions. Causalspartan [49] also uses Hybrid Logical Clocks to optimize timestamps in causal consistency systems.

DST naturally piggybacks timestamp allocation to existing CC protocols, which avoids additional communications for maintaining timestamps. Further, DST can work with a border range of CC protocols and is orthogonal to prior optimizations on CC protocols [37, 43].

Using timestamp for concurrency control. Many systems directly leverage a timestamp-based mechanism to commit transactions orderly [12, 5, 20, 34, 35, 71, 8]. CLOCC [5] combines optimistic timestamp ordering with loosely synchronized clocks, which avoids a centralized counter for checking serializability in the original OCC protocol [29]. Granola [20] uses the timestamp based on a distributed voting mechanism to order independent transactions deterministically and treats distributed transactions in locking mode. TAPIR [71] uses loosely synchronized clocks at the clients in OCC's validation for read-write transactions. The clock drift in these systems will increase false aborts and impact the execution of read-write transactions. On the contrary, the clock drift in DST only affects the freshness of snapshot reads.

Several variant timestamp schemes have been proposed to *mitigate the cost from frequent aborts due to the violation between timestamp and transaction ordering*. Lomet et al. [38] introduce timestamp ranges to reduce transaction conflicts, while the timestamp management is centralized. MaaT [41] uses dynamic timestamp ranges to avoid distributed locking for the atomic commitment in OCC. Further, some prior systems also use decentralized timestamp schemes, but most of them focus on *optimizing one particular CC protocol*. TicToc [68] introduces a data-driven timestamp scheme for multicore platforms, which allows each read-write transaction to compute a valid commit timestamp from tuples before it commits. However, the read-only transaction still needs additional validations and incurs more aborts due to conflicts. Clock-SI [24] also uses loosely synchronized clocks to create consistent snapshots with fewer network round trips, while snapshot reads must be delayed due to concurrent transactions and clock drift. Sundial [69] uses logical timestamps as leases to reduce aborts in distributed read-write transactions. Pelieus [52] derives a commit timestamp for the read-write transaction from all involved servers (not tuples), which is used in the validation phase with different rules to support different concurrency levels (e.g., SI and Serializability).

Differently, DST is a decentralized timestamp scheme for various CC protocols and can piggyback on them efficiently. Thus, DST will not interfere with the execution of read-write transactions and has no need of extra validations and aborts.

8 Conclusion

This paper presents DST, a decentralized scalar timestamp that can unify timestamp management with existing CC protocols. We have integrated DST with two classic protocols, namely 2PL and OCC, and a recent research proposal, ROCOCO. Our evaluation with three transactional systems and three benchmarks confirmed the benefit of DST.

9 Acknowledgment

We sincerely thank our shepherd Irene Zhang and the anonymous reviewers for their insightful comments and feedback. We also thank Hong Yang for porting DST to ROCOCO. This work was supported in part by the Key-Area Research and Development Program of Guangdong Province (No. 2020B010164003), the National Natural Science Foundation of China (No. 61772335, 61925206, 61902242), the High-Tech Support Program from Shanghai Committee of Science and Technology (No. 19511121100, 20ZR1428100), and research grants from Ant Group through Ant Research Program and Alibaba Group through Alibaba Innovative Research (AIR) Program. Corresponding author: Rong Chen (rongchen@sjtu.edu.cn).

References

[1] MySQL Cluster. http://www.mysql.com/products/cluster.

[2] MySQL/InnoDB. http://www.mysql.com.

[3] PostgreSQL. http://www.postgresql.org.

[4] Oracle Database Concepts: Data Concurrency and Consistency. https://docs.oracle.com/cd/B28359_01/server.111/b28318/consist.htm, Janeury 2017.

[5] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 23–34, New York, NY, USA, 1995. ACM.

[6] M. K. Aguilera, J. B. Leners, R. Kotla, and M. Walfish. Yesquel: scalable sql storage for web applications. In *SOSP*. ACM, 2015.

[7] P. Ajoux, N. Bronson, S. Kumar, W. Lloyd, and K. Veeraraghavan. Challenges to adopting stronger consistency at scale. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

[8] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro. Cure: Strong semantics meets high availability and low latency. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pages 405–414. IEEE, 2016.

[9] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Scalable atomic visibility with ramp transactions. *ACM Transactions on Database Systems (TODS)*, 41(3):15, 2016.

[10] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the 5th biennial Conference on Innovative Data Systems Research*, CIDR'11, pages 223–234, 2011.

[11] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck. Tango: Distributed data structures over a shared log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 325–340. ACM, 2013.

[12] P. A. Bernstein and N. Goodman. Timestamp-based algorithms for concurrency control in distributed database systems. In *Proceedings of the sixth international conference on Very Large Data Bases-Volume 6*, pages 285–300. VLDB Endowment, 1980.

[13] P. A. Bernstein and N. Goodman. Multiversion concurrency control theory and algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, Dec. 1983.

[14] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian. The end of slow networks: It's time for a redesign. *Proceedings of the VLDB Endowment*, 9(7):528–539, 2016.

[15] C. Binnig, S. Hildenbrand, F. Färber, D. Kossmann, J. Lee, and N. May. Distributed snapshot isolation: global transactions pay globally, local transactions pay locally. *The VLDB Journal*, 23(6):987–1011, 2014.

[16] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. C. Li, et al. Tao: Facebook's distributed data store for the social graph. In *USENIX Annual Technical Conference*, pages 49–60, 2013.

[17] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. *ACM Trans. Database Syst.*, 34(4):20:1–20:42, Dec. 2009.

[18] Y. Chen, X. Wei, J. Shi, R. Chen, and H. Chen. Fast and general distributed transactions using rdma and htm. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 26. ACM, 2016.

[19] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[20] J. Cowling and B. Liskov. Granola: Low-overhead distributed transaction coordination. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 223–235, Boston, MA, 2012. USENIX.

[21] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1243–1254. ACM, 2013.

[22] B. Ding, L. Kot, A. Demers, and J. Gehrke. Centiman: Elastic, high performance optimistic concurrency control by watermarking. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 262–275, New York, NY, USA, 2015. ACM.

[23] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP'15, pages 54–70, New York, NY, USA, 2015. ACM.

[24] J. Du, S. Elnikety, and W. Zwaenepoel. Clock-si: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *IEEE 32nd International Symposium on Reliable Distributed Systems (SRDS)*, pages 173–184, 2013.

[25] J. M. Faleiro and D. J. Abadi. Rethinking serializable multiversion concurrency control. *Proceedings of the VLDB Endowment*, 8(11):1190–1201, 2015.

[26] A. Kalia, M. Kaminsky, and D. G. Andersen. Fasst: fast, scalable and simple distributed transactions with two-sided (rdma) datagram rpcs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 185–201. USENIX Association, 2016.

[27] A. K. M. Kaminsky and D. G. Andersen. Design guidelines for high performance rdma systems. In *2016 USENIX Annual Technical Conference*, page 437, 2016.

[28] K. Kim, T. Wang, R. Johnson, and I. Pandis. Ermia: Fast memory-optimized database system for heterogeneous workloads. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1675–1687. ACM, 2016.

[29] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981.

[30] L. Lamport, D. Malkhi, and L. Zhou. Vertical paxos and primary-backup replication. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, PODC'09, pages 312–313, New York, NY, USA, 2009. ACM.

[31] P.-Å. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. *Proceedings of the VLDB Endowment*, 5(4):298–309, 2011.

[32] P.-A. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. *Proc. VLDB Endow.*, 5(4):298–309, Dec. 2011.

[33] V. Leis, A. Kemper, and T. Neumann. Exploiting hardware transactional memory in main-memory databases. In *IEEE 30th International Conference on Data Engineering*, ICDE'14, pages 580–591. IEEE, 2014.

[34] J. Levandoski, D. Lomet, S. Sengupta, R. Stutsman, and R. Wang. High performance transactions in deuteronomy. 2015.

[35] J. Levandoski, D. Lomet, S. Sengupta, R. Stutsman, and R. Wang. Multi-version range concurrency control in deuteronomy. *Proceedings of the VLDB Endowment*, 8(13):2146–2157, 2015.

[36] J. Li, E. Michael, and D. R. K. Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 104–120, New York, NY, USA, 2017. ACM.

[37] H. Lim, M. Kaminsky, and D. G. Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 21–35. ACM, 2017.

[38] D. Lomet, A. Fekete, R. Wang, and P. Ward. Multi-version concurrency via timestamp range conflict management. In *IEEE 28th International Conference on Data Engineering*, ICDE, pages 714–725, 2012.

[39] H. Lu, C. Hodsdon, K. Ngo, S. Mu, and W. Lloyd. The snow theorem and latency-optimal read-only transactions. In *Proceedings of 12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'16, page 135, 2016.

[40] H. Lu, K. Veeraraghavan, P. Ajoux, J. Hunt, Y. J. Song, W. Tobagus, S. Kumar, and W. Lloyd. Existential consistency: Measuring and understanding consistency at facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 295–310, New York, NY, USA, 2015. ACM.

[41] H. A. Mahmoud, V. Arora, F. Nawab, D. Agrawal, and A. El Abbadi. Maat: Effective and scalable coordination of distributed transactions in the cloud. *Proc. VLDB Endow.*, 7(5):329–340, Jan. 2014.

[42] C. Mohan. Aries/kvl: A key-value locking method for concurrency control of multiaction transactions operating on b-tree indexes. In *Proceedings of the Sixteenth International Conference on Very Large Databases*, pages 392–405, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[43] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li. Extracting more concurrency from distributed transactions. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 479–494, Berkeley, CA, USA, 2014. USENIX Association.

[44] T. Neumann, T. Mühlbauer, and A. Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 677–689. ACM, 2015.

[45] A. Pavlo and M. Aslett. What's really new with newsql? *SIGMOD Rec.*, 45(2):45–55, Sept. 2016.

[46] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI*, volume 10, pages 1–15, 2010.

[47] 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*, volume 10, pages 1–15, 2010.

[48] M. Reimer. Solving the phantom problem by predicative optimistic concurrency control. In *Proceedings of the 9th International Conference on Very Large Data Bases*, VLDB '83, pages 81–88, San Francisco, CA, USA, 1983. Morgan Kaufmann Publishers Inc.

[49] M. Roohitavaf, M. Demirbas, and S. Kulkarni. Causalspartan: Causal consistency for distributed data stores using hybrid logical clocks. In *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*, pages 184–193. IEEE, 2017.

[50] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd. Efficient transaction processing in sap hana database: the end of a column store myth. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 731–742. ACM, 2012.

[51] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 385–400. ACM, 2011.

[52] D. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, and M. K. Aguilera. Transactions with consistency choices on geo-replicated cloud storage. 2013.

[53] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of the Third International Conference on on Parallel and Distributed Information Systems*, PDIS '94, pages 140–150, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[54] The H-Store Team. SmallBank Benchmark. http://hstore.cs.brown.edu/documentation/deployment/benchmarks/smallbank/.

[55] The H-Store Team. TATP Benchmark. https://github.com/apavlo/h-store/tree/master/src/benchmarks/edu/brown/benchmark/tm1/.

[56] The Transaction Processing Council. TPC-C Benchmark V5.11. http://www.tpc.org/tpcc/.

[57] The Transaction Processing Council. TPC-E Benchmark V1.14. http://www.tpc.org/tpce/.

[58] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD'12, pages 1–12. ACM, 2012.

[59] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32. ACM, 2013.

[60] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP'13, pages 18–32. ACM, 2013.

[61] V. Vasudevan, M. Kaminsky, and D. G. Andersen. Using vector interfaces to deliver millions of iops from a networked key-value storage server. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 8. ACM, 2012.

[62] Z. Wang, H. Qian, J. Li, and H. Chen. Using restricted transactional memory to build a scalable in-memory database. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys'14, pages 26:1–26:15. ACM, 2014.

[63] X. Wei, Z. Dong, R. Chen, and H. Chen. Deconstructing rdma-enabled distributed transactions: Hybrid is better! In *13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '18, pages 233–251, 2018.

[64] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 87–104, New York, NY, USA, 2015. ACM.

[65] Y. Wu, J. Arulraj, J. Lin, R. Xian, and A. Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proc. VLDB Endow.*, 10(7):781–792, Mar. 2017.

[66] Y. Wu, J. Arulraj, J. Lin, R. Xian, and A. Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proceedings of the VLDB Endowment*, 10(7):781–792, 2017.

[67] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proc. VLDB Endow.*, 8(3):209–220, Nov. 2014.

[68] X. Yu, A. Pavlo, D. Sanchez, and S. Devadas. Tictoc: Time traveling optimistic concurrency control. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1629–1642, New York, NY, USA, 2016. ACM.

[69] X. Yu, Y. Xia, A. Pavlo, D. Sanchez, L. Rudolph, and S. Devadas. Sundial: Harmonizing concurrency control and caching in a distributed oltp database management system. *Proc. VLDB Endow.*, 11(10):1289–1302, June 2018.

[70] E. Zamanian, C. Binnig, T. Harris, and T. Kraska. The end of a myth: Distributed transactions can scale. *Proc. VLDB Endow.*, 10(6):685–696, Feb. 2017.

[71] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP'15, pages 263–278, New York, NY, USA, 2015. ACM.

When to Hedge in Interactive Services

Mia Primorac
Oracle Labs[*]

Katerina Argyraki
EPFL

Edouard Bugnion
EPFL

Abstract

In online data-intensive (OLDI) services, each client request typically executes on multiple servers in parallel; as a result, "system hiccups", although rare within a single server, can interfere with many client requests and cause violations of service-level objectives. Service providers have long been fighting this "tail at scale" problem through "hedging", *i.e.,* issuing redundant queries to mask system hiccups. This, however, can potentially cause congestion that is more detrimental to tail latency than the hiccups themselves.

This paper asks: when does it make sense to hedge in OLDI services, and how can we hedge enough to mask system hiccups but not as much as to cause congestion? First, we show that there are many realistic scenarios where hedging can have no benefit—where *any* hedging-based scheduling policy, including the state-of-the-art, yields no latency reduction compared to optimal load balancing without hedging. Second, we propose LÆDGE, a scheduling policy that combines optimal load balancing with work-conserving hedging, and evaluate it in an AWS cloud deployment. We show that LÆDGE strikes the right balance: first, unlike the state of the art, it never causes unnecessary congestion; second, it performs close to an ideal scheduling policy, improving the 99th percentile latency by as much as 49%, measured on 60% system utilization—without any difficult parameter training as found in the state of the art.

1 Introduction

This work concerns Online Data-Intensive (OLDI) services like web search (searching through inverted document indices), content-based image similarity search, recommendation services, graph processing and social applications. Such services involve hundreds or thousands of "leaf" nodes, each holding a part ("shard") of the data needed to answer client requests; a tier of "root" nodes receives client requests, breaks each client request into distinct queries, forwards the queries to different leaves, and waits for the slowest query to finish in order to create the final client response.

OLDI services typically operate under hard-to-meet service-level objectives (SLOs) expressed in terms of tail latency [7, 8, 16, 18, 32, 36, 37, 40]. Each SLO captures a customer expectation and failing to meet it has concrete consequences, *e.g.,* a hit to the service provider's reputation, a loss of customers, and a drop in revenue [11, 15, 23, 73]. The nature of OLDI services makes meeting such SLOs challenging at large scale: because answering a client request involves many queries, a small fraction of slow queries can impact a significant fraction of client requests [16].

We focus on two main causes of latency variability: One is variable queuing delay, *e.g.,* due to load fluctuations [16, 20, 31, 62]. Another one is variable service time, which, in turn, comes from two distinct sources: (1) Different queries may take different amounts of time to execute on a given hardware and software stack, because of different complexities [33, 44, 53, 81]. (2) Different instances of the same query may take significantly different amounts of time to execute on a given system because of system events that are unrelated to the service itself: decisions made by an OS scheduler or power-management algorithm, interrupts, garbage collection, virtualization effects, interfering with other applications, *etc.,* [12, 16, 19, 26, 41–43, 46–48, 54, 57, 59, 69, 75, 80].

Even though a lot can and has been done to reduce latency variability, completely eliminating its causes has proved elusive. In a modern cloud environment, where different services share resources, there always exist unexpected performance "hiccups". Debugging these hiccups is notoriously hard. For instance, there is the case of an application suffering random 12*ms* scheduling delays, because a kernel feature caused the jitter of interrupt requests to be significantly higher than the timer interval [12]; or the case of non-work-conserving scheduling, in which the kernel was throttling programs that exceeded a misconfigured purchase quota [69]. But even when performance hiccups are easy to debug, fixing them is often beyond the control of the interested parties: most service providers do not own a datacenter and do not develop their

[*]The project was completed while the author was at EPFL.

own operating systems and entire software stacks—yet they still offer interactive services that need to meet strict SLOs.

This reality has given birth to *hedging*. In a modern large-scale service, each data shard is typically replicated in at least two nodes for fault tolerance [16, 55]. Hence, any query for a distinct shard can be replicated and sent to multiple nodes that serve that shard. This way, the system "hedges its bets", as it needs to wait only for the fastest response to each query. Hedging was initially proposed and adopted in combination with performance monitoring, to improve completion time of map-reduce-style [17] jobs that may take tens to hundreds of seconds [3–5]. More recently, hedging is proposed as a general solution for reducing tail latency in large-scale services, including OLDI services [16, 25, 32, 37, 39, 40, 64, 67, 72, 74, 78], where expected query completion time is orders of magnitude shorter. This change in time scale makes it significantly harder to determine whether hedging will improve tail latency or make it worse.

Hedging masks service-time variability at the cost of extra system load (caused by the replicated queries), hence extra queuing delay. So, if we take any standard hedging policy and any standard load-balancing policy (that tries to minimize latency without hedging), and we measure latency as a function of system load, we expect to see a tipping point: at first, hedging will outperform plain load balancing; however, for some offered system load, the cost of replicating queries will start to outweigh the benefit and the situation will be reversed. The challenge, then, is knowing *when* to hedge in order to operate before the tipping point.

In this paper, we look critically at hedging for OLDI services. We make two contributions:

- We study *when* hedging makes sense: when does it have the potential to improve tail latency relative to plain load balancing, and by how much? To answer this question, we define Idealized Hedge, a theoretical hedging policy that, by design, maximizes the hedging potential of any implementable hedging policy for a given setup. We experimentally compare Idealized Hedge to Per-Shard Queuing, which is, in our context, the best load-balancing policy that does not hedge [45, 76]. This allows us to identify regimes where hedging has the potential to improve tail latency, and to bound the potential improvement.
- We propose LÆDGE (short for "Load-Aware Hedge", pronounced like "*ledge*"), a combination of Per-Shard Queuing and hedging that hedges only if the current system load allows for latency improvement through hedging. The gist of LÆDGE is to only hedge when a replica is idle, through a work-conserving centralized scheduler.

Through a combination of simulations and experiments, we show that LÆDGE approximates Idealized Hedge and outperforms the state-of-the-art hedging policies. We implemented LÆDGE within an open-source OLDI framework [28] and evaluated it on a popular enterprise search engine deployed in the AWS cloud. Our experimental results closely match

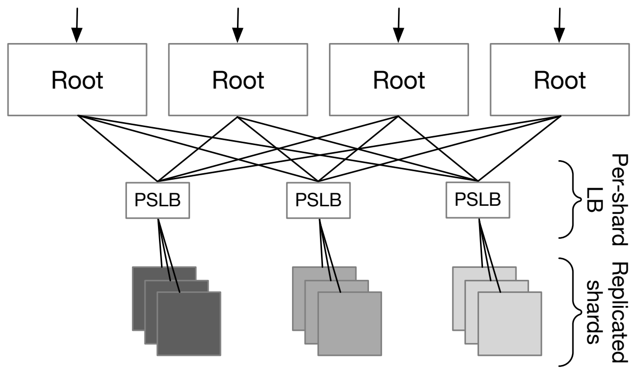


Figure 1: Architecture of an OLDI service with per-shard load balancers (PSLBs).

theoretical expectations and show significant gains: for jobs with mean service times as low as 800µs, and system utilization up to 60%, LÆDGE reduces the 99^{th} percentile latency of Per-Shard Queuing by, on average, 5.3*ms*—an improvement that corresponds to 6.4× the mean service time.

The rest of the paper is organized as follows: §2 presents our simulation setup and defines IQ-jitter—the metric we use to model and measure performance hiccups. §3 presents Idealized Hedge and uses it to identify when hedging has the potential to improve tail latency. §4 presents LÆDGE and evaluates it through simulation, while §5 evaluates it based on a system implementation using a real OLDI application deployed in the AWS cloud. §6 discusses open issues, §7 presents related work, and §8 concludes.

2 Background and Setup

In this section, we describe a standard OLDI setup (§2.1); define IQ-jitter, which helps us model performance hiccups (§2.2); summarize state-of-the-art hedging (§2.3) and load-balancing (§2.4) policies; and describe the simulation (§2.5) that drives the rest of the paper (up until §5).

2.1 OLDI Setup

We consider a cluster that serves OLDI applications, as illustrated in Fig. 1: a tier of root nodes serves client requests, while a tier of leaf nodes stores shards of application data sets; the root and leaf tiers are connected via a tier of per-shard load balancers (PSLBs). The load balancers are "per shard," in the sense that they maintain *a distinct queue per shard*. In practice, the same process implements multiple PSLBs; PSLB processes run either in servers that are dedicated to load balancing, or within the root nodes when that is permitted by the scheduling policy.

A client request requires accessing multiple data shards. When a root receives a client request, it issues one query per distinct shard, and sends each query to the corresponding

	Policy	Queuing model	Hedging probability	Load balancing
a)	*Naïve Hedge* [16, 72]	push	1	random
b)	*d-Hedge* [16, 40]	push	$Pr(RTT > d)$	random
c)	*p-Hedge* [40]	push	$q \cdot Pr(RTT > d)$	randomization + JSQ
d)	*Random load balancing*	push	0	none
e)	*Join-shortest-queue (JSQ)* [29]	push	0	JSQ
f)	*Join-bounded-shortest-queue (JBSQ)* [45]	push & pull	0	JBSQ
g)	*Per-Shard Queuing (PSQ)*	pull	0	centralized queue
h)	*Idealized Hedge (IH)* (§3)	pull	load-dependent	centralized queue
i)	*Load-aware Hedge* (LÆDGE) (§4)	pull	load-dependent	centralized queue

Table 1: Existing and proposed tail-mitigation strategies using hedging and load balancing.

PSLB, which schedules the query on one or more leaves. Each leaf is equipped with a queue and processes queries first-come, first-serve (FCFS) [40, 57], which is the best non-preemptive scheduling policy when tail latency is the most important metric [10,40,48,57,76]. To compute the final client response, the root needs one response per distinct shard.

We define the *end-to-end latency* of a client request as the time from the moment a root fans-out the original request into multiple queries until it has received at least one response per distinct shard. It is equal to the service time plus queuing delay experienced by the slowest query that needs to be answered in order to compute the final response.

2.2 IQ-jitter: Modelling Hiccups

Consider a set of leaves with identical hardware and software configuration, serving queries of a given application.

We define *IQ-jitter* (short for *intra-query jitter*), denoted by J, as the service-time variability that results from the fact that two leaves hosting the same data shard may take different amounts of time to serve the same query.

Similarly to Mirhosseini *et al.* [57], we express the query service time S as the sum of two random variables:

$$S = P + J,$$

where P is determined by query complexity and shard content/size, while J (IQ-jitter) is determined by system events that are independent of the application: OS-scheduler decisions, power-management algorithm decisions, interference by other applications—in general, the current software and hardware state of the leaf executing the query.

2.3 Hedging State of the Art

Hedging was invented to mitigate the effect of IQ-jitter on tail latency: As long as the system events causing performance hiccups are independent across leaves, two or more leaves hosting the same data shard are unlikely to all suffer a hiccup while serving the same query. Hence, by replicating a query across multiple leaves and collecting the response that arrives first, we reduce the probability of the query suffering a hiccup.

In this paper, we consider three representative hedging policies (top three rows in Table 1):

Naïve Hedge [72]: The PSLB *always and immediately* sends each query to any two leaves that host the corresponding shard. This is conceptually the simplest hedging policy as it does not require storing any state at the PSLB. It has been applied to many different contexts, including map-reduce jobs [3–5, 14, 82], DNS queries, database servers, and packet forwarding [72].

Delayed Hedge (d-Hedge) [16, 40]: For each query, the PSLB randomly picks a leaf that hosts the relevant shard and sends the query to it; if the reply does not arrive within a pre-configured, fixed delay d, the PSLB replicates the query on another leaf. When a query finishes, the PSLB cancels its replica if it exists. The value of d is tuned by the system operator to control the number of replicated queries in the system. This is the policy that was proposed by Dean *et al.* when they introduced the "tail at scale" problem [16].

Probabilistic Hedge (p-Hedge) [40]: This is similar to d-Hedge, but introduces an extra tuning knob: the probability q of replicating each delayed query; both the probability q and the delay d are trained based on the workload. This was proposed by Kaler *et al.* in their recent study of hedging policies for data centers [40] and is the most sophisticated hedging policy that we found in the literature.

We should clarify that all these policies—as any hedging policy we are aware of—limit the number of simultaneously run hedged requests to two. We follow the same strategy in all the policies that we define in this paper.

Ideally, a hedging policy walks the fine line between (a) replicating too many queries and adding too much system load (hence queuing delay), and (b) not replicating enough queries and failing to mitigate the effect of IQ-jitter on tail latency. Naïve Hedge errs toward the former (it always replicating as much as possible); Vulimiri *et al.* [72] have showed that this reduces tail latency only when system load is below 30% [72]. d-Hedge and p-Hedge provide knobs for controlling the added load; however, as we will see, they still do not enable a good balance between (a) and (b), *i.e.,* they can still unnecessarily overload the system or fail to mitigate the effect of IQ-jitter, even if their knobs are carefully tuned (§3.3).

2.4 Load Balancing: State of the Art

The standard approach to managing latency is load balancing (LB). In this paper, we compare hedging with four standard LB policies ((d) to (g) in Table 1):

Random: For each query, the PSLB randomly picks a leaf that hosts the relevant shard and sends the query to it.

Join-shortest-queue (JSQ) [29]: For each query, the PSLB picks a leaf that hosts the relevant shard and sends the query to it; of all the candidate leaves, the PSLB picks the one with the smallest number of pending queries for that shard. JSQ outperforms Random but is far from optimal for FCFS servers with highly-variable job sizes [29,34,35]. We consider it because it is simple to implement and very popular in the industry, *e.g.,* it is widely deployed in reverse-proxies [29,58].

Per-Shard Queuing (PSQ): The PSLB stores each query until a leaf that can serve it becomes available. In other words, the PSLB dispatches a query to a leaf *only* if the leaf is idle. From a queuing-theory perspective, PSQ corresponds to a single-queue $M/G/k$ model, where k is the number of leaves to choose from. In theory, this outperforms any LB policy that uses multiple distinct queues (*e.g.,* JSQ) in the presence of non-deterministic service times [68]. In practice, PSQ exposes the round-trip latency between the PSLB and the leaf tiers, which may significantly impact throughput. It is, however, an excellent candidate for low-latency environments like modern datacenters [57,60] (*e.g.,* it takes $< 20\mu$s to perform an RPC between two VMs in Microsoft Azure [22]).

Join-bounded-shortest-queue (JBSQ) [45]: This policy combines PSQ and JSQ by splitting the pending queries between the PSLB and leaf tiers. It takes a parameter n, which specifies the number of pending requests that each leaf can hold, *e.g.,* JBSQ(1) is equivalent to PSQ, while JBSQ(∞) is equivalent to JSQ. The value of n can be configured so as to hide the round-trip latency between the PSLB and leaf tiers and enable full throughput.

2.5 Simulation Setup

In the next two sections, we rely on discrete event simulation to compare hedging against standard LB policies. The goal is not to evaluate precisely how these policies perform in real systems (we use different experiments for that, later in the paper), but to understand some of their fundamental properties, *e.g.,* how does hedging compare to PSQ in an idealized setting (where PSQ is the optimal LB policy)?

Our simulation setup mimics an OLDI application deployed in a small- to medium-sized cluster: (a) The number of shards ranges from $N = 5$, which represents small-scale public-cloud deployments, and it is the default number of shards in Elasticsearch [21]; to $N = 500$ distinct shards, which represents larger public-cloud deployments [21,70]. Each shard is replicated in $r = 2$, 3 or 6 leaves (depending on the experiment). The number of replicas in OLDI applications tends to be limited given DRAM costs [55]; 6 was the highest number of replicas per shard that we found in the literature [14,40,63,70]. There are $N \times r$ leaves processing queries, all with the same hardware and software configuration. Leaves process queries FCFS [40]. All queues have infinite capacity. (b) Each PSLB fans out queries to leaves according to the simulated (hedging or LB) policy. Network latency between any two nodes is zero (revisited in § 5). (c) Client requests arrive at the root tier following an open-loop Poisson arrival process. (d) As stated earlier, guided by previous studies [57] and our own observations, we model query service time as the sum of two components, $S = P + J$. Unless otherwise noted, P follows an exponential distribution (the randomness comes from different queries of a workload), while J (IQ-jitter) follows a bimodal distribution. This means that, in any single experiment, an instance of a query executing at a leaf node experiences a hiccup with some probability (*e.g.,* 10^{-3}), while all hiccups have the same duration (*e.g.,* $15 \times \bar{P}$)—which approximates the results in [12] and reflects our observations from § 5.1. Across experiments, we vary hiccup probability and duration to model a range of real problematic system events.

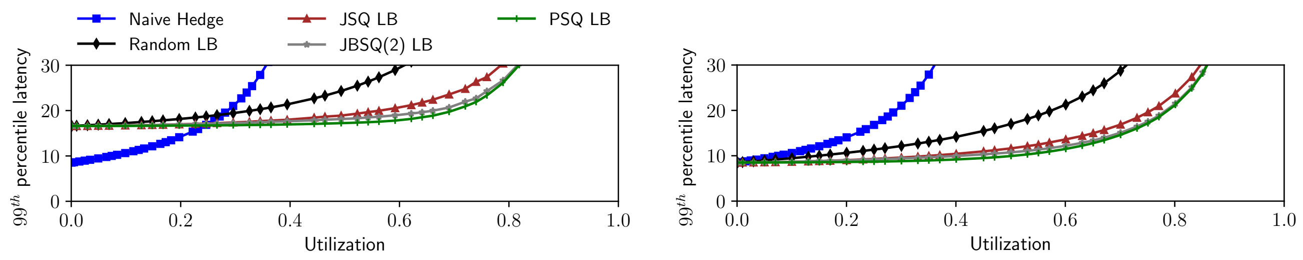
We report tail latency as a multiple of $\bar{P}$ (average application-dependent latency). For example, when we report that 99th percentile latency is equal to 30, this means that the slowest 1% of client requests experience end-to-end latency that is higher than $30 \times \bar{P}$. We think that this provides more insight than an absolute number, especially since (in these particular experiments) we are measuring a simulated, idealized setup. In several plots, we show tail latency as a function of system utilization; a maximum utilization of 1.0 corresponds to $r/\bar{S}$ queries per time unit, where r is the replication factor.

3 Idealized Hedging

In this section, we present Idealized Hedge—an idealized hedging policy—and use it to gain insight into the applicability of hedging, as a general policy, to OLDI services.

3.1 Two Simple Observations

We start by comparing all the LB policies and Naïve Hedge: Fig. 2 shows 99th percentile latency as a function of system utilization in a cluster with $N = 50$ shards, each replicated



(a) IQ-jitter with hiccup probability $\frac{1}{1000}$, hiccup duration $15 \times \bar{P}$.

(b) No IQ-jitter.

Figure 2: 99^{th} percentile latency as a function of utilization, in a cluster with 50×2 leaves, with and without IQ-jitter.

in $r = 2$ leaves, first with IQ-jitter of probability 10^{-3} and duration $15 \times \bar{P}$ (left), then without IQ-jitter (right).

We make two observations:

First, PSQ outperforms the other LB policies in all situations. This is expected from queuing theory. The performance difference is greater in the presence of IQ-jitter (Fig. 2a), which makes sense given that performance hiccups increase the dispersion of service-time distribution—and more service-time dispersion creates a bigger challenge for policies like JSQ and Random.

Second, in the presence of IQ-jitter (Fig. 2a), there is a clear "turning point": At low utilization, Naïve Hedge (despite its naïveté) delivers significantly lower tail latency than any LB policy. For instance, in an unloaded system, tail latency is $8 \times \bar{P}$ with Naïve Hedge and $17 \times \bar{P}$ with PSQ—close to a $2\times$ improvement. However, when utilization exceeds 25%, the LB policies deliver lower tail latency and higher throughput.

As a side note, in the absence of IQ-jitter (Fig. 2b), hedging unsurprisingly does not improve tail latency relative to LB, for any system utilization.

These observations suggest that a combined hedging/PSQ policy, which adapts the fraction of hedged queries to system utilization, might achieve lower tail latency than either hedging or LB alone.

3.2 The Design of Idealized Hedge

Idealized Hedge maximizes the potential of hedging to reduce latency in the following way:

1. A leaf is never idle when it can serve a pending query currently being served by at most one other leaf.

2. A leaf serves a hedged (replicated) query only when it cannot serve a non-hedged (yet-unserved) one.

The second property ensures that hedged queries never increase the queuing delay experienced by non-hedged queries. The two properties together ensure what we might call *work conservation in the presence of hedging*: no resources are idle when they could be doing useful work, and no resources are dedicated to hedging when they could be used for other work.

Idealized Hedge is not implementable because it requires perfect prediction of the completion times of currently executing queries: To ensure that the two properties stated above always hold, the system may need to cancel one copy of a hedged query currently executing on a leaf (so that the leaf can serve a new, non-hedged query that just arrived). To maximize the potential of hedging in our setup, Idealized Hedge must always cancel the copy that will take longer to complete, hence the need for perfect prediction.

We simulated Idealized Hedge as follows: Each PSLB maintains a queue with all the pending queries in order of arrival, and it knows the status of each leaf and which query it is processing (if busy). Moreover, if two copies of a hedged query are executing on different leaves and a new query arrives (that can be served by the same leaves), the PSLB's scheduler perfectly predicts which copy will finish executing first. With this knowledge, the PSLB performs the following operations: (a) Dispatches queries to the leaves in an FCFS manner using a pull-based discipline like PSQ. (b) Hedges a query as soon as a leaf that can serve it becomes idle. (c) Cancels any copy of a hedged query if another copy finishes first (we call this a *cleanup cancellation* or CC for brevity). (d) Cancels one copy of a hedged query upon arrival of a new query that can be served by the same leaf (we call this a *pre-emptive cancellation* or PC).

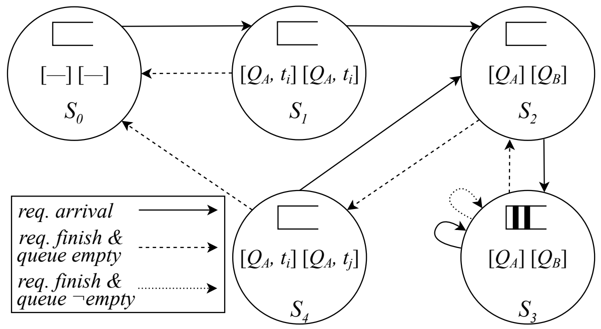
Fig. 3 shows the finite-state machine of Idealized Hedge for a given shard that is replicated in two leaves:

S_0 Both leaves are idle. There are no pending queries.

S_1 (Initial hedging) Both leaves are serving the same query Q_A, which they started to serve simultaneously; there are no other pending queries. This state occurs on a transition from S_0, following the arrival of query Q_A.

S_2 (No hedging) The two leaves are serving different queries, Q_A and Q_B, and there are no additional pending queries. This state occurs on a transition from S_1, following the arrival of query Q_B, at which point one copy of Q_A (the one that would have finished later) was pre-emptively cancelled.

S_3 (PSQ) The two leaves are serving different queries, Q_A and Q_B, and there are additional pending queries. In this



[—] [—]
S_0
$[Q_A, t_i]$ $[Q_A, t_i]$
S_1
$[Q_A]$ $[Q_B]$
S_2
$[Q_A, t_i]$ $[Q_A, t_j]$
S_4
$[Q_A]$ $[Q_B]$
S_3
req. arrival
req. finish & queue empty
req. finish & queue ¬empty

Figure 3: The finite-state machine of both Idealized Hedge and LÆDGE with CC+PC on one shard with two replicas. The states show: whether the shard queue is empty, whether replicas are running the same or different queries (Q), and whether the queries started at the same time (t).

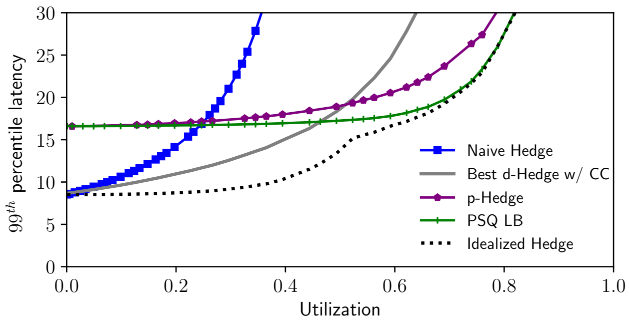


Figure 4: 99th percentile latency as a function of utilization. Ideal Hedge vs. existing hedging policies and PSQ. Same setup as in Fig. 2a.

state, when a leaf finishes serving its query, it pulls the next one from the head of the queue (without hedging), as in the PSQ policy.

S_4 (Delayed hedging) Both leaves are serving the same query Q_A, which they started to serve at different times; there are no other pending queries. This state occurs on a transition from S_2, following the completion of one query, at which point the other query (Q_A) is reissued to the otherwise idle leaf.

3.3 Idealized Hedge versus State of the Art

Fig. 4 compares Idealized Hedge against the three hedging policies (Naïve Hedge, d-Hedge, p-Hedge), as well as PSQ (the best LB policy). The experimental setup matches that of Fig. 2a (50×2 leaves and IQ-jitter with probability 10^{-3} and duration $15 \times \bar{P}$).

Idealized Hedge outperforms (as expected) the real policies and exhibits the following behavior: at low utilization (until around 5%), when the leaves are mostly in states S_0 and S_1, it behaves like Naïve Hedge; at high utilization (from around 60%), when the leaves are mostly in state S_3, it converges to PSQ; in between, it clearly outperforms (by up to $8 \times \bar{P}$) all the real policies.

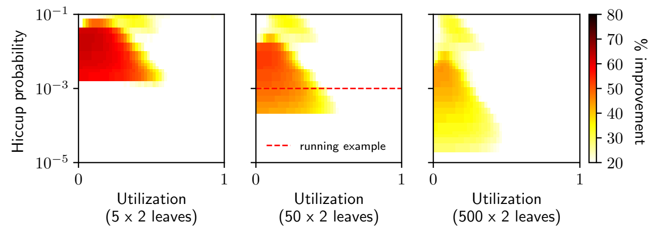
The most interesting comparison is between Idealized Hedge (black dotted line) and PSQ (green solid line with vertical lines), because it provides an upper bound on the tail-latency benefit that can be expected from *any* form of hedging. This comparison indicates two points: (1) There exists a significant utilization range (from ∼60% and up, in our setup) where no real hedging policy may bring any significant benefit relative to PSQ. (2) Outside this range, hedging *may* bring significant benefit, but the two state-of-the-art hedging policies cannot fulfill this potential. Only Naïve Hedge (blue solid line with squares) achieves all the benefit that hedging could achieve, but only at low utilization (until around 5%, in our setup). d-Hedge (gray solid line) outperforms the other real policies for a utilization range between ∼10 and ∼45%, but it remains far from Idealized Hedge. p-Hedge (purple solid line with pentagons) is outperformed by PSQ in all situations.

Of course, the behavior of d-Hedge and p-Hedge depends dramatically on how their configuration parameters are tuned; we followed all the instructions in the relevant literature, and we did our best to maximize their performance. For instance, in d-Hedge, we set the delay after which hedging occurs to $d = 5 \times \bar{P}$, because we experimentally found that higher values do not noticeably mitigate tail latency on low and medium loads; differently said, we allowed the least amount of hedging that has an impact on tail latency on low and medium loads comparable to that of Naïve Hedge (and yet the algorithm still led to congestion collapse at ∼60% utilization). In p-Hedge, we trained the parameters d and q, using the most successful of the methods explored in [40]: for each level of system utilization, we computed a "reissue budget" (the percentage of hedged queries in the system) using their iterative algorithm; then, for each level of system utilization, we trained d and q on sampled latency measurements using the proposed training algorithm that accounts for queuing delays for a fixed reissue budget. We tried sampling rates up to 80%; the results we show are for a sampling rate of 60%, because increasing it further did not significantly change the results: p-Hedge did not capture the rare hiccups through sampling, as they only occur once every 1000 queries. None of this proves that d-Hedge and p-Hedge could not perform any better, but it illustrates the difficulty of tuning them so as to achieve a desired balance between too little and too much hedging.

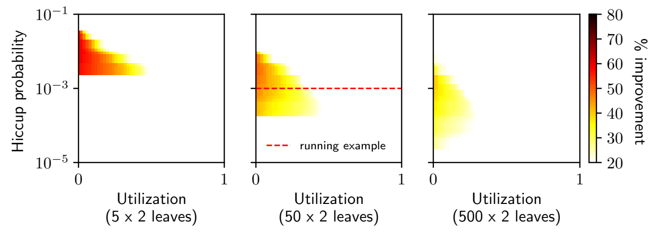
3.4 Beyond One Example

We now extend our observations beyond the specific setup of Fig. 4: how much potential does hedging have to improve tail latency as the cluster size and nature of IQ-jitter vary?

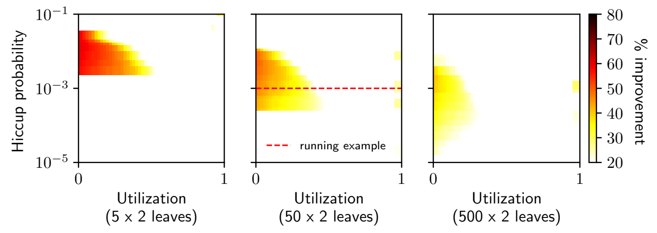
We consider the following scenarios: clusters of 5×2, 50×2, and 500×2 leaves, *i.e.,* small, medium, and large; hiccup probability ranging from 10^{-1} to 10^{-5}; hiccup duration $15 \times$



(a) Idealized Hedge

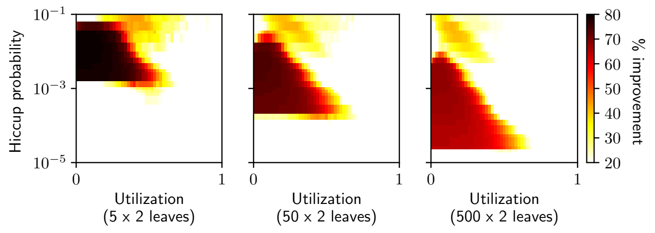


(b) LÆDGE

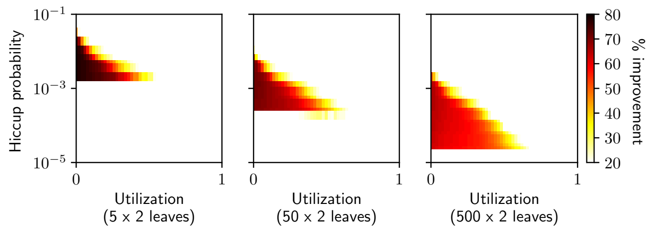


(c) LÆDGE with CC

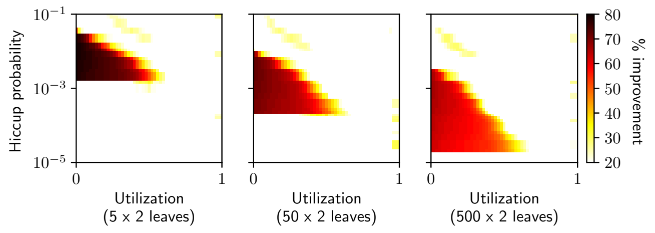
Figure 5: Heat maps showing how much various hedging policies improve the 99th percentile latency relative to PSQ. Hiccup duration is $15 \times \bar{P}$.



(a) Idealized Hedge



(b) LÆDGE



(c) LÆDGE with CC

Figure 6: Heat maps showing how much various hedging policies improve the 99th percentile latency relative to PSQ. Hiccup duration is $30 \times \bar{P}$.

$\bar{P}$ and $30 \times \bar{P}$. Regarding the latter, our choice of parameters is motivated by the literature: $15 \times \bar{P}$ has occurred as the result of badly configured timer intervals [12], while $30 \times \bar{P}$ as a result of non-conserving job-to-core allocation [69].

We summarize our results in two sets of heat maps, one for hiccup duration $15 \times \bar{P}$ (Fig. 5a), the other for $30 \times \bar{P}$ (Fig. 6a). Each heat map illustrates the relative improvement in 99th percentile latency that Idealized Hedge brings relative to PSQ: the x-axis is system utilization, the y-axis is hiccup probability (on a logarithmic scale), and the intensity of each data point is the relative improvement in the 99th percentile latency (so, a darker data point indicates higher potential for hedging to improve tail latency). We only show improvement greater than 20%, to focus on scenarios with significant improvement potential. Each column corresponds to a different cluster size: 5×2, 50×2, and 500×2 leaves, from left to right. The dashed horizontal line in Fig. 5a, middle heat map (so, 50×2 leaves) corresponds to the setup of Fig. 2a and 4.

First, we observe that **hedging cannot significantly improve tail latency when system utilization exceeds** $\sim$**60%** (all heat maps are empty beyond $\sim$60% utilization). Beyond this turning point, hedging improves latency at most by 20%, independently from cluster size and hiccup probability or duration. The intuition is simple: as system utilization increases and leaves become busier, opportunities for hedging disappear; as a result, Idealized Hedge eventually converges to PSQ. The turning point corresponds to medium-heavy utilization, where queues are starting to form, and below the point where the well-known heavy-traffic approximation determines behavior irrespective of service-time distribution [30]. Between $\sim$60% and $\sim$80% utilization, hedging provides at most 10% improvement (not visible in the heat maps); and beyond $\sim$80% utilization, no improvement at all.

Second, **hiccup duration does not affect the existence of potential improvement (the heat-map shape), only the amount of potential improvement (the heat-map intensity)**. Compare any two heat maps for the same cluster size in Figures 5a and 6a: they shade mostly the same (x, y) surface, but the heat map on the right (longer hiccup duration) is darker than the one on the left. The intuition is that, for any given cluster size, hedging can be useful only within a given hiccup probability range; outside this range, performance hiccups are either too rare or too frequent for hedging to make any difference, and this is independent of hiccup duration.

Third, **the larger the cluster size, the smaller the hiccup probability for which hedging may be useful**. For example, when the hiccup probability is between 10^{-4} and 10^{-5}, hedging may be useful only in the 1000-leaf cluster (and would be useful in larger clusters as well); in the two smaller clusters, each request needs access to fewer distinct shards, and the probability of IQ-jitter slowing down the serving of one or more of these shards becomes insignificant. Conversely, the higher the hiccup probability, the smaller the cluster size for which hedging may be useful. For example, when hiccup probability exceeds 10^{-2}, hedging may be helpful in all three clusters (but less so in the 1000-leaf cluster, where it may improve tail latency by at most 35%).

4 LÆDGE: Approaching Idealized Hedge

In this section, we present Load-Aware Hedge (LÆDGE), an implementable policy that approximates Idealized Hedge by adapting hedging to system utilization.

Algorithm 1 shows LÆDGE in pseudo code: $SQ[i]$ is a shard queue that corresponds to shard i; q is a query that belongs to request r and concerns shard s, while q' is a replica of q. Each query that concerns a given shard may be: (a) hedged, i.e., sent to two nodes that serve the shard (line 7), (b) sent to a single node that serves the shard if that is the only available one (line 9), or (c) queued up at a shard queue if zero nodes that serve the shard are available (line 11). When a node becomes available, the oldest query in the shard's queue is sent to the node (line 18). If there are no queries in the shard's queue, the oldest running query that concerns the shard is hedged, i.e., a replica of the query is sent to the node (line 21).

LÆDGE has the following properties:

1. A leaf is never idle when it can serve a pending query currently being served by at most one other leaf.

2. A leaf *starts* to serve a hedged (replicated) query when it cannot serve a non-hedged (yet-unserved) one.

Unlike Idealized Hedge, LÆDGE does not guarantee that, at any point in time, any leaf serving a hedged query could not be serving a non-hedged one; it only guarantees that when a leaf *starts* to serve a hedged query it could not be serving a non-hedged one. Like PSQ, an efficient LÆDGE implementation requires a PSLB with µs-scale round-trip latency to the leaves, which is commonly found in datacenter environments [22,45].

We implemented three variants of LÆDGE, which differ only in the type of query cancellation that they support:

Plain LÆDGE: Similarly to PSQ, a PSLB dispatches queries to leaves from a centralized queue; similarly to Idealized Hedge, a leaf serves both non-hedged and hedged queries, prioritizing the former. There are no query cancellations.

LÆDGE with CC: This policy augments plain LÆDGE with cleanup cancellation (CC), *i.e.,* all copies of a hedged query are cancelled when another copy finishes executing first. An efficient implementation of this policy requires a low-overhead mechanism for interrupting a query and cleaning up its side effects. Whether such a mechanism exists or not depends on the application itself (for instance, Boucher *et al.* [13] enabled efficient µs-scale cancellations for microservices [1,27,56] written in Rust).

Algorithm 1: Generalized LÆDGE

```
1  // Initialize a shard queue (SQ) per shard
2  $SQ[i] \leftarrow [\,],\ i \in [1,\dots,n_{shards}]$ ;
3  on request $r$ arrival
4      for each shard $s$ do
5          if available replicas of shard $s \geq 2$ then
6              // Replication on arrival
7              send $q$ and $q'$ to 2 random replicas of shard $s$;
8          else if available replicas of shard $s == 1$ then
9              send $q$ to the available replica ;  // No replication
10         else
11             enqueue $q$ to $SQ[s]$ ;
12         end
13     end
14 end;
15 on response $p$ arrival from node $n$ serving shard $s$
16     if size($SQ[s]$) $> 0$ then
17         pop a pending query $q$ from $SQ[s]$ ;
18         send $q$ to $n$ ;                          // No replication
19     else
20         if $\exists$ a non-replicated unfinished query on shard $s$ then
21             replicate the oldest query to node $n$ ;
                 //Delayed replication
22         end
23     end
24 end;
```

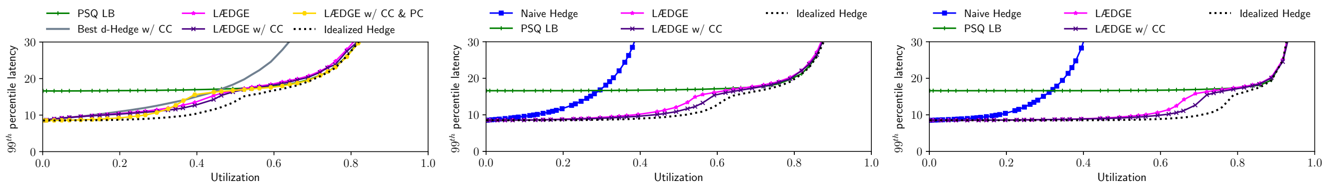
LÆDGE with CC+PC: This policy adds pre-emptive cancellation (PC), *i.e.,* one copy of a hedged query is cancelled when a new query arrives that can be served by the same leaves. The state machine corresponds to that of Idealized Hedge, shown in Fig. 3; the only difference is the lack of perfect completion-time prediction (while transitioning from S_1 to S_2, and from S_4 to S_2); instead, this policy cancels the copy that started executing most recently.

All cancellations are zero cost, in the sense that they introduce no extra processing delay and no extra communication between the PSLB and leaf tiers.

4.1 LÆDGE versus Idealized Hedge

Fig. 7a compares the three LÆDGE variants against Idealized Hedge, as well as d-Hedge and PSQ (the two best existing policies among the ones we simulated). The setup matches that of Figures 2a and 4.

We observe that plain LÆDGE reduces the gap to Idealized Hedge to at most $3.8 \times \bar{P}$ (while hiccup duration is $15 \times \bar{P}$, in this setup). At low utilization (up to ~20%), it behaves like d-Hedge, which is the best existing policy at that utilization range. From some point on (~50%), it converges to PSQ, which is the best existing policy at that utilization range. In between 20% and 50% utilization, it outperforms the existing policies, without using cancellations or parameter training, closing the average gap to Idealized Hedge to only $2.16 \times \bar{P}$.



(a) 50×2 shards (2 replicas) (b) 50×3 shards (3 replicas) (c) 50×6 shards (6 replicas)

Figure 7: 99th percentile latency as a function of utilization. Comparing LÆDGE with different number of replicas per shard.

Second, we observe that cancellations not only do not significantly help LÆDGE but may actually hinder it. Adding CCs to LÆDGE marginally improves tail latency (given our assumption of zero-cost cancellations, it could not increase it). More interestingly, adding PCs to LÆDGE-with-CC *increases* tail latency at some utilization levels, starting from $\sim$30%. It turns out that cancelling the wrong copy of a hedged query (the one that would have finished first) is an expensive mistake; without any sophisticated completion-time predictors, one is better off not cancelling at all.

To better understand these results, we completed a careful analysis of PCs and their effect on tail latency. We define the "prediction accuracy" of a LÆDGE policy with cancellations as the proportion of PCs that correctly cancel the copy that would finish later. Overall, LÆDGE achieves $> 99\%$ prediction accuracy, which is unsurprising given the rarity of IQ-jitter events. However, once we consider only the PCs where at least one copy of the cancelled query experiences IQ-jitter, LÆDGE's prediction accuracy drops significantly. For example, at 40% system utilization, when transitioning from S_1 to S_2 in the presence of IQ-jitter, LÆDGE achieves prediction accuracy $\sim$50%; when transitioning from S_4 to S_2, prediction accuracy drops to $\sim$19%. The reason is that LÆDGE does not attempt to predict whether an IQ-jitter event is likely to have occurred—it simply picks the most-recently started copy. Cancellations appear to pay off only if we can assume a good predictor of performance hiccups due to system events; while this may be feasible according to Hao *et al.* [32], LÆDGE was designed in the absence of such an assumption.

So far, we considered only $r = 2$ replicas per shard. While this is typical in practice (the replication factor is often limited by high DRAM costs [55]), research proposals consider replication factors between $r = 2$ and 6 replicas per shard [40, 63]. The simulation behind the Figures 7b and 7c is set up the same way as Figure 7a, but with 3 and 6 replicas per shard, respectively. We see that, for the larger number of replicas, (1) the tail latency of Idealized Hedge and LÆDGE follows the minimal latency of Naïve Hedge up to 60% utilization (for 6 replicas), and (2) LÆDGE continues to achieve a significant part of the tail latency reduction of Idealized Hedge.

Finally, we should note that, out of curiosity, we experimented with two more types of application-independent noise (other than bimodal): exponential and bimodal+exponential. For the former, not even Idealized Hedge can improve tail latency, PSQ is the best policy, and LÆDGE performs almost the same as PSQ; this is not surprising, given that hedging was invented to deal with noise due to unpredictable system events, which is better modelled with a bimodal distribution. For the latter, the results were almost identical to the ones we got for bimodal noise.

4.2 Beyond One Example

We now extend our observations beyond the specific setup of Fig. 7: how well does LÆDGE fulfill the hedging potential as the cluster size and nature of IQ-jitter vary?

We consider the same scenarios as in §3.4 and once more summarize our results in heat maps in Figures 5 and 6. To assess how well LÆDGE approximates Idealized Hedge, we have to compare the heat maps. To simplify the comparison, we introduce Table 2, which summarizes Figures 5 and 6 as a percentage of the "surface" of each heat map that indicates improvement above a certain threshold (20%, 30% and 40%). For instance, consider the row that corresponds to hiccup duration $30 \times \bar{P}$ and cluster size 50×2 leaves, and the two columns that correspond to Idealized Hedge and LÆDGE, with $> 30\%$ latency improvement; the two cells where this row and columns intersect indicate that Idealized Hedge achieves such improvement for 29.7 % of the data points, while LÆDGE does for 12.6 % of data points.

To summarize, LÆDGE fulfills as much as half of the hedging potential, depending on the setup. Consider again the columns that correspond to Idealized Hedge and LÆDGE with $> 30\%$ latency improvement, and compare the values of these two columns that are in the same row; LÆDGE improves from 23% to 56% of the data points that are improved by Idealized Hedge (*i.e.,* that could possibly be improved through hedging). In general, LÆDGE is closer to Idealized Hedge for the medium and large clusters, and the longer hiccup duration.

On a side note, LÆDGE does not deteriorate lower latency percentiles compared to PSQ (*e.g.,* the median), but improves the tail. The rare hiccups that we analyzed, however, only influence the tail—not, for example, the 50th percentile latency.

	% reduction	Idealized Hedge			LÆDGE			LÆDGE with CC			LÆDGE with PC		
	wrt PSQ$\rightarrow$	>20	>30	>40	>20	>30	>40	>20	>30	>40	>20	>30	>40
$15\times\bar{P}$	5×2 leaves	26.3	19.6	15.6	9.8	8.3	6.4	13.6	11.8	9.6	14.4	11.6	10.1
	50×2 leaves	29.9	21.5	15.4	13.4	9.8	3.8	16.8	12.2	5.2	15.9	12.3	9.2
	500×2 leaves	31.9	17.9	4.4	12.6	4.2	0.2	14.8	5.2	0.2	15.5	8.0	0.9
$30\times\bar{P}$	5×2 leaves	36.9	28.1	22.3	9.6	7.6	6.2	18.9	13.4	11.8	21.8	15.6	12.6
	50×2 leaves	40.5	29.7	22.5	15.2	12.6	11.1	22.6	17.4	15.5	23.3	18.2	14.5
	500×2 leaves	43.1	33.4	26.7	20.9	19.0	16.7	27.2	22.2	20.5	25.1	21.2	17.9

Table 2: Percentage of the total surface (in Figures 5 and 6) that has more than 20%, 30% and 40% reduction in 99th percentile tail latency over Per-Shard Queuing for the workloads with the hiccup duration of $15\times\bar{P}$ and $30\times\bar{P}$.

4.3 Are Cancellations Worth the Effort?

First, in agreement with [4], our sensitivity analysis confirms that cleanup cancellation (CC) improves LÆDGE only marginally. Consider the columns of Table 2 that correspond to LÆDGE and LÆDGE with CC; on average, LÆDGE with CC offers the same improvement to 2.3% more data points than LÆDGE when hiccup duration is $15\times\bar{P}$, and to 5.6% more data points than LÆDGE when hiccup duration is $30\times\bar{P}$.

Second, while our sensitivity analysis in Table 2 shows that PCs have a marginally positive impact on LÆDGE, actually there exist a few areas of highly-negative impact. We have already discussed the intuition in §4.1; further analysis shows that the effect persists across the parameter space.

In a way, the non-effectiveness of cancellations is good news for application developers: Cancellations can be complicated and expensive to implement, they often require non-trivial application changes and language-specific mechanisms to avoid memory leaks and inconsistent application state [13], as well as additional interaction between the scheduler and the leaves. The fact that our simulated zero-cost cancellations bring no significant improvement to tail latency suggests that real-life cancellations are not worth the effort in our context.

5 System Evaluation with Lucene

We now evaluate LÆDGE on Lucene, a popular open-source enterprise search engine. It is representative of OLDI services because (1) it involves sharding, and (2) client requests are interactive, with expected latency in the millisecond scale.

Our workload is Lucene's standard nightly regression test, which consists of $\sim$10,000 search queries of four different types: phrase, term, multiterm and boolean [51]. The data consists of an inverted index of 18 million Wikipedia English web pages [77], split into 16 sub-indices (for parallel execution in 16 threads). Lucene comes in C++ and Java flavors; we used the former [52].

We implemented two LB policies (Random and PSQ) and two hedging policies (Naïve Hedge and LÆDGE) in OLDIsim, Google's open-source OLDI cloud benchmarking framework [28]. We changed its architecture to support shard replication, and we added PSLBs (as in Fig. 1). The I/O part of the framework stayed unchanged: it uses the event-based `libevent` API [49], on top of vanilla Linux and TCP. We extended the framework with request generation following an open-loop Poisson arrival process.

We ran our Lucene workload on AWS EC2 virtual machines (VMs) organized in a "cluster" placement group [2] in the same availability zone. We used two VM types: (1) compute-optimized instances with 16 vCPUs @3.0 *GHz* and 32 *GB* of memory (*c*5.4*xlarge*), and (2) general-purpose instances with 16 vCPUs @2.2 *GHz* and 64 *GB* of memory (*m*5*a*.4*xlarge*). All VMs were running the default Ubuntu 16.04.6 image, kernel version 4.4.0-1092-aws.

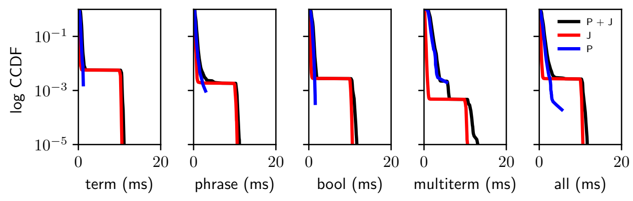
We deployed 5×2 leaf servers, *i.e.,* 5 distinct shards, each replicated in 2 leaves. When a leaf executes a query, it uses 16 parallel threads (one per vCPU). To avoid the introduction of data-driven bias in our results, we replicated the same index on all 5 shards (though, from the point of view of the application, they are still distinct shards served by different leaves). This decision simplifies the comparison with the simulation results in §4 without fundamentally changing the conclusions.

5.1 Empirical IQ-jitter Measurement

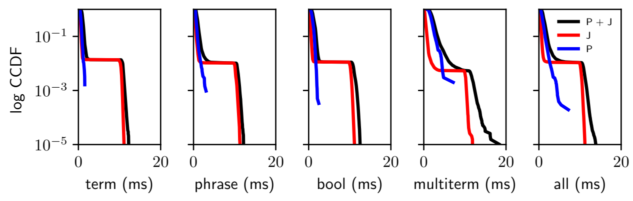
We started our experimental evaluation by measuring the real IQ-jitter experienced by our Lucene workload.

We executed the 10,000 queries of our workload 1000 times each, in a random order, always on the same server type. Consider a specific query Q. For each execution of this query, $Q_i, i = 1\ldots1000$, we measured the service time S_i (which does not include any queuing or network delay). We approximated the application-dependent component of the service time experienced by Q as the minimum service time across executions: $P(Q) = \min_{i=1\ldots1000} S_i$. Then we approximated the IQ-jitter experienced by each query execution Q_i as $J_i = S_i - P(Q)$. By putting together all the IQ-jitter values for a given query type, we obtained the IQ-jitter distribution for this query type.

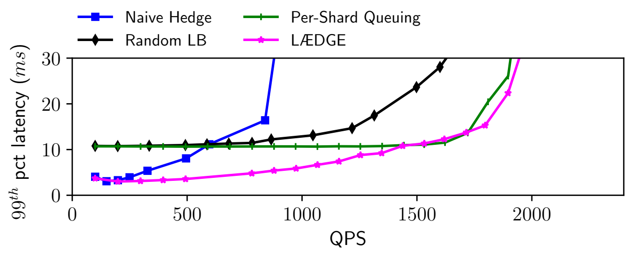
Fig. 8a and 8b show (in the form of CCDFs) the empirical distributions of P, J, and $P+J$, for the four query types of our workload, and for the two different server types. The curves differ in length as P's distribution size depends on the number of queries ($\sim$10,000), whereas the two other distribution sizes depend on the duration of the measurement experiment. The means of P and J in the two server types are 0.637 *ms* and



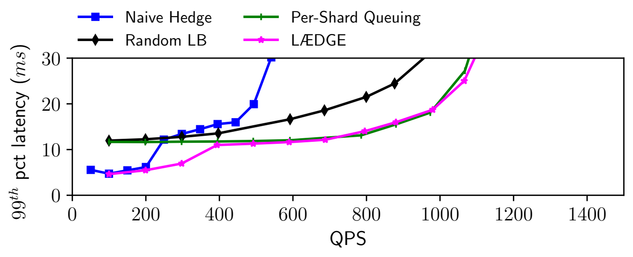
(a) Lucene service time in compute-optimized VM



(b) Lucene service time in general-purpose VM



(c) Latency vs throughput with compute-optimized VMs



(d) Latency vs throughput with general-purpose VMs

Figure 8: Mitigating the Lucene hiccups in a system implementation deployed on 2×5 leaves in EC2 VMs.

0.198 *ms* in Fig. 8a, and 0.926 *ms* and 0.444 *ms* in Fig. 8b, while the hiccup duration and probability are 10.162 *ms* and 0.0027 in Fig. 8a, and 10.249 *ms* and 0.0109 in Fig. 8b.

As a side note, we measured the latency between our VMs and found that the round-trip time is on average 91.2µs, with minimum and maximum latency 61.5µs and 150.6µs, respectively. The average network latency amounts to 10.94% and 6.7% of Lucene's mean service time ($P+J$) in the compute-optimized and general-purpose scenario, respectively.

We observe that IQ-jitter is substantial in both server types, and that the empirical distributions are consistent with our simulation setup: the application-dependent component (P) can be well approximated with an exponential distribution (a straight line on a log-based CCDF), while the IQ-jitter component (J) has a clear bimodal nature. This holds across the four query types that vary significantly in complexity (with "multiterm" queries being the most complex ones).

We investigated the reasons behind IQ-jitter and found that a significant part is due to involuntary rescheduling of Lucene threads, which occurs less frequently in the compute-optimized VMs. Of course, different Lucene deployments may experience less IQ-jitter: if the same workload runs on fully-controlled physical machines, IQ-jitter can be reduced by tweaking the OS or the application itself to mitigate the impact of involuntary thread rescheduling on tail latency.

5.2 Mitigating Tail Latency Through Hedging

Next, we measured the end-to-end latency experienced by our Lucene workload under varying system load. Figures 8c and 8d show the 99th percentile latency as a function of queries per second, for the two server types, respectively.

LÆDGE behaves as expected: it matches Naïve Hedge at low utilization, converges to PSQ at some point, and outperforms the best alternative in between. The exact behavior depends on server type: On the compute-optimized VMs, LÆDGE converges to PSQ at about 60% utilization; before that, it improves tail latency by 49%, or 5.3 *ms*, on average, relative to PSQ. On the general-purpose VMs, convergence to PSQ happens quite earlier—at about 27% utilization—and the improvement of tail latency before that point is somewhat smaller (40%, or 4.7 *ms*, on average).

Our LÆDGE implementation (deployed in the cloud with real system noise and non-zero network latencies) behaved as our simulation predicted: With compute-optimized VMs, our experimental setup consists of 5×2 leaves with IQ-jitter of hiccup probability 0.0027 and hiccup duration $15.95 \times \bar{P}$. The closest simulated setup is a cluster of the same size with IQ-jitter of the same hiccup probability and hiccup duration $15 \times \bar{P}$. This corresponds to the leftmost heat map in Fig. 5b, y-axis value 0.0027 (which is close to the base of the triangular shape of the heat map). If we observe this heat map at the given y-axis value, we can see that our simulated LÆDGE policy significantly outperforms PSQ until utilization ∼40% and then converges to PSQ at utilization ∼60%—matching the behavior of our LÆDGE implementation.

6 Discussion

Scalability: PSQ-based scheduling scales naturally with the number of leaves, because the queues are centralized *only within* a shard. The degree of shard replication is typically small due to DRAM costs [55], and easily manageable by a single PSLB. As more shards are needed, this design scales horizontally by adding more machines for (co-located)

PSLBs, as well as adding more root nodes.

Higher network latency: LÆDGE exposes the round-trip latency between the PSLB and leaf tiers. In the Amazon environment that we used for our system evaluation (§5), this latency is an order of magnitude smaller than the service time. In an environment with higher PSLB-leaf latency, however, exposing this latency could significantly impact throughput. Adapting LÆDGE to such an environment would be straightforward: we would replace LÆDGE's Per-Shard Queuing component with JBSQ [45] (described in §2.4). More specifically, in Alg. 1, instead of always enqueuing the query at line 11, the PSLB would keep track of the queue sizes on the leaf nodes and send the query to the leaf with the shortest queue.

Hedging to more replicas: Kaler *et al.* [40] have shown that hedging the same query to more than two replicas does not provide additional latency benefits. This is consistent with our evaluation results, where hedging to two replicas was enough to mitigate *rare* hiccups.

Fault-tolerance: The architecture in Fig. 1 is resilient to the failure of all of its components: (1) the state in PSLBs is soft, (2) the shards are replicated across different leaf nodes, and (3) if a root fails, another one can take over.

Scheduling in the leaf nodes: We considered only FCFS scheduling in the leaf nodes (see § 2.1). It is possible that a change in scheduling discipline affects service time. However, as long as service time has a component J that depends on application-independent events and has a bimodal distribution, our insights still apply.

7 Related Work

Taming map-reduce latency: Early attempts of hedging studied long-running map-reduce jobs with execution times measured in seconds or minutes [17]. This timescale allows for "observe-then-predict" type of algorithms, with sophisticated execution profiling based on which a decision can be made about when it pays off to hedge [4, 5, 61, 82]. Systems such as LATE [82], Mantri [5] and Dolly [4] used hedging to mitigate the stragglers that would delay the entire phase. We focused on OLDI services operating at different timescales and do not lend themselves to heavy-weight profiling.

Hedging at low latencies: § 2.3 describes Naïve Hedge [72], d-Hedge [16] and p-Hedge [40] that we compared against. State-of-the-art reissue policies such as p-Hedge [40] address the throughput limitations of Naïve Hedge and d-Hedge. Recent work by Mirhosseini *et al.* [57], advocates PSQ as a plausible means to reduce tail latency. We show in §3.3 that PSQ-based hedging policies can outperform carefully-designed push-based ones. Hedging techniques that target network latency [24, 72, 79] are out of the scope of this work.

Early stop: In some cases, the user can receive a meaningful response without her request querying all the shards. In the literature, there are two main such scenarios: First, when accuracy of the results is sacrificed for lower latency [36, 66]. Second, when the final result can be decoded if only *a part* of queries finish [46, 65]. Such approaches are orthogonal ours and can be integrated to further reduce latency.

Advanced load balancing: Lu *et al.* [50] decouple discovery of lightly-loaded servers from job assignment. Since it offers no redundancy, this technique is prone to increased tail latency in the presence of IQ-jitter. "Snitching" is another interesting LB technique in which the root node monitors request latency and picks the fastest replica [6, 71]. This technique also offers no redundancy and is ineffective in case of bursty noise [32].

Adaptive parallelism: Many researchers have tried to predict the service time of interactive services and accordingly adjust the level of parallelism in the processing nodes, or prioritize short-running queries over long-running ones [33, 38, 44, 53, 70]. This approach trades-off service-time for throughput adaptively as a function of the load but does not attempt to reduce jitter due to underlying system events.

Cancellations: Prior work extensively studied cancellations [4, 9, 13, 16, 32]. Ananthanarayanan *et al.* [4] observed that cancellations do not improve tail latency of map-reduce workloads. Recent advances have shown that canceling microsecond-scale RPCs can be feasible, but memory leaks remain a problem. Bashir *et al.* [9] recently studied duplications and cancellations at multiple layers of a cloud system, while Hao *et al.* studied application-specific or OS-level instrumentation to increase the accuracy of cancellation decisions [32]. LÆDGE is simple and can be deployed everywhere, including in the cloud. It applies duplication only at the application level, and our results show comparable latency reductions with and without cancellations. We leave the study of combining cancellations with profiling to future work.

Infrastructure jitter: §1 includes numerous examples of system events that cause jitter. Hao *et al.* [32] observe and quantify noise in EC2 with a focus on disk read and write jitter. Our work focuses on in-memory, CPU-bound applications, which also observe a varying amount of jitter determined in part by the underlying cloud VM.

8 Conclusion

This paper demonstrates that hedging can be applied without significant throughput reduction by combining the best of hedging and load balancing, *i.e.,* by hedging only when the current system load allows it. We show the drawbacks of existing policies and, in turn, propose LÆDGE, an integration of hedging within a per-shard load-balancer, and quantify its benefits against an idealized hedging policy designed to outperform any realistic hedging policy. We show that LÆDGE can yield significant latency reductions over the state-of-the-art approaches. We also validate its benefits with a cloud-based deployment of an interactive web search application.

Acknowledgments

We would like to thank our shepherd Asaf Cidon and anonymous reviewers for their constructive feedback. This research is supported in part by a research grant from VMware.

References

[1] Amazon. AWS Lambda. https://aws.amazon.com/lambda/.

[2] Amazon. Placement groups. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/placement-groups.html.

[3] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Why Let Resources Idle? Aggressive Cloning of Jobs with Dolly. In *HOTCLOUD*, 2012.

[4] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective Straggler Mitigation: Attack of the Clones. In *NSDI*, pages 185–198, 2013.

[5] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *OSDI*, pages 265–278, 2010.

[6] Apache. Cassandra snitches. https://docs.datastax.com/en/archived/cassandra/3.0/cassandra/architecture/archSnitchesAbout.html.

[7] L. A. Barroso, J. Clidaras, and U. Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2013.

[8] L. A. Barroso, J. Dean, and U. Hölzle. Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro*, 23(2):22–28, 2003.

[9] H. M. Bashir, A. B. Faisal, M. A. Jamshed, P. Vondras, A. M. Iftikhar, I. A. Qazi, and F. R. Dogar. Reducing tail latency using duplication: a multi-layered approach. In *CONEXT*, pages 246–259, 2019.

[10] A. Belay, G. Prekas, M. Primorac, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. The IX Operating System: Combining Low Latency, High Throughput, and Efficiency in a Protected Dataplane. *ACM Trans. Comput. Syst.*, 34(4):11:1–11:39, 2017.

[11] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. " O'Reilly Media, Inc.", 2016.

[12] M. Bligh, M. Desnoyers, and R. Schultz. Linux kernel debugging on google-sized clusters. In *Proceedings of the Linux Symposium*, pages 29–40, 2007.

[13] S. Boucher, A. Kalia, D. G. Andersen, and M. Kaminsky. Putting the "Micro" Back in Microservice. In *USENIX ATC*, pages 645–650, 2018.

[14] H. Casanova. Benefits and Drawbacks of Redundant Batch Requests. *J. Grid Comput.*, 5(2):235–250, 2007.

[15] Chris Jones and John Wilkes and Niall Murphy and Cody Smith. Service Level Objectives. https://landing.google.com/sre/sre-book/chapters/service-level-objectives/.

[16] J. Dean and L. A. Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, 2013.

[17] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[18] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *SOSP*, pages 205–220, 2007.

[19] C. Delimitrou and C. Kozyrakis. iBench: Quantifying interference for datacenter applications. In *IISWC*, pages 23–33, 2013.

[20] C. Delimitrou and C. Kozyrakis. Amdahl's law for tail latency. *Commun. ACM*, 61(8):65–72, 2018.

[21] Elasticsearch. https://www.elastic.co/what-is/elasticsearch.

[22] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. M. Caulfield, E. S. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. G. Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *NSDI*, pages 51–66, 2018.

[23] B. Forrest. Bing and Google Agree: Slow Pages Lose Users. http://radar.oreilly.com/2009/06/bing-and-google-agree-slow-pag.html.

[24] A. Frömmgen, T. Erbshauser, A. P. Buchmann, T. Zimmermann, and K. Wehrle. ReMP TCP: Low latency multipath TCP. In *ICC*, pages 1–7, 2016.

[25] K. Gardner, S. Zbarsky, S. Doroudi, M. Harchol-Balter, and E. Hyytiä. Reducing Latency via Redundant Requests: Exact Analysis. In *SIGMETRICS*, pages 347–360, 2015.

[26] P. Garraghan, X. Ouyang, R. Yang, D. McKee, and J. Xu. Straggler Root-Cause and Impact Analysis for Massive-scale Virtualized Cloud Datacenters. *IEEE Trans. Serv. Comput.*, 12(1):91–104, 2019.

[27] Google. Cloud functions. https://cloud.google.com/functions/.

[28] Google Cloud Platform Blog. Benchmarking web search latencies. http://cloudplatform.googleblog.com/2015/03/benchmarking-web-search-latencies.html.

[29] V. Gupta, M. Harchol-Balter, K. Sigman, and W. Whitt. Analysis of join-the-shortest-queue routing for web server farms. *Perform. Evaluation*, 64(9-12):1062–1081, 2007.

[30] S. Halfin and W. Whitt. Heavy-Traffic Limits for Queues with Many Exponential Servers. *Oper. Res.*, 29(3):567–588, 1981.

[31] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. W. Moore, G. Antichi, and M. Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *SIGCOMM*, pages 29–42, 2017.

[32] M. Hao, H. Li, M. H. Tong, C. Pakha, R. O. Suminto, C. A. Stuardo, A. A. Chien, and H. S. Gunawi. MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface. In *SOSP*, pages 168–183, 2017.

[33] M. E. Haque, Y. H. Eom, Y. He, S. Elnikety, R. Bianchini, and K. S. McKinley. Few-to-Many: Incremental Parallelism for Reducing Tail Latency in Interactive Services. In *ASPLOS-XX*, pages 161–175, 2015.

[34] M. Harchol-Balter. Task assignment with unknown duration. *J. ACM*, 49(2):260–288, 2002.

[35] M. Harchol-Balter, M. Crovella, and C. D. Murta. On Choosing a Task Assignment Policy for a Distributed Server System. *J. Parallel Distributed Comput.*, 59(2):204–228, 1999.

[36] Y. He, S. Elnikety, J. R. Larus, and C. Yan. Zeta: scheduling interactive services with partial execution. In *SOCC*, page 12, 2012.

[37] V. Jalaparti, P. Bodík, S. Kandula, I. Menache, M. Rybalkin, and C. Yan. Speeding up distributed request-response workflows. In *SIGCOMM*, pages 219–230, 2013.

[38] M. Jeon, S. Kim, S. won Hwang, Y. He, S. Elnikety, A. L. Cox, and S. Rixner. Predictive parallelization: taming tail latencies in web search. In *SIGIR*, pages 253–262, 2014.

[39] G. Joshi, E. Soljanin, and G. W. Wornell. Efficient Redundancy Techniques for Latency Reduction in Cloud Systems. *TOMPECS*, 2(2):12:1–12:30, 2017.

[40] T. Kaler, Y. He, and S. Elnikety. Optimal Reissue Policies for Reducing Tail Latency. In *SPAA*, pages 195–206, 2017.

[41] M. Kambadur, T. Moseley, R. Hank, and M. A. Kim. Measuring interference between live datacenter applications. In *SC*, page 51, 2012.

[42] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: predictable low latency for data center applications. In *SOCC*, page 9, 2012.

[43] H. Kasture and D. Sánchez. Ubik: efficient cache sharing with strict qos for latency-critical workloads. In *ASPLOS-XIX*, pages 729–742, 2014.

[44] S. Kim, Y. He, S. won Hwang, S. Elnikety, and S. Choi. Delayed-Dynamic-Selective (DDS) Prediction for Reducing Extreme Tail Latency in Web Search. In *WSDM*, pages 7–16, 2015.

[45] M. Kogias, G. Prekas, A. Ghosn, J. Fietz, and E. Bugnion. R2P2: Making RPCs first-class datacenter citizens. In *USENIX ATC*, pages 863–880, 2019.

[46] J. Kosaian, K. V. Rashmi, and S. Venkataraman. Parity models: erasure-coded resilience for prediction serving systems. In *SOSP*, pages 30–46, 2019.

[47] J. Leverich and C. Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *EUROSYS*, pages 4:1–4:14, 2014.

[48] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *SOCC*, pages 9:1–9:14, 2014.

[49] libevent – an event notification library. https://libevent.org/.

[50] Y. Lu, Q. Xie, G. Kliot, A. Geller, J. R. Larus, and A. G. Greenberg. Join-Idle-Queue: A novel load balancing algorithm for dynamically scalable web services. *Perform. Evaluation*, 68(11):1056–1071, 2011.

[51] Lucene nightly benchmarks. https://home.apache.org/~mikemccand/lucenebench/.

[52] Lucene++. https://github.com/luceneplusplus/LucenePlusPlus.

[53] C. Macdonald, N. Tonellotto, and I. Ounis. Learning to predict response times for online query scheduling. In *SIGIR*, pages 621–630, 2012.

[54] A. Maricq, D. Duplyakin, I. Jimenez, C. Maltzahn, R. Stutsman, R. Ricci, and A. Klimovic. Taming Performance Variability. In *OSDI*, pages 409–425, 2018.

[55] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch. Power management of online data-intensive services. In *ISCA*, pages 319–330, 2011.

[56] Microsoft. Azure functions. https://azure.microsoft.com/services/functions/.

[57] A. Mirhosseini and T. F. Wenisch. The Queuing-First Approach for Tail Management of Interactive Services. *IEEE Micro*, 39(4):55–64, 2019.

[58] NGINX. https://www.nginx.com/.

[59] D. M. Novakovic, N. Vasic, S. Novakovic, D. Kostic, and R. Bianchini. DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments. In *USENIX ATC*, pages 219–230, 2013.

[60] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: distributed, low latency scheduling. In *SOSP*, pages 69–84, 2013.

[61] X. Ouyang, P. Garraghan, B. Primas, D. McKee, P. Townend, and J. Xu. Adaptive Speculation for Efficient Internetware Application Execution in Clouds. *ACM Trans. Internet Techn.*, 18(2):15:1–15:22, 2018.

[62] G. Prekas, M. Kogias, and E. Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *SOSP*, pages 325–341, 2017.

[63] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. R. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *ISCA*, pages 13–24, 2014.

[64] Z. Qiu and J. F. Pérez. Evaluating the Effectiveness of Replication for Tail-Tolerance. In *CCGRID*, pages 443–452, 2015.

[65] K. V. Rashmi, M. Chowdhury, J. Kosaian, I. Stoica, and K. Ramchandran. EC-Cache: Load-Balanced, Low-Latency Cluster Caching with Online Erasure Coding. In *OSDI*, pages 401–417, 2016.

[66] L. Ravindranath, J. Padhye, R. Mahajan, and H. Balakrishnan. Timecard: controlling user-perceived delays in server-based mobile applications. In *SOSP*, pages 85–100, 2013.

[67] N. B. Shah, K. Lee, and K. Ramchandran. When Do Redundant Requests Reduce Latency? *IEEE Trans. Communications*, 64(2):715–722, 2016.

[68] J. F. Shortle, J. M. Thompson, D. Gross, and C. M. Harris. *Fundamentals of queueing theory*, volume 399. John Wiley & Sons, 2018.

[69] R. Sites. Data Center Computers: Modern Challenges in CPU Design. `https://youtu.be/QBu2Ae8-8LM?t=1205`.

[70] A. Sriraman and T. F. Wenisch. μTune: Auto-Tuned Threading for OLDI Microservices. In *OSDI*, pages 177–194, 2018.

[71] P. L. Suresh, M. Canini, S. Schmid, and A. Feldmann. C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection. In *NSDI*, pages 513–527, 2015.

[72] A. Vulimiri, P. B. Godfrey, R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker. Low latency via redundancy. In *CONEXT*, pages 283–294, 2013.

[73] J. Wagner. Why Performance Matters. `https://developers.google.com/web/fundamentals/performance/why-performance-matters/`.

[74] D. Wang, G. Joshi, and G. W. Wornell. Efficient task replication for fast response times in parallel computation. In *SIGMETRICS*, pages 599–600, 2014.

[75] Q. Wang, Y. Kanemasa, J. Li, D. Jayasinghe, T. Shimizu, M. Matsubara, M. Kawaba, and C. Pu. Detecting Transient Bottlenecks in n-Tier Applications through Fine-Grained Analysis. In *ICDCS*, pages 31–40, 2013.

[76] A. Wierman and B. Zwart. Is Tail-Optimal Scheduling Possible? *Oper. Res.*, 60(5):1249–1257, 2012.

[77] Wikipedia: Database download. `https://en.wikipedia.org/wiki/Wikipedia:Database_download#english`.

[78] Z. Wu, C. Yu, and H. V. Madhyastha. CosTLO: Cost-Effective Redundancy for Lower Latency Variance on Cloud Storage Services. In *NSDI*, pages 543–557, 2015.

[79] H. Xu and B. Li. RepFlow: Minimizing flow completion times with replicated flows in data centers. In *INFOCOM*, pages 1581–1589, 2014.

[80] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey. Bobtail: Avoiding Long Tails in the Cloud. In *NSDI*, pages 329–341, 2013.

[81] X. Yang, S. M. Blackburn, and K. S. McKinley. Elfen Scheduling: Fine-Grain Principled Borrowing from Latency-Critical Workloads Using Simultaneous Multithreading. In *USENIX ATC*, pages 309–322, 2016.

[82] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *OSDI*, pages 29–42, 2008.

Move Fast and Meet Deadlines: Fine-grained Real-time Stream Processing with Cameo

Le Xu[1*], Shivaram Venkataraman[2], Indranil Gupta[1], Luo Mai[3], and Rahul Potharaju[4]

[1]University of Illinois at Urbana-Champaign, [2]UW-Madison, [3]University of Edinburgh, [4]Microsoft

Abstract

Resource provisioning in multi-tenant stream processing systems faces the dual challenges of keeping resource utilization high (without over-provisioning), and ensuring performance isolation. In our common production use cases, where streaming workloads have to meet latency targets and avoid breaching service-level agreements, existing solutions are incapable of handling the wide variability of user needs. Our framework called Cameo uses fine-grained stream processing (inspired by actor computation models), and is able to provide high resource utilization while meeting latency targets. Cameo dynamically calculates and propagates priorities of events based on user latency targets and query semantics. Experiments on Microsoft Azure show that compared to state-of-the-art, the Cameo framework: i) reduces query latency by 2.7×in single tenant settings, ii) reduces query latency by 4.6×in multi-tenant scenarios, and iii) weathers transient spikes of workload.

1 Introduction

Stream processing applications in large companies handle tens of millions of events per second [16, 68, 89]. In an attempt to scale and keep total cost of ownership (TCO) low, today's systems: a) parallelize operators across machines, and b) use multi-tenancy, wherein operators are collocated on shared resources. Yet, resource provisioning in production environments remains challenging due to two major reasons:
(i) High workload variability. In a production cluster at a large online services company, we observed orders of magnitude variation in event ingestion and processing rates, *across time*, *across data sources*, *across operators*, and *across applications*. This indicates that resource allocation needs to be dynamically tailored towards each operator in each query, in a nimble and adept manner at run time.
(ii) Latency targets vary across applications. User expectations come in myriad shapes. Some applications require quick responses to events of interest, i.e., short end-to-end

*Contact author: Le Xu <lexu1@illinois.edu>

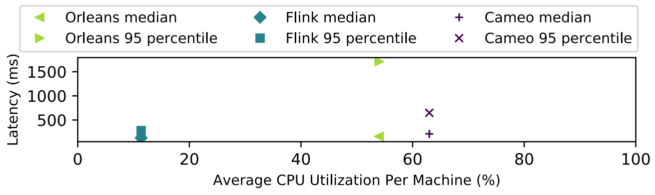


Figure 1: *Slot-based system (Flink), Simple Actor system (Orleans), and our framework Cameo.*

latency. Others wish to maximize throughput under limited resources, and yet others desire high resource utilization. Violating such user expectations is expensive, resulting in breaches of service-level agreements (SLAs), monetary losses, and customer dissatisfaction.

To address these challenges, we explore a new *fine-grained* philosophy for designing a multi-tenant stream processing system. Our key idea is to provision resources to each operator based solely on its *immediate* need. Concretely we focus on deadline-driven needs. Our fine-grained approach is inspired by the recent emergence of event-driven data processing architectures including actor frameworks like Orleans [10, 25] and Akka [1], and serverless cloud platforms [5, 7, 11, 51].

Our motivation for exploring a fine-grained approach is to enable resource sharing directly among operators. This is more efficient than the traditional *slot-based* approach, wherein operators are assigned dedicated resources. In the slot-based approach, operators are mapped onto processes or threads—examples include task slots in Flink [27], instances in Heron [57], and executors in Spark Streaming [90]. Developers then need to either assign applications to a dedicated subset of machines [13], or place execution slots in resource containers and acquire physical resources (CPUs and memory) through resource managers [8, 47, 84].

While slot-based systems provide isolation, they are hard to dynamically reconfigure in the face of workload variability. As a result it has become common for developers to "game" their resource requests, asking for over-provisioned resources, far above what the job needs [34]. Aggressive users starve other jobs which might need immediate resources, and the upshot is unfair allocations and low utilization.

At the same time, today's fine-grained scheduling systems like Orleans, as shown in Figure 1, cause high tail latencies. The figure also shows that a slot-based system (Flink on YARN), which maps each executor to a CPU, leads to low resource utilization. The plot shows that our approach, Cameo, can provide both high utilization and low tail latency.

To realize our approach, we develop a new priority-based framework for fine-grained distributed stream processing. This requires us to tackle several *architectural* design challenges including: 1) translating a job's performance target (deadlines) to priorities of individual messages, 2) developing interfaces to use real-time scheduling policies such as earliest deadline first (EDF) [65], least laxity first (LLF) [69] etc., and 3) low-overhead scheduling of operators for prioritized messages. We present *Cameo*, a new scheduling framework designed for data streaming applications. Cameo:

- *Dynamically* derives priorities of operators, using both: a) *static input*, e.g., job deadline; and b) *dynamic stimulus*, e.g., tracking stream progress, profiled message execution times.
- Contributes new mechanisms: i) *scheduling contexts*, which propagate scheduling states along dataflow paths, ii) a *context handling* interface, which enables pluggable scheduling strategies (e.g., laxity, deadline, etc.), and iii) tackles required scheduling issues including per-event synchronization, and semantic-awareness to events.
- Provides low-overhead scheduling by: i) using a stateless scheduler, and ii) allowing scheduling operations to be driven purely by message arrivals and flow.

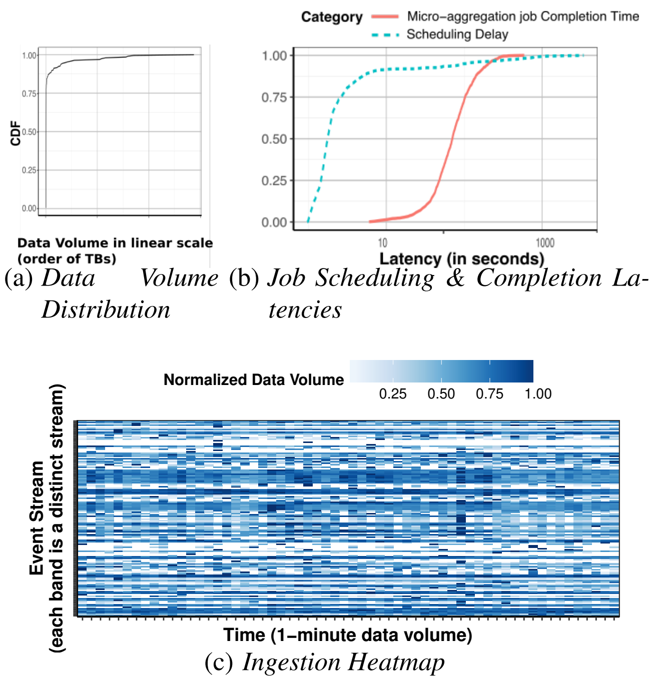
We build Cameo on Flare [68], which is a distributed data flow runtime built atop Orleans [10, 25]. Our experiments are run on Microsoft Azure, using production workloads. Cameo, using a laxity-based scheduler, reduces latency by up to 2.7× in single-query scenarios and up to 4.6× in multi-query scenarios. Cameo schedules are resilient to transient workload spikes and ingestion rate skews across sources. Cameo's scheduling decisions incur less than 6.4% overhead.

2 Background and Motivation

2.1 Workload Characteristics

We study a production cluster that ingests more than 10 PB per day over several 100K machines. The shared cluster has several internal teams running streaming applications which perform debugging, monitoring, impact analysis, etc. We first make key observations about this workload.

Long-tail streams drive resource over-provisioning. Each data stream is handled by a standing *streaming* query, deployed as a dataflow job. As shown in Figure 2(a), we first observe that 10% of the streams process a majority of the data. Additionally, we observe that a long tail of streams, each processing small amount data, are responsible for over-



(a) *Data Volume Distribution* (b) *Job Scheduling & Completion Latencies*

(c) *Ingestion Heatmap*

Figure 2: *Workload characteristics collected from a production stream analytics system.*

provisioning—their users rarely have any means of accurately gauging how many nodes are required, and end up over-provisioning for their job.

Temporal variation makes resource prediction difficult. Figure 2(c) is a heat map showing incoming data volume for 20 different stream sources. The graph shows a high degree of variability across both sources and time. A single stream can have spikes lasting one to a few seconds, as well as periods of idleness. Further, this pattern is continuously changing. This points to the need for an agile and fine-grained way to respond to temporal variations, as they are occurring.

Users already try to do fine-grained scheduling. We have observed that instead of continuously running streaming applications, our users prefer to provision a cluster using external resource managers (e.g., YARN [2], Mesos [47]), and then run periodic micro-batch jobs. Their implicit aim is to improve resource utilization and throughput (albeit with unpredictable latencies). However, Figure 2(b) shows that this ad-hoc approach causes overheads as high as 80%. This points to the need for a common way to allow all users to perform fine-grained scheduling, without a hit on performance.

Latency requirements vary across jobs. Finally, we also see a wide range of latency requirements across jobs. Figure 2(b) shows that the job completion time for the micro-aggregation jobs ranges from less than 10 seconds up to 1000 seconds. This suggests that the range of SLAs required by queries will vary across a wide range. This also presents an opportunity for priority-based scheduling: applications have longer latency constraints tend to have greater flexibility in

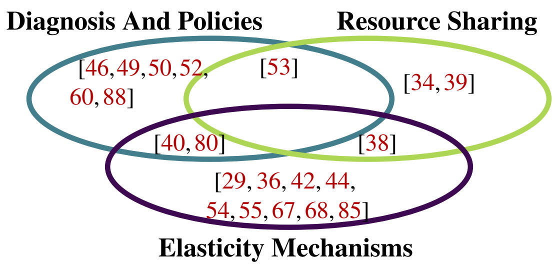


Figure 3: *Existing Dataflow Reconfiguration Solutions.*

terms of *when* its input can be processed (and vice versa).

2.2 Prior Approaches

Dynamic resource provisioning for stream processing. Dynamic resource provisioning for streaming data has been addressed primarily from the perspective of dataflow reconfiguration. These works fall into three categories as shown in Figure 3:
i) *Diagnosis And Policies*: Mechanisms for when and how resource re-allocation is performed;
ii) *Elasticity Mechanisms*: Mechanisms for efficient query reconfiguration; and
iii) *Resource Sharing*: Mechanisms for dynamic performance isolation among streaming queries.
These techniques make changes to the dataflows in reaction to a performance metric (e.g., latency) deteriorating.

Cameo's approach does not involve changes to the dataflow. It is based on the insight that the streaming engine can delay processing of those query operators which will not violate performance targets right away. This allows us to quickly prioritize and provision resources proactively for those other operators which could immediately need resources. At the same time, existing reactive techniques from Figure 3 are orthogonal to our approach and can be used alongside our proactive techniques.

The promise of event-driven systems. To achieve fine-grained scheduling, a promising direction is to leverage emerging event-driven systems such as actor frameworks [43,74] and serverless platforms [24]. Unlike slot-based stream processing systems like Flink [27] and Storm [83], operators here are not mapped to specific CPUs. Instead event-driven systems maintain centralized queues to host incoming messages and dynamically dispatch messages to available CPUs. This provides an opportunity to develop systems that can manage a unified queue of messages across query boundaries, and combat the over-provisioning of slot-based approaches. Recent proposals for this execution model also include [11,24,26,58].

Cameo builds on the rich legacy of work from two communities: classical real-time systems [63,75] and first-generation stream management systems (DSMS) in the database community [14, 15, 31, 71]. The former category has produced rich scheduling algorithms, but unlike Cameo, none build a full working system that is flexible in policies, or support streaming operator semantics. In the latter category the closest to our work are event-driven approaches [14,22,28]. But these do not interpret stream progress to derive priorities or support trigger analysis for distributed, user-defined operators. Further, they adopt a centralized, stateful scheduler design, where the scheduler *always* maintains state for all queries, making them challenging to scale.

Achieving Cameo's goal of dynamic resource provisioning is challenging. Firstly, messages sent by user-defined operators are a black-box to event schedulers. Inferring their impact on query performance requires new techniques to analyze and re-prioritize said messages. Secondly, event-driven schedulers must scale with message volume and not bottleneck.

3 Design Overview

Assumptions, System Model: We design Cameo to support streaming queries on clusters shared by cooperative users, e.g., within an organization. We also assume that the user specifies a latency target at query submission time, e.g., derived from product and service requirements.

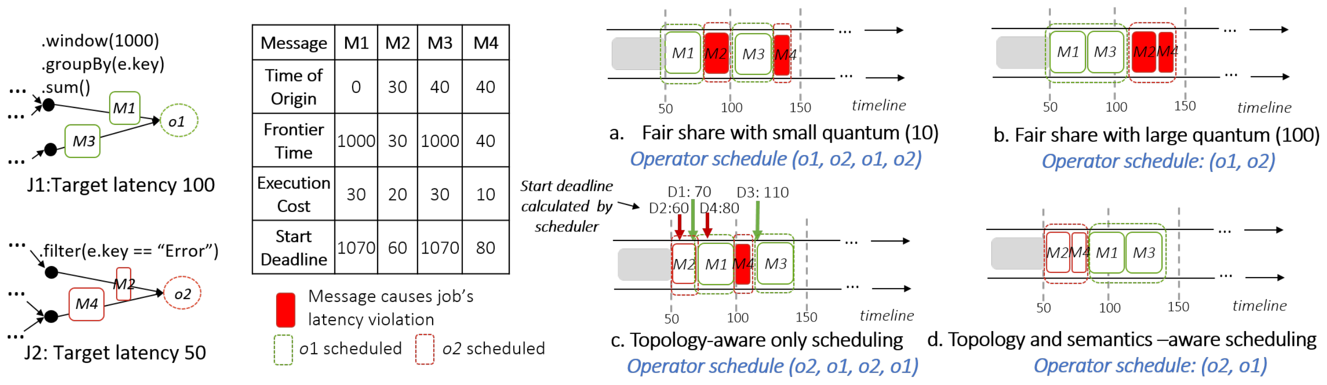
The architecture of Cameo consists of two major components: (i) a scheduling strategy which determines message priority by interpreting the semantics of query and data streams given a latency target. (Section 4), and (ii) a scheduling framework that 1. enables message priority to be generated using a pluggable strategy, and 2. schedules operators dynamically based on their current pending messsages' priorities (Section 5).

Cameo prioritizes operator processing by computing the *start deadlines* of arriving messages, i.e., latest time for a message to start execution at an operator without violating the downstream dataflow's latency target for that message. Cameo continuously reorders operator-message pairs to prioritize messages with earlier deadlines.

Calculating priorities requires the scheduler to continuously book-keep both: (i) per-job static information, e.g., latency constraint/requirement[1] and dataflow topology, and (ii) dynamic information such as the timestamps of tuples being processed (e.g., stream progress [19,61]), and estimated execution cost per operator. To scale such a fine-grained scheduling approach to a large number of jobs, Cameo utilizes *scheduling contexts*— data structures attached to messages that capture and transport information required to generate priorities.

The scheduling framework of Cameo has two levels. The upper level consists of *context converters*, embedded into each operator. A context converter modifies and propagates scheduling contexts attached to a message. The lower level is a *stateless scheduler* that determines target operator's priority by interpreting scheduling context attached to the message. We also design a programmable API for a pluggable scheduling strategy that can be used to handle scheduling contexts. In summary, these design decisions make our scheduler scale to a large number of jobs with low overhead.

[1]We use latency constraint and latency requirement interchangeably.


.window(1000)
.groupBy(e.key)
.sum()
M1
M3
o1
J1:Target latency 100
.filter(e.key == "Error")
M2
M4
o2
J2: Target latency 50
Message causes job's latency violation
o1 scheduled
o2 scheduled
Start deadline calculated by scheduler
M1 M2 M3 M4
50 100 150 timeline
a. Fair share with small quantum (10)
Operator schedule (o1, o2, o1, o2)
M1 M3 M2 M4
b. Fair share with large quantum (100)
Operator schedule: (o1, o2)
D1: 70 D3: 110
D2:60 D4:80
M2 M1 M4 M3
c. Topology-aware only scheduling
Operator schedule (o2, o1, o2, o1)
M2 M4 M1 M3
d. Topology and semantics –aware scheduling
Operator schedule: (o2, o1)

Message	M1	M2	M3	M4
Time of Origin	0	30	40	40
Frontier Time	1000	30	1000	40
Execution Cost	30	20	30	10
Start Deadline	1070	60	1070	80

Figure 4: *Scheduling Example: J1 is batch analytics, J2 is latency-sensitive. Fair-share scheduler creates schedules "a" and "b". Topology-aware scheduler reduces violations ("c"). Semantics-aware scheduler further reduces violations ("d"). We further explain these examples in Section 4.2*

Example. We present an example highlighting our approach. Consider a workload, shown in Figure 4, consisting of two streaming dataflows $J1$ and $J2$ where $J1$ performs a batch analytics query and $J2$ performs a latency sensitive anomaly detection pipeline. Each has an output operator processing messages from upstream operators. The default approach used by actor systems like Orleans is to: i) order messages based on arrival, and ii) give each operator a fixed time duration (called "quantum") to process its messages. Using this approach we derive the schedule "a" with a small quantum, and a schedule "b" with a large quantum — both result in two latency violations for $J2$. In comparison, Cameo discovers the opportunity to postpone less latency-sensitive messages (and thus their target operators). This helps $J2$ meet its deadline by leveraging topology and query semantics. This is depicted in schedules "c" and "d". This example shows that *when* and *how long* an operator is scheduled to run should be dynamically determined by the priority of the next pending message. We expand on these aspects in the forthcoming sections.

4 Scheduling Policies in Cameo

One of our primary goals in Cameo is to enable fine-grained scheduling policies for dataflows. These policies can prioritize messages based on information, like the deadline remaining or processing time for each message, etc. To enable such policies, we require techniques that can calculate the priority of a message for a given policy.

We model our setting as a non-preemptive, non-uniform task time, multi-processor, real-time scheduling problem. Such problems are known to be NP-Complete offline and cannot be solved optially online without complete knowledge of future tasks [33, 81]. Thus, we consider how a number of commonly used policies in this domain, including Least-Laxity-First (LLF) [69], Earliest-Deadline-First (EDF) [65] and Shortest-Job-First (SJF) [82], and describe how such policies can be used for event-driven stream processing.We use the LLF policy as the default policy in our description below.

The above policies try to prioritize messages to avoid violating latency constraints. Deriving the priority of a message requires analyzing the impact of each operator in the dataflow on query performance. We next discuss how being deadline-aware can help Cameo derive appropriate priorities. We also discuss how being aware of query semantics can further improve prioritization.

Symbol	Definition
ID_M	ID of Message M.
ddl_M	Message start deadline.
o_M	target operator of M.
C_{o_M}	Estimated execution cost of M on its target operator.
t_M, and p_M	Physical (and logical) time associated with the last event required to produce M.
L	Dataflow latency constraint of the dataflow that M belongs to.
p_{M_F}, and t_{M_F}	Frontier progress, and frontier time.

Table 1: *Notations used in paper for message M.*

4.1 Definitions and Underpinnings

Event. Input data arrives as *events*, associated with a *logical time* [30] that indicates the *stream progress* of these events in the input stream.

Dataflow job and operators. A dataflow job consists of a DAG of *stages*. Each stage operates a user-defined function. A stage can be parallelized and executed by a set of dataflow *operators*.

We say an operator o_k is *invoked* when it processes its input message, and o_k is *triggered* when it is invoked and leads to an output message, which is either passed downstream to further operators or the final job output.

Cameo considers two types of operators: i) regular operators that are triggered immediately on invocation; and ii) windowed operators [61] that partitions data stream into sections by logical times and triggers only when all data from the section are observed.

Message timestamps. We denote a message M as a tuple $(o_M, (p_M, t_M))$, where: a) o_M is the operator executing the message; b) p_M and t_M record the *logical* and *physical* time of the input stream that is associated with M, respectively. Intuitively, M is influenced by input stream with logical time $\leq p_M$. Physical time t_M marks the system time when p_M is observed at a source operator.

We denote C_{o_M} as the estimated time to process message M on target operator O, and L as the latency constraint for the dataflow that M belongs to.

Latency. Consider a message M generated as the output of a dataflow (at its sink operator). Consider the set of all events E that influenced the generation of M. We define latency as the difference between the last arrival time of any event in E and the time when M is generated.

4.2 Calculating Message Deadline

We next consider the LLF scheduling policy where we wish to prioritize messages which have the least laxity (i.e., flexibility). Intuitively, this allows us to prioritize messages that are closer to violating their latency constraint. To do this, we discuss how to determine the *latest time* that a message M can start executing at operator O without violating the job's latency constraint. We call this as the *start deadline* or in short the *deadline* of the message M, denoted as ddl_M. For the LLF scheduler, ddl_M is the message priority (lower value implies higher priority).

We describe how to derive the priority (deadline) using topology-awareness and then query (semantic)-awareness.

4.2.1 Topology Awareness

Single-operator dataflow, Regular operator. Consider a dataflow with only one regular operator o_M. The latency constraint is L. If an event occurs at time t_M, then M should complete processing before $t_M + L$. The start deadline, given execution estimate C_{o_M}, is:

$$ddl_M = t_M + L - C_{o_M} \tag{1}$$

Multiple-operator dataflow, Regular operator. For an operator o inside a dataflow DAG that is invoked by message M, the start deadline of M needs to account for execution time of downstream operators. We estimate the maximum of execution times of critical path [49] from o to any output operator as C_{path}. The start deadline of M is then:

$$ddl_M = t_M + L - C_{O_M} - C_{path} \tag{2}$$

Schedule "c" of Figure 4 showed an example of topology-aware scheduling and how topology awareness helps reduce violations. For example, $ddl_{M2} = 30 + 50 - 20 = 60$ means that $M2$ is promoted due to its urgency. We later show that even when query semantics are not available (e.g., UDFs), Cameo improves scheduling with topology information alone. Note that upstream operators are not involved in this calculation. C_{O_M} and C_{path} can be calculated by profiling.

4.2.2 Query Awareness

Cameo can also leverage dataflow semantics, i.e., knowledge of user-specified commands inside the operators. This enables the scheduler to identify messages which can tolerate further delay without violating latency constraints. This is common for windowed operations, e.g., a `WindowAggregation` operator can tolerate delayed execution if a message's logical time is at the start of the window as the operator will only produce output at the end of a window. Window operators are very common in our production use cases.

Multiple-operator dataflow, Windowed operator. Consider M that targets a windowed operator o_M, Cameo is able to determine (based on dataflow semantics) to what extent M can be delayed without affecting latency. This requires Cameo to identify the minimum logical time (p_{M_F}) required to trigger the target window operator. We call p_{M_F} *frontier progress*. Frontier progress denotes the stream progress that needs to be observed at the window operator before a window is complete. Thus a windowed operator will not produce output until frontier progresses are observed at all source operators. We record the system time when all frontier progresses become available at all sources as *frontier time*, denoted as t_{M_F}.

Processing of a message M can be safely delayed until all the messages that belong in the window have arrived. In other words when computing the start deadline of M, we can extend the deadline by $(t_{M_F} - t_M)$. We thus rewrite Equation 2 as:

$$ddl_M = \mathbf{t_{M_F}} + L - C_{O_M} - C_{path} \tag{3}$$

An example of this schedule was shown in schedule "d" of Figure 4. With query-awareness, scheduler derives t_{M_F} and postpones $M1$ and $M3$ in favor of $M2$ and $M4$. Therefore operator $o2$ is prioritized over $o1$ to process $M2$ then $M4$.

The above examples show the derivation of priority for a LLF scheduler. Cameo also supports scheduling policies including commonly used policies like EDF, SJF etc. In fact, the priority for EDF can be derived by a simple modification of the LLF equations. Our EDF policy considers the deadline of a message prior to an operator executing and thus we can compute priority for EDF by omitting C_{O_M} term in Equation 3. For SJF we can derive the priority by setting $ddl_M = C_{O_M}$—while SJF is not deadline-aware we compare its performance to other policies in our evaluation.

4.3 Mapping Stream Progress

For Equation 3 frontier time t_{M_F} may not be available until the target operator is triggered. However, for many fixed-sized window operations (e.g., `SlidingWindow`, `TumblingWindow`, etc.), we can *estimate* t_{M_F} based on the message's logical time

p_M. Cameo performs two steps: first we apply a TRANSFORM function to calculate p_{M_F}, the logical time of the message that triggers o_M. Then, Cameo infers the frontier time t_{M_F} using a PROGRESSMAP function. Thus t_{M_F} = PROGRESSMAP(TRANSFORM(p_M)). We elaborate below.

Step 1 (Transform): For a windowed operator, the completion of a window at operator o triggers a message to be produced at this operator. Window completion is marked by the increment of window ID [61,62], calculated using the stream's logical time. For message M that is sent from upstream operator o_u to downstream operator o_d, p_{M_F} can be derived using p_M using on a TRANSFORM function. With the definition provided by [62], Cameo defines TRANSFORM as:

$$p_{M_F} = \text{TRANSFORM}(p_M) = \begin{cases} (p_M/S_{o_d}+1)\cdot S_{o_d} & S_{o_u} < S_{o_d} \\ p_M & \text{otherwise} \end{cases}$$

For a sliding window operator o_d, S_{o_d} refers to the *slide size*, i.e., value step (in terms of logical time) for each window completion to trigger target operator. For the tumbling window operation (i.e., windows cover consecutive, non-overlapping value step), S_{o_u} equals the window size. For a message sent by an operator o_u that has a shorter slide size than its targeting operator o_d, p_{M_F} will be increased to the logical time to trigger o_d, that is, $= (p_M/S_{o_d}+1)\cdot S_{o_d}$.

For example if we have a tumbling window with window size 10 s, then the expected frontier progress, i.e., p_{M_F}, will occur every 10th second (1, 11, 21 ...). Once the window operator is triggered, the logical time of the resultant message is set to p_{M_F}, marking the latest time to influence a result.

Step 2 (ProgressMap): After deriving the frontier progress p_{M_F} that triggers the next dataflow output, Cameo then estimates the corresponding frontier time t_{M_F}. A temporal data stream typically has its logical time defined in one of three different time domains:

(1) *event time* [3,6]: a totally-ordered value, typically a timestamp, associated with original data being processed;

(2) *processing time*: system time for processing each operator [19]; and

(3) *ingestion time*: the system time of the data first being observed at the entry point of the system [3,6].

Cameo supports both event time and ingestion time. For processing time domain, M's timestamp could be generated when M is observed by the system.

To generate t_{M_F} based on progress p_{M_F}, Cameo utilizes a PROGRESSMAP function to map logical time p_{M_F} to physical time t_{M_F}. For a dataflow that defines its logical time by data's ingestion time, logical time of each event is defined by the time when it was observed. Therefore, for *all* messages that occur in the resultant dataflow, logical time is assigned by the system at the origin as $t_{M_F} = \text{PROGRESSMAP}(p_{M_F}) = p_{M_F}$.

For a dataflow that defines its logical time by the data's event time, $t_{M_F} \neq p_{M_F}$. Our stream processing run-time provides channel-wise guarantee of in-order processing for all target operators. Thus Cameo uses linear regression to map p_{M_F} to t_{M_F}, as: $t_{M_F} = \text{PROGRESSMAP}(p_{M_F}) = \alpha \cdot p_{M_F} + \gamma$, where α and γ are parameters derived via a linear fit with running window of historical p_{M_F}'s towards their respective t_{M_F}'s. E.g., For same tumbling window with window size 10s, if p_{M_F} occurs at times $(1, 11, 21\ldots)$, with a 2s delay for the event to reach the operator, t_{M_F} will occur at times $(3, 13, 23\ldots)$.

We use a linear model due to our production deployment characteristics: the data sources are largely real time streams, with data ingested soon after generation. Users typically expect events to affect results within a constant delay. Thus the logical time (event produced) and the physical time (event observed) are separated by only a small (known) time gap. When an event's physical arrival time cannot be inferred from stream progress, we treat windowed operators as regular operators. Yet, this conservative estimate of laxity does not hurt performance in practice.

5 Scheduling Mechanisms in Cameo

We next present Cameo's architecture that addresses three main challenges:

1 How to make static and dynamic information from both upstream and downstream processing available during priority assignment?

2 How can we efficiently perform fine-grained priority assignment and scheduling that scales with message volume?

3 How can we admit pluggable scheduling policies without modifying the scheduler mechanism?

Our approach to address the above challenges is to separate out the priority assignment from scheduling, thus designing a two-level architecture.This allows priority assignment for user-defined operators to become programmable.To pass information between the two levels (and across different operators) we piggyback information atop messages passed between operators.

More specifically, Cameo addresses challenge 1 by propagating *scheduling contexts* with messages. To meet challenge 2, Cameo uses a two-layer scheduler architecture. The top layer, called the *context converter*, is embedded into each operator and handles scheduling contexts whenever the operator sends or receives a message. The bottom layer, called the *Cameo scheduler*, interprets message priority based on the scheduling context embedded within a message and updates a priority-based data structure for both operators and operators' messages. Our design has advantages of: (i) avoiding the bottleneck of having a centralized scheduler thread calculate priority for each operator upon arrival of messages, and (ii) only limiting priority to be per-message. This allow the operators, dataflows, and the scheduler, to all remain stateless.

To address 3 Cameo allows the priority generation process to be implemented through the context handling API. A context converter invokes the API with each operator.

5.1 Scheduling Contexts

Scheduling contexts are data structures attached to messages, capturing message priority, and information required to perform priority-based scheduling. Scheduling contexts are *created*, *modified*, and *relayed* alongside their respective messages. Concretely, scheduling contexts allow capture of scheduling states of both upstream and downstream execution. A scheduling context can be seen and modified by both context converters and the Cameo scheduler. There are two kinds of contexts:

1. **Priority Context** (PC): PC is necessary for the scheduler to infer the priority of a message. In Cameo PCs are defined to include local and global priority as $(ID, PRI_{local}, PRI_{global}, Dataflow\_DefinedField)$. PRI_{local} and PRI_{global} are used for applications to enclose message priorities for scheduler to determine execution order, and $Dataflow\_DefinedField$ includes upstream information required by the pluggable policy to generate message priority.

A PC is attached to a message before the message is sent. It is either created at a source operator upon receipt of an event, or inherited and modified from the upstream message that triggers the current operator. Therefore, a PC is seen and modified by all executions of upstream operators that lead to the current message. This enables PC to address challenge 1 by capturing information of dependant upstream execution (e.g., stream progress, latency target, etc.).

2. **Reply Context** (RC): RC meets challenge 1 by capturing periodic feedback from the downstream operators. RC is attached to an acknowledgement message [2], sent by the target operator to its upstream operator after a message is received. RCs provide processing feedback of the target operator and all its downstream operators. RCs can be aggregated and relayed recursively upstream through the dataflow.

Cameo provides a programmable API to implement these scheduling contexts and their corresponding policy handlers in context converters. API functions include:

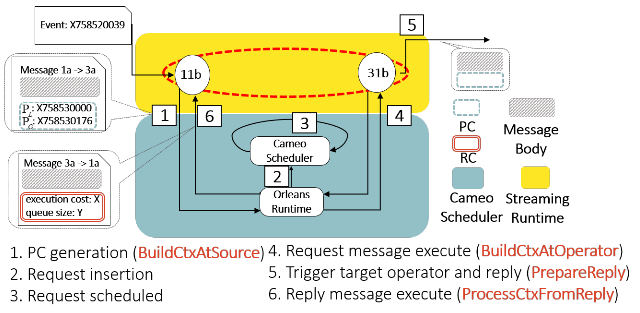
1. **function** BUILDCXTATSOURCE(EVENT e) that creates a PC upon receipt of an event e;
2. **function** BUILDCXTATOPERATOR(MESSAGE M) that modifies and propagates a PC when an operator is invoked (by M) and ready to send a message downstream;
3. **function** PROCESSCTXFROMREPLY(MESSAGE r) that processes RC attached to an acknowledgement message r received at upstream operator; and
4. **function** PREPAREREPLY(MESSAGE r) that generates RC containing user-defined feedbacks, attached to r sent by a downstream operator.

5.2 System Architecture

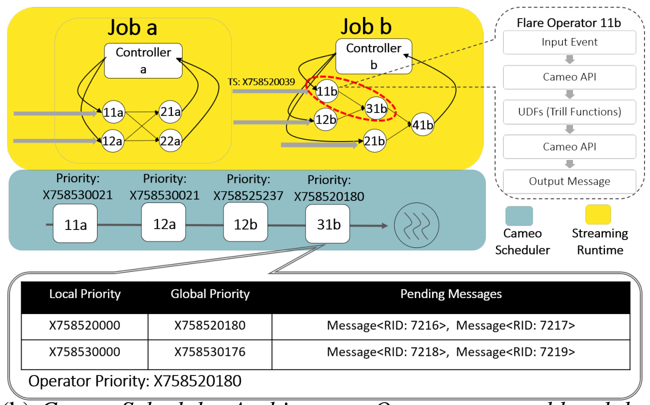
Figure 5(a) shows context converters at work. After an event is generated at a source operator $1a$ (step 1), the converter creates a PC through BUILDCXTATSOURCE and sends the message to Cameo scheduler. The target operator is scheduled (step 2) with the priority extracted from the PC, before it is executed. Once the target operator $3a$ is triggered (step 4), it calls BUILDCTXATOPERATOR, modifying and relaying PC with its message to downstream operators. After that $3a$ sends an acknowledgement message with an RC (through PREPAREREPLY) back to $1a$ (step 5). RC is then populated by the scheduler with runtime statistics (e.g, CPU time, queuing delays, message queue sizes, network transfer time, etc.) before it is scheduled and delivered at the source operator (step 6).

Cameo enables scheduling states to be managed and transported alongside the data. This allows Cameo to meet challenge 2 by keeping the scheduler away from centralized state maintenance and priority generation. The Cameo scheduler manages a two level priority-based data structure, shown in Figure 5(b). We use PRI_{local} to determine M's execution priority within its target operator, and PRI_{global} of the next message in an operator to order all operators that have pending messages. Cameo can schedule at either message granularity or a coarser time quanta. While processing a message, Cameo *peeks* at the priority of the next operator in the queue. If the next operator has higher priority, we swap with the current operator after a fixed time quantum (tunable).

[2]A common approach used by many stream processing systems [27, 57, 83] to ensure processing correctness



(a) *Scheduling contexts circulating between two operators.*



Local Priority	Global Priority	Pending Messages
X758520000	X758520180	Message<RID: 7216>, Message<RID: 7217>
X758530000	X758530176	Message<RID: 7218>, Message<RID: 7219>

Operator Priority: X758520180

(b) *Cameo Scheduler Architecture. Operators sorted by global priority. Messages at an operator sorted by local priority.*

Figure 5: *Cameo Mechanisms.*

Algorithm 1 Priority Context Conversion

```
1:  function BUILDCXTATSOURCE(EVENT $e$)  ▷ Generate PC for message $M_e$ at source triggered by event $e$
2:      $\texttt{PC}(M_e) \leftarrow$ INITIALIZEPRIORITYCONTEXT()
3:      $\texttt{PC}(M_e).(PRI_{local}, PRI_{global}) \leftarrow (e.p_e, e.t_e)$
4:      $\texttt{PC}(M_e) \leftarrow$ CONTEXTCONVERT($\texttt{PC}(M_e), \texttt{RC}_{\texttt{local}}$)
5:      return $\texttt{PC}(M_e)$
6:  function BUILDCXTATOPERATOR(MESSAGE $M_n$)  ▷ Generate PC for message $M_d$ at an intermediate operator triggered by upstream message $M_u$
7:      $\texttt{PC}(M_d) \leftarrow \texttt{PC}(M_u)$
8:      $\texttt{PC}(M_d).(PRI_{local}, PRI_{global}) \leftarrow \texttt{PC}(M_u).(p_{M_F}, t_{M_F})$
9:      $\texttt{PC}(M_d) \leftarrow$ CONTEXTCONVERT($\texttt{PC}(M_d), \texttt{RC}_{\texttt{local}}$)
10:     return $\texttt{PC}(M_d)$
11: function CXTCONVERT($\texttt{PC}(M), \texttt{RC}$)  ▷ Calculating message priority based on $\texttt{PC}(M), \texttt{RC}$ provided
12:     $p_{M_F} \leftarrow$ TRANSFORM($\texttt{PC}(M).p_M$)
13:     $t_{M_F} \leftarrow$ PROGRESSMAP($p_{M_F}$)  ▷ As in Section 4.3
14:     if $t_{M_F}$ defined in stream event time then
15:         PROGRESSMAP.UPDATE($\texttt{PC}.t_M, \texttt{PC}.p_M$)  ▷ Improving prediction model as in Section 5.3
16:     $\texttt{PC}(M).p_M, \texttt{PC}(M).t_M \leftarrow p_{M_F}, t_{M_F}$
17:     $ddl_M \leftarrow t_{M_F} + \texttt{PC}(M).L - \texttt{RC}.C_m - \texttt{RC}.C_{path}$
18:     $\texttt{PC}(M).(PRI_{local}, PRI_{global}) \leftarrow (p_{M_F}, ddl_M)$
19: function PROCESSCTXFROMREPLY(MESSAGE $r$)  ▷ Retrieve reply message's RC and store locally
20:     $\texttt{RC}_{\texttt{local}}.update(r.\texttt{RC})$
21: function PREPAREREPLY(MESSAGE $r$)  ▷ Recursively update maximum critical path cost $C_{path}$ before reply
22:     if SENDER($r$) = Sink then
23:         $r.\texttt{RC} \leftarrow$ INITIALIZEREPLYCONTEXT()
24:     else $r.\texttt{RC}.C_{path} \leftarrow \texttt{RC}.C_m + \texttt{RC}.C_{path}$
```

5.3 Implementing the Cameo Policy

To implement the scheduling policy of Section 4, a PC is attached to message M (denoted as $PC(M)$) with these fields:

ID	PRI_{local}	PRI_{global}	$Dataflow-DefinedField$
ID_M	p_{M_F}	ddl_{M_F}	(p_{M_F}, t_{M_F}, L)

The core of Algorithm 1 is CXTCONVERT, which generates PC for downstream message M_d (denoted as $\texttt{PC}(M_d)$), triggered by $\texttt{PC}(M_u)$ from the upstream triggering message. To schedule a downstream message M_d triggered by M_u, Cameo first retrieves stream progress p_{M_u} contained in $\texttt{PC}(M_u)$. It then applies the two-step process (Section 4.3) to calculate frontier time t_{M_F} using p_{M_u}. This may extend a message's deadline if the operator is not expected to trigger immediately (e.g., windowed operator). We capture p_{M_F} and estimated t_{M_F} in PC as message priority and propagate this downstream. Meanwhile, p_{M_u} and t_{M_u} are fed into a linear model to improve future prediction towards t_{M_F}. Finally, the context converter computes message priority ddl_{M_u} using t_{M_F} as described in Section 4.

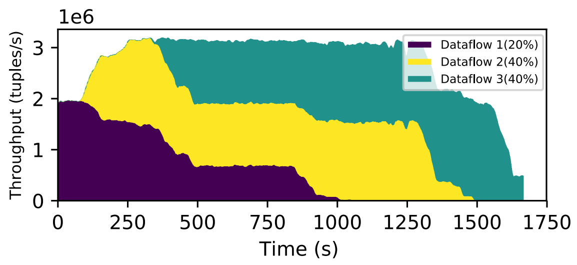


Figure 6: *Proportional fair sharing using Cameo.*

Cameo utilizes RC to track critical path execution cost C_{path} and execution cost C_{O_M}. RC contains the processing cost (e.g., CPU time) of the downstream critical path up to the current operator, obtained via profiling.

5.4 Customizing Cameo: Proportional Fair Scheduling

We next show how the pluggable scheduling policy in Cameo can be used to support other performance objectives, thus satisfying 3 . For instance, we show how a token-based rate control mechanism works, where token rate equals desired output rate. In this setting, each application is granted tokens per unit of time, based on their target sending rate. If a source operator exceeds its target sending rate, the remaining messages (and all downstream traffic) are processed with operator priority reduced to minimum. When capacity is insufficient to meet the aggregate token rate, all dataflows are downgraded equally. Cameo spreads tokens proportionally across the next time interval (e.g., 1 sec) by tagging each token with the timestamp at each source operator. For token-ed messages, we use token tag PRI_{global}, and interval ID as PRI_{local}. Messages without tokens have PRI_{global} set to `MIN_VALUE`. Through PC propagation, all downstream messages are processed when no tokened traffic is present.

Figure 6 shows Cameo's token mechanism. Three dataflows start with 20% (12), 40% (24), and 40% (24) tokens as target ingestion rate per source respectively. Each ingests 2M events/s, starting 300 s apart, and lasting 1500 s. Dataflow 1 receives full capacity initially when there is no competition. The cluster is at capacity after Dataflow 3 arrives, but Cameo ensures token allocation translates into throughput shares.

6 Experimental Evaluation

We next present experimental evaluation of Cameo. We first study the effect of different queries on Cameo in a single-tenant setting. Then for multi-tenant settings, we study Cameo's effects when:

- **Varying environmental parameters** (Section 6.2): This includes: a) workload (tenant sizes and ingestion rate), and b) available resources, i.e., worker thread pool size, c) workload bursts.
- **Tuning internal parameters and optimization** (Section 6.3): We study: a) effect of scheduling granularity, b)

frontier prediction for event time windows, and c) starvation prevention.

We implement streaming queries in Flare [68] (built atop Orleans [10, 25]) by using Trill [30] to run streaming operators.We compare Cameo vs. both i) default **Orleans** (version 1.5.2) scheduler, and ii) a custom-built **FIFO** scheduler. By default, we use the 1 ms minimum re-scheduling grain (Section 5.2). This grain is generally shorter than a message's execution time. Default Orleans implements a global run queue of messages using a ConcurrentBag [9] data structure. ConcurrentBag optimizes processing throughput by prioritizing processing thread-local tasks over the global ones. For the FIFO scheduler, we insert operators into the global run queue and extract them in FIFO order. In both approaches, an operator processes its messages in FIFO order.

Machine configuration. We use DS12-v2 Azure virtual machines (4 vCPUs/56GB memory/112G SSD) as server machines, and DS11-v2 Azure virtual machines (2 vCPUs/14GB memory/28G SSD) as client machines [12]. Single-tenant scenarios are evaluated on a single server machine. Unless otherwise specified, all multi-tenant experiments are evaluated using a 32-node Azure cluster with 16 client machines.

Evaluation workload. For the multi-job setting we study performance isolation under concurrent dataflow jobs. Concretely, our workload is divided into two control groups:

- **Latency Sensitive Jobs (Group 1):** This is representative of jobs connected to user dashboards, or associated with SLAs, ongoing advertisement campaigns, etc. Our workload jobs in Group 1 have sparse input volume across time (1 msg/s per source, with 1000 events/msg), and report periodic results with shorter aggregation windows (1 second). These have strict latency constraints.
- **Bulk Analytic Jobs (Group 2):** This is representative of social media streams being processed into longer-term analytics with longer aggregation windows (10 seconds). Our Group 2 jobs have input of both higher and variable volume and high speed, but with lax latency constraints.

Our queries feature multiple stages of windowed aggregation parallelized into a group of operators. Each job has 64 client sources. All queries assume input streams associated with event time unless specified otherwise.

Latency constraints. In order to determine the latency constraint of one job, we run multiple instances of the job until the resource (CPU) usage reaches 50%. Then we set the latency constraint of the job to be twice the tail (95th percentile) latency. This emulates the scenario where users with experience in isolated environments deploy the same query in a shared environment by moderately relaxing the latency constraint. Unless otherwise specified, a latency target is marked with grey dotted line in the plots.

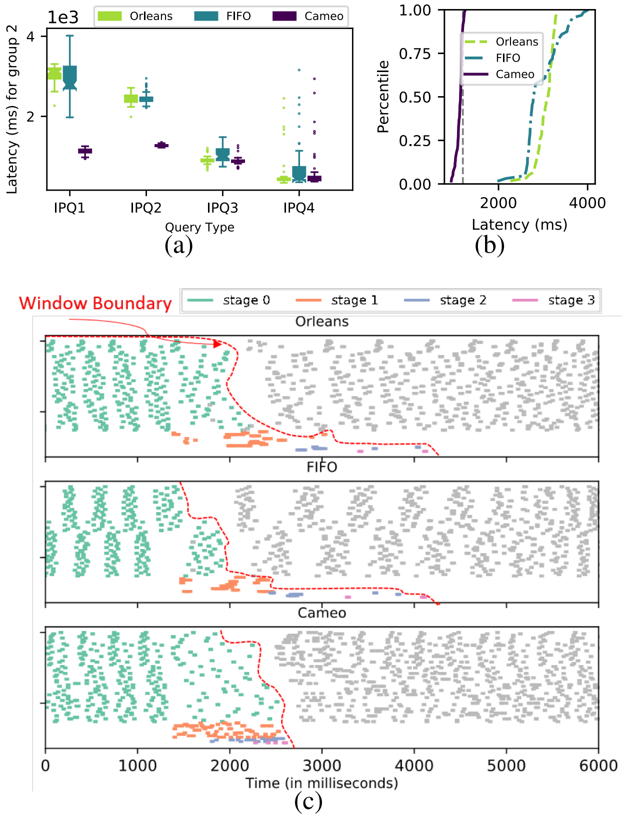


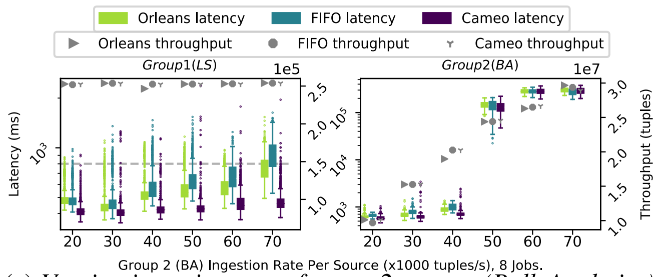
Figure 7: *Single-Tenant Experiments: (a) Query Latency. (b) Latency CDF. (c) Operator Schedule Timeline: X axis = time when operator was scheduled. Y axis = operator ID color coded by operator's stage. Operators are triggered at each stage in order (stage 0 to 3). Job latency is time from all events that belong to the previous window being received at stage 0, until last message is output at stage 3.*

6.1 Single-tenant Scenario

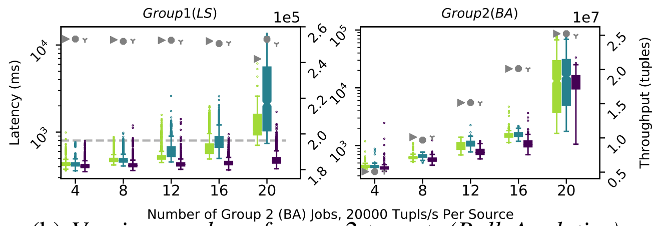
In Figure 7 we evaluate a single-tenant setting with 4 queries: IPQ1 through IPQ4. IPQ1 and IPQ3 are periodic and they respectively calculate sum of revenue generated by real time ads, and the number of events generated by jobs groups by different criteria. IPQ2 performs similar aggregation operations as IPQ1 but on a sliding window (i.e., consecutive window contains overlapped input). IPQ4 summarizes errors from log events via running a windowed join of two event stream, followed by aggregation on a tumbling window (i.e., where consecutive windows contain non-overlapping ranges of data that are evenly spaced across time).

From Figure 7(a) we observe that Cameo improves median latency by up to 2.7×and tail latency by up to 3.2×.We also observe that default Orleans performs almost as well as Cameo for IPQ4. This is because IPQ4 has a higher execution time with heavy memory access, and performs well when pinned to a single thread with better access locality.

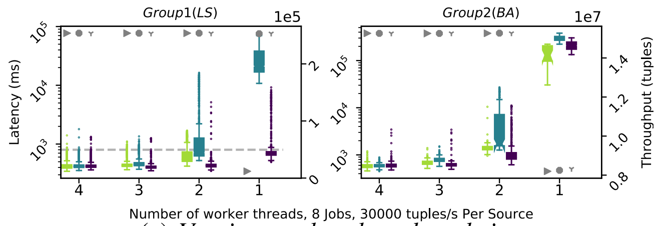
Effect on intra-query operator scheduling. The CDF in



(a) *Varying ingestion rate of group 2 tenants (Bulk Analytics).*



(b) *Varying number of group 2 tenants (Bulk Analytics).*



(c) *Varying worker thread pool size.*

Figure 8: *Latency-sensitive jobs under competing workloads.*

Figure 7(b) shows that Orleans' latency is about 3× higher than Cameo. While FIFO has a slightly lower median latency, its tail latency is as high as in Orleans.

Cameo's prioritization is especially evident in Figure 7(c), where dots are message starts, and red lines separate windows. We first observe that Cameo is faster, and it creates a clearer boundary between windows. Second, messages that contribute to the first result (colored dots) and messages that contribute to the second result (grey dots) do not overlap on the timeline. For the other two strategies, there is a *drift* between stream progress in early stages vs. later stages, producing a prolonged delay. In particular, in Orleans and FIFO, early-arriving messages from the next window are executed *before* messages from the first window, thus missing deadlines.

6.2 Multi-tenant Scenario

Figure 8 studies a control group of latency-constrained dataflows (group 1 LS jobs) by fixing both job count and data ingestion rate. We vary data volume from competing workloads (group 2 BA jobs) and available resources. For LS jobs we impose a latency target of 800 ms, while for BA jobs we use a 7200s latency constraint.

Cameo under increasing data volume. We run four group 1 jobs alongside group 2 jobs. We increase the competing group 2 jobs' traffic, by increasing the ingestion speed (Figure 8(a)) and number of tenants (Figure 8(b)). We observe that all three strategies (Cameo, Orleans, FIFO) are comparable up to per-source tuple rate of 30K/s in Figure 8(a), and up to twelve group 2 jobs in Figure 8(b). Beyond this, overloading causes massive latency degradation, for group 1 (LS) jobs at median and 99 percentile latency (respectively): i) Orleans is worse than Cameo by up to 1.6 and 1.5× in Figure 8(a), up to 2.2 and 2.8× in Figure 8(b), and ii) FIFO is worse than Cameo by up to 2 and 1.8× in Figure 8(a), up to 4.6 and 13.6× in Figure 8(b). Cameo stays stable. Cameo's degradation of group 2 jobs is small— with latency similar or lower than Orleans and FIFO, and Cameo's throughput only 2.5% lower.

Effect of limited resources. Orleans' [74] underlying SEDA architecture [86] resizes thread pools to achieve resource balance between execution steps, for dynamic re-provisioning. Figure 8(c) shows latency and throughput when we decrease the number of worker threads. Cameo maintains the performance of group 1 jobs except in the most restrictive 1 thread case (although it still meets 90% of deadlines). Cameo prefers messages with impending deadlines and this causes back-pressure for jobs with less-restrictive latency constraints, lowering throughput. Both Orleans and FIFO observe large performance penalties for group 1 and 2 jobs (higher in the former). Group 2 jobs with much higher ingestion rate will naturally receive more resources upon message arrivals, leading to back-pressure and lower throughput for group 1 jobs.

Effect of temporal variation of workload. We use a Pareto distribution for data volume in Figure 9, with four group 1 jobs and eight group 2 jobs. (This is based on Figures 2(a), 2(c), which showed a Power-Law-like distribution.) The cluster utilization is kept under 50%.

High ingestion rate can suddenly prolong queues at machines. Visualizing timelines in Figures 9(a), 9(b), and 9(c) shows that for latency-constrained jobs (group 1), Cameo's latency is more stable than Orleans' and FIFO's. Figure 9(d) shows that Cameo reduces (median, 99th percentile) latency by (3.9×, 29.7×) vs. Orleans, and (1.3×, 21.1×) vs. FIFO. Cameo's standard deviation is also lower by 23.2× and 12.7× compared to Orleans and FIFO respectively. For group 2, Cameo produces smaller average latency and is less affected by ingestion spikes. Transient workload bursts affect many jobs, e.g., all jobs around $t = 400$ with FIFO, as a spike at one operator affects all its collocated operators.

Ingestion pattern from production trace. Production workloads exhibit high degree of skew across data sources. In Figure 10 we show latency distribution of dataflows consuming two workload distributions derived from Figure 2(c): Type 1 and 2. Type 1 produces twice as many events as Type 2. However, Type 2 is heavily skewed and its ingestion rate varies by 200× across sources. This heavily impacts operators that are collocated. The success rate (i.e., the fraction of outputs that meet their deadline) is only 0.2% and 1.5% for Orleans and 7.9% and 9.5% for FIFO. Cameo prioritizes critical messages, maintaining success rates of 21.3% and 45.5% respectively.

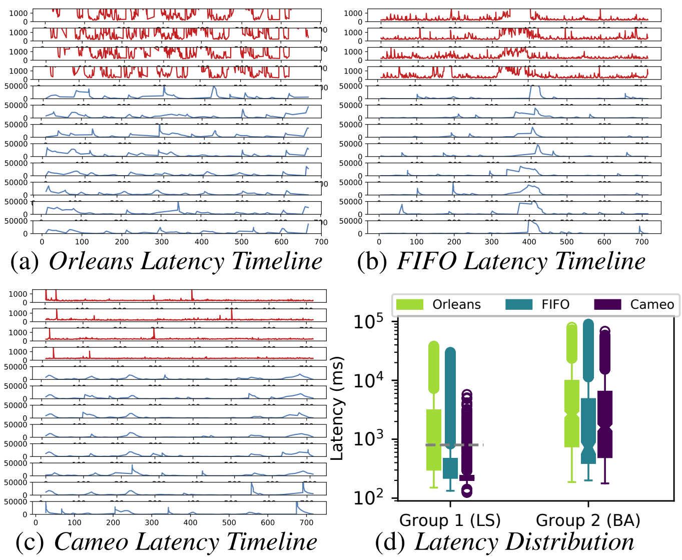

(a) *Orleans Latency Timeline* (b) *FIFO Latency Timeline* (c) *Cameo Latency Timeline* (d) *Latency Distribution*

Figure 9: *Latency under Pareto event arrival.*

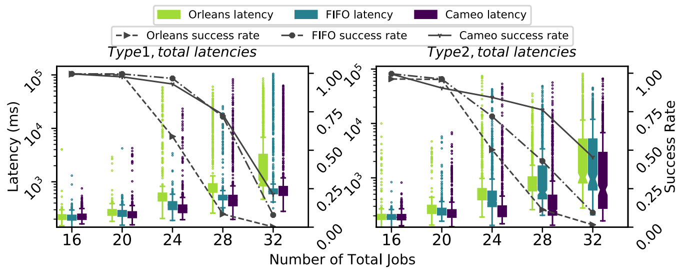


Figure 10: *Spatial Workload Variation.*

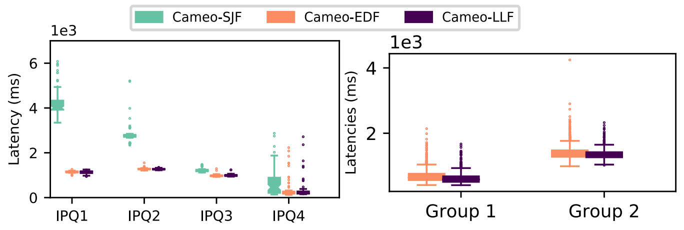


Figure 11: *Cameo Policies. Left: Single query latency distribution. Right: Multi-Query Latency Distribution.*

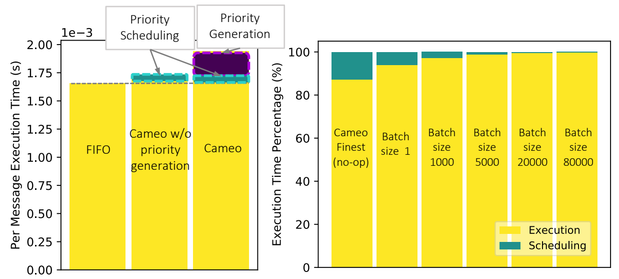


Figure 12: *Cameo Scheduling Overhead.*

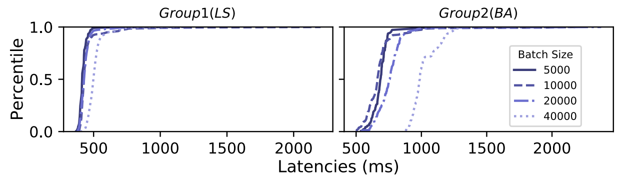


Figure 13: *Effect of Batch Size.*

6.3 Cameo: Internal Evaluation

We next evaluate Cameo's internal properties.

LLF vs. EDF vs. SJF. We implement three scheduling policies using the Cameo context API and evaluate using Section 6.1's workload. The three scheduling policies are: Least Laxity First (LLF, our default), Earliest Deadline First (EDF), and Shortest Job First (SJF). Figure 11 shows that SJF is consistently worse than LLF and EDF (with the exception of query IPQ4– due to the lack of queuing effect under lighter workload). Second, EDF and LLF perform comparably.

In fact we observed that EDF and LLF produced similar schedules for most of our queries. This is because: i) our operator execution time is consistent within a stage, and ii) operator execution time is $\ll$ window size. Thus, excluding operator cost (EDF) does not change schedule by much.

Scheduling Overhead. To evaluate Cameo with many small messages, we use one thread to run a no-op workload (300-350 tenants, 1 msg/s/tenant, same latency needs). Tenants are increased to saturate throughput.

Figure 12 (left) shows breakdown of execution time (inverse of throughput) for three scheduling schemes: FIFO, Cameo without priority generation (overhead only from priority scheduling), and Cameo with priority generation and the LLF policy from Section 4 (overhead from both priority scheduling and priority generation). Cameo's scheduling overhead is $<15\%$ of processing time in the worst case, comprising of 4% overhead from priority-based scheduling and 11% from priority generation.

In practice, Cameo encloses a columnar batch of data in each message like Trill [30]. Cameo's overhead is small compared to message execution costs. In Figure 12 (right), under Section 6's workload, scheduling overhead is only 6.4% of execution time for a local aggregation operator with batch size 1. Overhead falls with batch size. When Cameo is used as a generalized scheduler and message execution costs are small (e.g., with <1 ms), we recommend tuning scheduling quantum and message size to reduce scheduling overhead.

In Figure 13, we batch more tuples into a message, while maintaining same overall tuple ingestion rate. In spite of decreased flexibility available to the scheduler, group 1 jobs' latency is unaffected up to 20K batch size. It degrades at higher batch size (40K), due to more lower priority tuples blocking higher priority tuples. Larger messages hide scheduling overhead, but could starve some high priority messages.

Varying Scope of Scheduler Knowledge. If Cameo is unaware of query semantics (but aware of DAG and latency constraints), Cameo conservatively estimates t_{M_F} without dead-

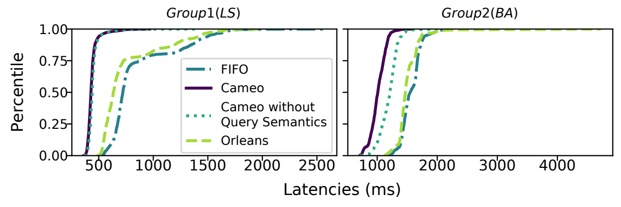


Figure 14: *Benefit of Query Semantics-awareness in Cameo.*

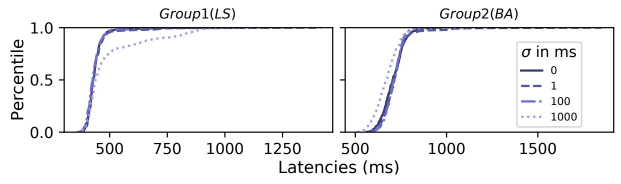


Figure 15: *Profiling Inaccuracy. Standard deviation in ms.*

line extension for window operators, causing a tighter ddl_M. Figure 14 shows that Cameo performs slightly worse without query semantics (19% increase in group 2 median latency). Against baselines, Cameo still reduces group 1 and group 2's median latency by up to 38% and 22% respectively. Hence, even without query semantic knowledge, Cameo still outperforms Orleans and FIFO.

Effect of Measurement Inaccuracies. To evaluate how Cameo reacts to inaccurate monitoring profiles, we perturb measured profile costs (C_{O_M} from Equation 3) by a normal distribution (μ=0), varying standard deviation (σ) from 0 to 1 s. Figure 15 shows that when σ of perturbation is close to window size (1 s), latency is: i) stable at the median, and ii) modestly increases at tail, e.g., only by 55.5% at the 90th percentile. Overall, Cameo's performance is robust when standard deviation is $\leq$ 100ms, i.e., when measurement error is reasonably smaller than output granularity.

7 Related Work

Streaming system schedulers. The first generation of Data Stream Management Systems (DSMS) [15, 32], such as Aurora [28], Medusa [23] and Borealis [14], use QoS based control mechanisms with load shedding to improve query performance at run time. These are either centralized (single-threaded) [28], or distributed [14, 23] but do not handle timestamp-based priorities for partitioned operators. TelegraphCQ [31] orders input tuples before query processing [21, 79], while Cameo addresses operator scheduling within and across query boundaries. Stanford's STREAM [71] uses chain scheduling [22] to minimize memory footprints and optimize query queues, but assumes all queries and scheduler are execute in a single-thread. More recent works in streaming engines propose operator scheduling algorithms for query throughput [20] and latency [41, 64]. Reactive and operator-based policies include [20, 64], while [41] assumes arrivals are periodic or Poisson—however, these works do not build a framework (like Cameo), nor do they handle per-event semantic awareness for stream progress.

Modern stream processing engines such as Spark Streaming [90], Flink [27], Heron [57], MillWheel [18], Naiad [72], Muppet [59], Yahoo S4 [73]) do not include *native* support for multi-tenant SLA optimization. These systems also rely on coarse-grained resource sharing [13] or third-party resource management systems such as YARN [84] and Mesos [47].

Streaming query reconfiguration. Online reconfiguration has been studied extensively [48]. Apart from Figure 3, prior work addresses operator placement [39, 76], load balancing [56, 66], state management [29], policies for scale-in and scale-out [44–46, 67]. Among these are techniques to address latency requirements of dataflow jobs [44, 67], and ways to improve vertical and horizontal elasticity of dataflow jobs in containers [87]. The performance model in [60] focuses on dataflow jobs with latency constraints, while we focus on interactions among operators. Online elasticity was targeted by System S [40, 80], StreamCloud [42] and TimeStream [78]. Others include [35, 53]. Neptune [38] is a proactive scheduler to suspend low-priority batch tasks in the presence of streaming tasks. Yet, there is no operator prioritization *within* each application. Edgewise [37] is a queue-based scheduler based on operator load but not query semantics. All these works are orthogonal to, and can be treated as pluggables in, Cameo.

Event-driven architecture for real-time data processing. This area has been popularized by the resource efficiency of serverless architectures [4, 5, 7]. Yet, recent proposals [17, 26, 58, 70] for stream processing atop event-based frameworks do not support performance targets for streaming queries.

8 Conclusion

We proposed Cameo, a fine-grained scheduling framework for distributed stream processing. To realize flexible per-message scheduling, we implemented a stateless scheduler, contexts that carry important static and dynamic information, and mechanisms to derive laxity-based priority from contexts. Our experiments with real workloads, and on Microsoft Azure, showed that Cameo achieves $2.7\times - 4.6\times$ lower latency than competing strategies and incurs overhead less than 6.4%.

Acknowledgements

We thank our shepherd Matei Zaharia and our anonymous referees for their reviews and help with improving the paper. We thank Kai Zeng for providing feedbacks for initial ideas. This work was supported in part by the following grants: NSF IIS 1909577, NSF CNS 1908888, NSF CNS 1319527, NSF CNS 1838733, a Facebook faculty research award, the Helios project at Microsoft [77], and another generous gift from Microsoft. Shivaram Venktaraman is also supported by the Office of the Vice Chancellor for Research and Graduate Education with funding from the Wisconsin Alumni Research Foundation. We are grateful to the Cosmos, Azure Data Lake, and PlayFab teams at Microsoft.

References

[1] Akka. https://akka.io/.

[2] Apache Hadoop. https://hadoop.apache.org/.

[3] Apache Kafka Core Concepts. https://kafka.apache.org/11/documentation/streams/core-concepts.

[4] AWS Lambda. https://aws.amazon.com/lambda/.

[5] Azure Functions. https://azure.microsoft.com/en-us/services/functions/.

[6] Flink Time Attribute. https://ci.apache.org/projects/flink/flink-docs-release-1.7/dev/event_time.html.

[7] Google Cloud Functions. https://cloud.google.com/functions.

[8] Kubernetes: Production-Grade Container Orchestration. https://kubernetes.io/.

[9] .NET ConcurrentBag. https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentbag-1?view=netframework-4.8.

[10] Orleans. https://dotnet.github.io/orleans/.

[11] Serverless Streaming Architectures and Best Practices, amazon web services. https://d1.awsstatic.com/whitepapers/Serverless_Streaming_Architecture_Best_Practices.pdf.

[12] Sizes for Windows virtual machines in Azure. https://docs.microsoft.com/en-us/azure/virtual-machines/windows/sizes.

[13] Storm Multitenant Scheduler. https://storm.apache.org/releases/current/javadocs/org/apache/storm/scheduler/multitenant/package-summary.html.

[14] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. The design of the Borealis stream processing engine. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, volume 5, pages 277–289, 2005.

[15] Daniel J Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *Proceedings of the VLDB Endowment*, 12(2):120–139, 2003.

[16] Lior Abraham, John Allen, Oleksandr Barykin, Vinayak Borkar, Bhuwan Chopra, Ciprian Gerea, Daniel Merl, Josh Metzler, David Reiss, Subbu Subramanian, et al. Scuba: diving into data at facebook. *Proceedings of the VLDB Endowment*, 6(11):1057–1067, 2013.

[17] Adil Akhter, Marios Fragkoulis, and Asterios Katsifodimos. Stateful functions as a service in action. *Proceedings of the VLDB Endowment*, 12(12):1890–1893, 2019.

[18] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013.

[19] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8:1792–1803, 2015.

[20] Lisa Amini, Navendu Jain, Anshul Sehgal, Jeremy Silber, and Olivier Verscheure. Adaptive control of extreme-scale stream processing systems. In *Proceedings of the IEEE 26th International Conference on Distributed Computing Systems (ICDCS)*, pages 71–71. IEEE, 2006.

[21] Ron Avnur and Joseph M Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 261–272, 2000.

[22] Brian Babcock, Shivnath Babu, Rajeev Motwani, and Mayur Datar. Chain: Operator scheduling for memory minimization in data stream systems. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 253–264. ACM, 2003.

[23] Magdalena Balazinska, Hari Balakrishnan, and Michael Stonebraker. Load management and high availability in the medusa distributed stream processing system. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 929–930. ACM, 2004.

[24] Philip A Bernstein, Todd Porter, Rahul Potharaju, Alejandro Z Tomsic, Shivaram Venkataraman, and Wentao Wu. Serverless event-stream processing over virtual actors. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2019.

[25] Sergey Bykov, Alan Geller, Gabriel Kliot, James R Larus, Ravi Pandya, and Jorgen Thelin. Orleans: Cloud computing for everyone. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, page 16. ACM, 2011.

[26] Paris Carbone, Marios Fragkoulis, Vasiliki Kalavri, and Asterios Katsifodimos. Beyond Analytics: the Evolution of Stream Processing Systems. In *Proceedings of the 2020 ACM SIGMOD international conference on Management of data*, 2020.

[27] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[28] Don Carney, Uğur Çetintemel, Alex Rasin, Stan Zdonik, Mitch Cherniack, and Mike Stonebraker. Operator scheduling in a data stream manager. In *Proceedings of the VLDB Endowment*, pages 838–849. VLDB Endowment, 2003.

[29] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*, pages 725–736. ACM, 2013.

[30] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C Platt, James F Terwilliger, and John Wernsing. Trill: A high-performance incremental query processor for diverse analytics. *Proceedings of the VLDB Endowment*, 8(4):401–412, 2014.

[31] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J Franklin, Joseph M Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R Madden, Fred Reiss, and Mehul A Shah. Telegraphcq: continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 668–668. ACM, 2003.

[32] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Cetintemel, Ying Xing, and Stanley B Zdonik. Scalable distributed stream processing. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, volume 3, pages 257–268, 2003.

[33] Edward G Coffman, Jr, Michael R Garey, and David S Johnson. An application of bin-packing to multiprocessor scheduling. *SIAM Journal on Computing*, 7(1):1–17, 1978.

[34] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. Dhalion: Self-regulating stream processing in heron. *Proceedings of the VLDB Endowment*, August 2017.

[35] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. Dhalion: self-regulating stream processing in Heron. *Proceedings of the VLDB Endowment*, 10(12):1825–1836, 2017.

[36] Tom ZJ Fu, Jianbing Ding, Richard TB Ma, Marianne Winslett, Yin Yang, and Zhenjie Zhang. DRS: dynamic resource scheduling for real-time analytics over fast streams. In *Proceedings of the 35th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 411–420. IEEE, 2015.

[37] Xinwei Fu, Talha Ghaffar, James C Davis, and Dongyoon Lee. Edgewise: a better stream processing engine for the edge. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 929–946, 2019.

[38] Panagiotis Garefalakis, Konstantinos Karanasos, and Peter Pietzuch. Neptune: Scheduling suspendable tasks for unified stream/batch applications. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, pages 233–245, 2019.

[39] Panagiotis Garefalakis, Konstantinos Karanasos, Peter R Pietzuch, Arun Suresh, and Sriram Rao. Medea: scheduling of long running applications in shared production clusters. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys)*, pages 4–1, 2018.

[40] Buğra Gedik, Scott Schneider, Martin Hirzel, and Kun-Lung Wu. Elastic scaling for data stream processing. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1447–1463, 2014.

[41] Yu Gu, Ge Yu, and Chuanwen Li. Deadline-aware complex event processing models over distributed monitoring streams. *Mathematical and Computer Modelling*, 55(3-4):901–917, 2012.

[42] Vincenzo Gulisano, Ricardo Jimenez-Peris, Marta Patino-Martinez, Claudio Soriente, and Patrick Valduriez. Streamcloud: An elastic and scalable data streaming system. *IEEE Transactions on Parallel and Distributed Systems*, 23(12):2351–2365, 2012.

[43] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2-3):202–220, 2009.

[44] Thomas Heinze, Zbigniew Jerzak, Gregor Hackenbroich, and Christof Fetzer. Latency-aware elastic scaling for distributed data stream processing systems. In *Proceedings of the 8th ACM International Conference on*

Distributed Event-Based Systems, pages 13–22. ACM, 2014.

[45] Thomas Heinze, Valerio Pappalardo, Zbigniew Jerzak, and Christof Fetzer. Auto-scaling techniques for elastic data stream processing. In *Proceedings of the IEEE Data Engineering Workshops (ICDEW)*, pages 296–302. IEEE, 2014.

[46] Thomas Heinze, Lars Roediger, Andreas Meister, Yuanzhen Ji, Zbigniew Jerzak, and Christof Fetzer. Online parameter optimization for elastic data stream processing. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, pages 276–287. ACM, 2015.

[47] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, 2011.

[48] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. A catalog of stream processing optimizations. *ACM Computing Surveys (CSUR)*, 46(4):46, 2014.

[49] Moritz Hoffmann, Andrea Lattuada, John Liagouris, Vasiliki Kalavri, Desislava Dimitrova, Sebastian Wicki, Zaheer Chothia, and Timothy Roscoe. Snailtrail: Generalizing critical paths for online analysis of distributed dataflows. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 95–110, 2018.

[50] Moritz Hoffmann, Andrea Lattuada, and Frank McSherry. Megaphone: latency-conscious state migration for distributed streaming dataflows. *Proceedings of the VLDB Endowment*, 12(9):1002–1015, 2019.

[51] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. Cloud programming simplified: A Berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019.

[52] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 783–798, 2018.

[53] Faria Kalim, Le Xu, Sharanya Bathey, Richa Meherwal, and Indranil Gupta. Henge: Intent-driven multi-tenant stream processing. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, pages 249–262. ACM, 2018.

[54] Evangelia Kalyvianaki, Themistoklis Charalambous, Marco Fiscato, and Peter Pietzuch. Overload management in data stream processing systems with latency guarantees. In *Proceedings of the 7th IEEE International Workshop on Feedback Computing*, 2012.

[55] Evangelia Kalyvianaki, Marco Fiscato, Theodoros Salonidis, and Peter Pietzuch. Themis: Fairness in federated stream processing under overload. In *Proceedings of the 2016 International Conference on Management of Data*, pages 541–553. ACM, 2016.

[56] Wilhelm Kleiminger, Evangelia Kalyvianaki, and Peter Pietzuch. Balancing load in stream processing with the cloud. In *Proceedings of the 27th IEEE International Conference on Data Engineering Workshops*, pages 16–21. IEEE, 2011.

[57] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthikeyan Ramasamy, and Siddarth Taneja. Twitter Heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD international conference on Management of data*, 2015.

[58] Avinash Kumar, Zuozhi Wang, Shengquan Ni, and Chen Li. Amber: a debuggable dataflow system based on the actor model. *Proceedings of the VLDB Endowment*, 13(5):740–753, 2020.

[59] Wang Lam, Lu Liu, STS Prasad, Anand Rajaraman, Zoheb Vacheri, and AnHai Doan. Muppet: Mapreduce-style processing of fast data. *Proceedings of the VLDB Endowment*, 5(12):1814–1825, 2012.

[60] Boduo Li, Yanlei Diao, and Prashant Shenoy. Supporting scalable analytics with latency constraints. *Proceedings of the VLDB Endowment*, 8(11):1166–1177, 2015.

[61] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 311–322. ACM, 2005.

[62] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson, and David Maier. Out-of-order processing: a new architecture for high-performance stream systems. *Proceedings of the VLDB Endowment*, 1(1):274–288, 2008.

[63] Na Li and Qiang Guan. Deadline-aware event scheduling for complex event processing systems. In *Proceedings of the International Conference on Intelligent Data Engineering and Automated Learning*, pages 101–109. Springer, 2013.

[64] Xin Li, Zhiping Jia, Li Ma, Ruihua Zhang, and Haiyang Wang. Earliest deadline scheduling for continuous queries over data streams. In *Proceedings of the IEEE International Conference on Embedded Software and Systems*, pages 57–64. IEEE, 2009.

[65] Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[66] Simon Loesing, Martin Hentschel, Tim Kraska, and Donald Kossmann. Stormy: an elastic and highly available streaming service in the cloud. In *Proceedings of the 2012 Joint EDBT/ICDT Workshops*, pages 55–60. ACM, 2012.

[67] Björn Lohrmann, Peter Janacik, and Odej Kao. Elastic stream processing with latency guarantees. In *Proceedings of the 35th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 399–410. IEEE, 2015.

[68] Luo Mai, Kai Zeng, Rahul Potharaju, Le Xu, Steve Suh, Shivaram Venkataraman, Paolo Costa, Terry Kim, Saravanan Muthukrishnan, Vamsi Kuppa, et al. Chi: a scalable and programmable control plane for distributed stream processing systems. *Proceedings of the VLDB Endowment*, 11(10):1303–1316, 2018.

[69] Aloysius Ka-Lau Mok. *Fundamental design problems of distributed systems for the hard-real-time environment*. PhD thesis, Massachusetts Institute of Technology, 1983.

[70] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging AI applications. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577, 2018.

[71] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query processing, resource management, and approximation in a data stream management system. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2003.

[72] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM, 2013.

[73] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed stream computing platform. In *Proceedings of the IEEE Data Mining Workshops (ICDMW)*, pages 170–177. IEEE, 2010.

[74] Andrew Newell, Gabriel Kliot, Ishai Menache, Aditya Gopalan, Soramichi Akiyama, and Mark Silberstein. Optimizing distributed actor systems for dynamic interactive services. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 38. ACM, 2016.

[75] Zhengyu Ou, Ge Yu, Yaxin Yu, Shanshan Wu, Xiaochun Yang, and Qingxu Deng. Tick scheduling: A deadline based optimal task scheduling approach for real-time data stream systems. In *Proceedings of the International Conference on Web-Age Information Management*, pages 725–730. Springer, 2005.

[76] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. Network-aware operator placement for stream-processing systems. In *Proceedings of the 22nd International Conference on Data Engineering, (ICDE)*, pages 49–49. IEEE, 2006.

[77] Rahul Potharaju, Terry Kim, Wentao Wu, Vidip Acharya, Steve Suh, Andrew Fogarty, Apoorve Dave, Sinduja Ramanujam, Tomas Talius, Lev Novik, and Raghu Ramakrishnan. Helios: Hyperscale indexing for the cloud & edge. *Proceedings of the VLDB Endowment*, 13(12), 2020.

[78] Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. Timestream: Reliable stream computation in the cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 1–14. ACM, 2013.

[79] Vijayshankar Raman, Bhaskaran Raman, and Joseph M Hellerstein. Online dynamic reordering for interactive data processing. In *VLDB*, volume 99, pages 709–720, 1999.

[80] Scott Schneider, Henrique Andrade, Bugra Gedik, Alain Biem, and Kun-Lung Wu. Elastic scaling of data parallel operators in stream processing. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12. IEEE, 2009.

[81] John A Stankovic, Marco Spuri, Marco Di Natale, and Giorgio C Buttazzo. Implications of classical scheduling results for real-time systems. *Computer*, 28(6):16–25, 1995.

[82] Andrew S Tanenbaum and Herbert Bos. *Modern operating systems*. Pearson, 2015.

[83] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156, 2014.

[84] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache Hadoop Yarn: Yet another resource negotiator. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, page 5. ACM, 2013.

[85] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J Franklin, Benjamin Recht, and Ion Stoica. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 374–389. ACM, 2017.

[86] Matt Welsh, David E. Culler, and Eric A. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the 18th ACM Symposium on Operating System Principles SOSP*, pages 230–243. ACM, 2001.

[87] Yingjun Wu and Kian-Lee Tan. ChronoStream: Elastic stateful stream computation in the cloud. In *Proceedings of the 31st IEEE International Conference on Data Engineering (ICDE)*, pages 723–734. IEEE, 2015.

[88] Le Xu, Boyang Peng, and Indranil Gupta. Stela: Enabling stream processing systems to scale-in and scale-out on-demand. In *Proceedings of the IEEE International Conference on Cloud Engineering (IC2E)*, pages 22–31. IEEE, 2016.

[89] Da Yu, Yibo Zhu, Behnaz Arzani, Rodrigo Fonseca, Tianrong Zhang, Karl Deng, and Lihua Yuan. Dshark: A general, easy to program and scalable framework for analyzing in-network packet traces. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI'19, page 207–220, USA, 2019.

[90] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, pages 423–438. ACM, 2013.

WHIZ: Data-Driven Analytics Execution

Robert Grandl[†][*]
Google

Arjun Singhvi[†]
University of Wisconsin–Madison

Raajay Viswanathan[*]
Uber Technologies Inc.

Aditya Akella
University of Wisconsin–Madison

***Abstract*—** Today's data analytics frameworks are compute-centric, with analytics execution almost entirely dependent on the predetermined physical structure of the high-level computation. Relegating intermediate data to a second class entity in this manner hurts flexibility, performance, and efficiency. We present WHIZ, a new analytics execution framework that cleanly separates computation from intermediate data. This enables runtime visibility into intermediate data via programmable monitoring, and data-driven computation where data properties drive when/what computation runs. Experiments with a WHIZ prototype on a 50-node cluster using batch, streaming, and graph analytics workloads show that it improves analytics completion times 1.3-2× and cluster efficiency 1.4× compared to state-of-the-art.

1 Introduction

Many important applications in diverse settings rely on analyzing large datasets, including relational tables, event streams, and graph-structured data. To analyze such data, several execution frameworks have been introduced [4, 7, 15, 24, 36, 42–45, 50, 51]. These enable data parallel computation, where an analytics job's logic is run in parallel on data shards spread across cluster machines.

Almost all these frameworks build on the *MapReduce execution engine* [21]. Like MapReduce, they leverage *compute-centric* execution (§2). Their execution engines' focus is on splitting a job's computational logic, and distributing it across *tasks* to be run in parallel. All aspects of the subsequent execution of the job are rooted in the job's computational logic, and its task-level computation distribution. These include the fact that compute logic running inside tasks is static and/or predetermined; intermediate data is partitioned and routed to where it is consumed based on the task-level structure; and dependent tasks are launched when a fraction of upstream tasks they depend on finish. These attributes of job execution are not related to, or driven by, the properties of intermediate data, i.e., how much and what data is generated. Thus, intermediate data is a *second-class citizen*.

[†]These authors contributed equally to this work.
[*]Work done while at University of Wisconsin–Madison.

Compute-centricity was a natural early choice: knowing job structure beforehand simplifies carving containers to execute tasks; compute-centricity provided clean mechanisms to recover from failures – only tasks on a failed machine needed to be re-executed; and job scheduling became simple because of having to deal with static inputs, i.e., fixed tasks/dependency structures.

Unfortunately, today, compute-centricity severely hinders analytics performance and cluster efficiency due to four fundamental issues (§2, §9): (1) Intermediate data-unawareness means there is no way to quickly adapt job execution based on changing run-time data properties (e.g., volume, key distribution, etc.) to ensure performance- and resource-optimal data processing. (2) Likewise, static parallelism and intermediate data partitioning inherent to compute-centricity prevent adaptation to intermediate data skew and resource flux which are difficult to predict ahead of time, yet, common to modern datasets [30] and multi-tenancy. (3) Execution schedules being tied to compute structure can lead to resource waste while tasks wait for input data to become available - an effect that is exacerbated under multi-tenancy. (4) The skew due to compute-based organization of intermediate data can result in storage hotspots and poor cross-job I/O isolation; it also curtails data locality.

We observe that the above limitations arise from (1) tight *coupling* between intermediate data and compute, and (2) *intermediate data agnosticity* in today's execution frameworks. To improve analytics performance, efficiency, isolation, and flexibility, we develop a new execution framework, WHIZ, that eschews compute-centricity, cleanly separates computation from all intermediate data, and treats both intermediate data and compute as equal first-class entities during analytics applications' execution. WHIZ applies equally to batch analytics, streaming and graph processing.

In WHIZ, intermediate data is written to/read from a *logically* separate distributed key-value datastore. The store offers programmable visibility – applications can provide custom routines for monitoring runtime data properties. The store notifies an execution layer when an application's runtime data satisfies predicates based on data properties. Decoupling, monitoring, and predicates enable *data-driven incremental computation*: based on data properties, WHIZ decides on the fly *what logic* to launch in order to further process the data generated, *how many parallel tasks* to launch, *when/where to launch* them, and *what resources* to allocate to tasks.

We make the following contributions in designing WHIZ: (1) We present a scalable approach for programmable intermediate data monitoring which forms the basis for data-driven actions. (2) We show how to organize intermediate data from multiple jobs in the datastore so as to achieve data locality, fault tolerance, and cross-job isolation. Since obtaining an optimal data organization is intractable, we develop novel heuristics that carefully trade-off among these objectives. (3) We build an execution layer that incrementally decides all aspects of the job execution based on data property predicates being satisfied. We develop novel iterative heuristics for the execution layer to decide, for each ready-to-run analytics stage, task parallelism, task placement, and task sizing. This minimizes runtime skew in the data processed, lowers data shuffle cost and ensures optimal efficiency under resource dynamics. The execution layer also decides the optimal per-task logic to use at run-time.

We build a WHIZ prototype using Tez [47] and YARN [52] (15K LOC). We conduct experiments on a 50 machine cluster in CloudLab [6]. We compare against several state-of-the-art compute-centric (CC) batch, stream and graph processing approaches. Using data-driven incremental computation, WHIZ improves median (95%-ile) job completion time (JCT) $1.3-1.6\times$ $(1.5-2.2\times)$ and cluster efficiency $1.4\times$ by launching the right number of appropriately-sized tasks only when predicates are met. We observe up to $2.8\times$ improvement in efficiency due to WHIZ's ability to change processing logic on the fly. Furthermore, we observe that the impact on JCT under failures is minimal due to WHIZ's data organization. We observe that WHIZ's gains relative to CC improve with contention due to data-driven execution and better data management which mitigate I/O hotspots and minimize resource wastage.

2 Compute-Centric vs. Data-Driven

We begin with an overview of existing data analytics frameworks (§2.1). We then discuss the key design principles of WHIZ (§2.2). Finally, we list the performance issues arising from *compute-centricity* and show how the *data-driven* design adopted by WHIZ overcomes them (§2.3).

2.1 Today: Compute-Centric Engines

Frameworks for batch, graph and streaming analytics rely on execution engines [2, 57]; the engine can be an internal component of the framework or a stand-alone one that the system leverages. The engine is responsible for orchestrating the execution of the analytics job across the cluster.

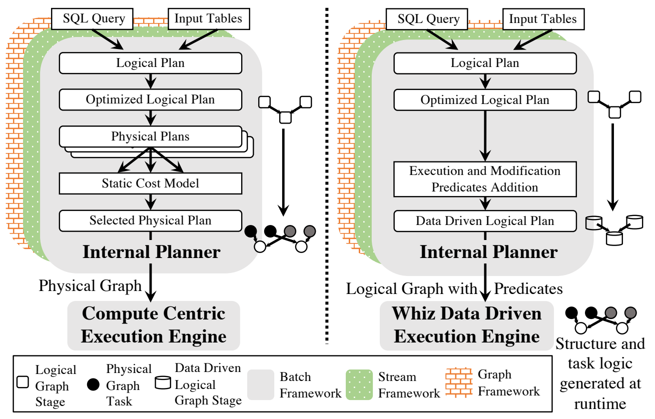


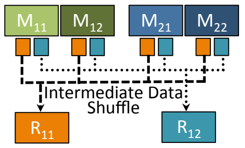
Figure 1: Job Execution Pipelines: Today frameworks hand over physical graphs to the underlying CC execution engine. With WHIZ, *the framework instead hands down a data-driven logical graph and* WHIZ *decides the physical graph at runtime.*

Users submit their jobs to these frameworks (Figure 1) via high-level interfaces (e.g., SQL-like query in case of batch analytics). On submission, the high-level job is handed over to the *internal planner* of the framework which decides the execution plan of the job (expressed in the form of a directed graph). Specifically, the high-level job is translated to a *logical graph* in which different vertices represent different compute *stages* of the overall job and edges represent the dependencies. The logical graph may optionally undergo further optimizations (e.g., to decide the execution order of the stages) and is finally converted to a *physical graph* by undergoing physical optimizations during which low-level execution details such as number of *tasks* per stage (parallelism), dependencies between tasks, resource needs and exact task processing logic are decided.

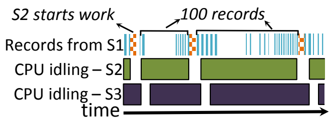
The execution engine takes the physical graph and orchestrates its execution starting with root stages' tasks processing input data to generate *intermediate data*, which is consumed by downstream stages' tasks.

We explain how the execution engine orchestrates the physical graph and its interplay with intermediate data for different analytics. Figure 2a is an example of a simple *batch analytics* job. Here, two tables need to be filtered based on provided predicates and joined to produce a new table. There are 3 stages: two maps for filtering and one reduce to perform the join. Execution proceeds as follows: (1) Map tasks from both the stages execute first with each task processing a partition of the corresponding input table. (2) Map intermediate results are written to local disk by each task, split into files, one per consumer reduce task. (3) Reduce tasks are launched when the map stages are nearing completion; each reduce task *shuffles* relevant intermediate data from all map tasks' locations, and generates output.

A *stream analytics* job (e.g., Figure 2b) has a similar model [11,37,58]; the main difference is that tasks in all stages are always running. A *graph analytics job*, in a framework



(a) A batch analytics job. Intermediate data is partitioned into two key ranges, one per reduce task, and stored in local files at map tasks.



(b) Data flow in a streaming job. Tasks in all stages are always running. Output of a stage is immediately passed to a task in downstream stage. However, CPU is idle until task in Stage 2 receives 100 records after which computation is triggered.

Figure 2: Simplified examples of existing analytics systems.

that relies on the popular message passing abstraction [42], has a similar but simplified model: the different stages are iterations in a graph algorithm, and thus all stages execute the same processing logic (with the input being the output of the previous iteration).

Compute-centricity: Today's execution engines *early-bind* to a physical graph at job launch-time. Their primary goal is to split up and distribute computation across machines. The composition of this distributed computation, in terms of physical tasks and their dependencies, is a first class entity. The exact computation in each task is assumed to be known beforehand. The way in which intermediate data is partitioned and routed to consumer tasks, and when and how dependent computation is launched, are tied to compute structure. We use the term *compute-centric* to refer to this design pattern. Here, intermediate data is a second class entity as important aspects of job execution such as parallelism, processing and scheduling logic are decided without taking it into account (§2.3).

2.2 Whiz: A Data-Driven Framework

Whiz is an *execution engine* that makes intermediate data a first class citizen and supports diverse analytics. Whiz adopts the following design principles:

1. Decoupling compute and data: Whiz decouples compute from intermediate data, and the data from all stages across all jobs is written/read to/from a logically separate key-value (KV) datastore (§4), i.e, the datastore resides across the same set of machines on which computations take place. The store is managed by a distinct data management layer called the data service (DS). Similarly, an execution service (ES) manages compute tasks.

2. Programmable data visibility: The above separation enables low-overhead and scalable approaches to gain visibility into *all* runtime data (§5.1). Whiz DS allows gathering custom runtime properties of intermediate data, via narrow, well-defined APIs.

3. Runtime physical graph generation: During the job execution pipeline, Whiz skips physical optimization (Figure 1) and thus, *does not early-bind* to a physical graph. Instead, the framework's internal planner performs *data-driven embellishment* on the logical graph to give a *data-driven logical graph*. This embellishment adds predicates to decide when and what logic should be used to process data of each stage, and gives Whiz the ability to incrementally generate the physical graph at runtime (§6).

4. Data-driven computation: Building on data visibility and data-driven logical graphs, Whiz initiates data-driven computation by notifying applications when intermediate data predicates within each stage are satisfied (§5.2). Data properties drive all further aspects of computation: task logic, parallelism and sizing (§6).

2.3 Overcoming Compute-centricity Issues

We contrast Whiz with compute-centricity along flexibility, performance, efficiency, placement, and isolation.

Data opacity, and compute rigidity: In compute-centric frameworks, there is no visibility into intermediate data of a job and the tasks' computational logic are decided a priori. This prevents adapting the tasks' logic based on their input data. Consider the job in Figure 2a. Existing frameworks determine the type of join for the *entire* reduce stage based on coarse statistics [3]; unless one of the tables is small, a sort-merge join is employed to avoid out-of-memory (OOM) errors. On the other hand, having fine-grained visibility into input data for each task enables dynamically determining the type of join to use for *different* reduce tasks. A task can use hash join if the total size of *its* input is less than the available memory, and merge join otherwise. Whiz enables deciding the logic at runtime through its ability to provide visibility and incrementally generate the physical graph (§6.1).

Static Parallelism, Partitioning: Today, jobs' per-stage parallelism, inter-task edges and intermediate data partitioning strategy are decided independent of runtime data and resource dynamics. In Spark [57] the number of tasks in a stage is determined a priori by the user application or by SparkSQL [10]. A hash partitioner is used to place an intermediate (k, v) pair into one of $|tasks|$ buckets. Pregel [42] vertex-partitions the input graph; partitions do not change during the execution of the algorithm.

This limits adaptation to *resource flux* and *data skew*. A running stage cannot utilize newly available compute resources [26, 27, 41] and dynamically increase its parallelism. If some key in a partition has an abnormally large number of records to process, then the corresponding task is significantly slowed down [14], affecting both stage and overall job completion times.

By not early-binding, Whiz can decide task parallelism and task size based on resources available and data volume. This controls data skew, and provisions task resources proportional to the data to be processed (§6.2).

Idling due to compute-driven scheduling: Modern schedulers [23, 52] decide when to launch tasks for a stage based on the static computation structure. When a stage's computation is commutative+associative, schedulers launch its tasks once 90% of all tasks in upstream stages complete [5]. But the remaining 10% of producers can take long to complete [14], resulting in tasks idling.

Idling is worse in streaming, where consumer tasks are continuously waiting for data from upstream tasks. E.g., consider the streaming job in Figure 2b. Stage 2 computes and outputs the median for every 100 records received. Between computation, S2's tasks stay idle. As a result, the tasks in the downstream S3 stage *also* stay idle. To avoid idling, tasks should be scheduled *only when, and only as long as, relevant input is available*. In our example, computation should be launched only after $\geq$ 100 records have been generated by an S1 task. Likewise, in batch analytics, if computation is commutative+associate, it is beneficial to "eagerly" launch tasks to process intermediate data whenever enough data has been generated to process in one batch, and exit soon after it's done.

Idling is easily avoided with WHIZ as it does data-driven scheduling: launches tasks only when predicates are met, i.e, relevant data has been generated (§5.2).

Placement, and storage isolation: Because intermediate data is spread across producer tasks' locations, it is impossible to place *consumer tasks* in a data-local fashion. Such tasks are placed at random [21] and forced to engage in expensive shuffles that consume a significant portion of job runtimes ($\sim$30% [20]).

Also, when tasks from multiple jobs are collocated, it becomes difficult to isolate their hard-to-predict intermediate data I/O. Tasks from jobs generating large intermediate data may occupy much more local storage and I/O bandwidth than those generating less.

Since the WHIZ store manages data from all jobs, it can enforce policies to organize data to meet *per-job* objectives, e.g., data locality for *any* stage (not just input-reading stages), and to meet *cluster* objectives, such as I/O hotspot avoidance and cross-job isolation (§4).

3 WHIZ Overview

We now describe the end-to-end control flow in WHIZ. The end-user submits the job through a high-level interface exposed by the application-specific framework. The framework's internal planner converts the job into a data-driven logical graph through *data-driven embellishment* during which each stage in the graph is annotated with *execution predicates* and *modification predicates*.

Execution predicates determine when data generated by the current stage can be consumed by its downstream stages (e.g., start downstream processing when number of records cross a threshold). Modification predicates determine which processing logic should be chosen at runtime (e.g., decide the join algorithm for the task, say, sort-merge join or hash join) based on data properties.

This data-driven logical graph (e.g., directed acyclic graph in case of batch analytics) , that is expressed via WHIZ APIs (Appendix. A), is submitted to WHIZ via a client (Figure 3).

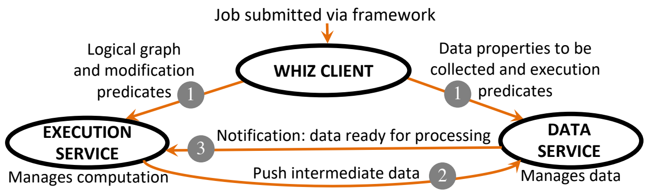

Job submitted via framework
WHIZ CLIENT
Logical graph and modification predicates
1
Data properties to be collected and execution predicates
1
EXECUTION SERVICE
Manages computation
3
Notification: data ready for processing
DATA SERVICE
Manages data
Push intermediate data
2

Figure 3: WHIZ *control flow.*

The WHIZ client is the primary interface between the framework running atop WHIZ and the core WHIZ services - the DS and the ES. The client provides the DS with details regarding data properties to be collected and execution predicates. The client also transfers the logical graph and modification predicates to the ES (step 1).

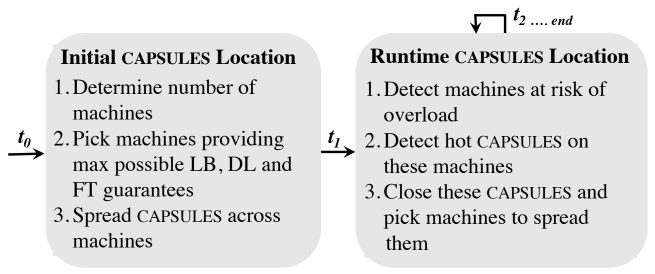
The ES runs the first stage(s) of the logical graph and writes its output to the datastore (step 2). The DS stores the received data and when the execution predicate corresponding to the stage(s) is met, it notifies the ES. The DS piggybacks data statistics (e.g., per-key counts) on this notification to the ES (step 3). On receiving the notification, the ES checks the modification predicates to decide the processing logic and then processes the data. This process repeats. Interactions between the ES, DS and the client are transparent to the framework.

The DS organizes data from all jobs and ensures both per-job and cross-job objectives are met (§4) while simultaneously enabling data visibility through programmable monitoring (§5). The ES, in addition to deciding the processing logic, also determines task parallelism, location and resource use at runtime (§6). In this manner end-users are no longer required to specify low-level details such as task parallelism and data partitioning strategy. In the rest of the paper, we focus on how WHIZ handles data-driven logical graphs and plan to explore designing data-driven embellishers (responsible for embellishment of logical graphs with predicates) that can be added to existing frameworks in the future.

4 Data Store

In WHIZ, all jobs' intermediate data is written to/read from a logically separate datastore (managed by the DS), where it is structured as <key,value> pairs. In batch/stream analytics, the keys are generated by the stage computation logic itself; in graph analytics, keys are identifiers of vertices to which messages (values) are destined for processing in the next iteration. The DS via a cluster-wide master **DS-M** organizes data in the store.

An ideal data organization should achieve three goals: (1) *load balance* and spread all jobs' data, specifically, avoid hotspots and *improve cross-job isolation*, and *minimize within-job skew* in tasks' data processing. (2) maximize job *data locality* by co-locating as much data having the same key as possible. (3) be *fault tolerant* - when a storage node fails, recovery should have minimal impact on job runtime. Our data storage granularity, described next, forms the basis for meeting our goals.



t_2 .... end
Initial CAPSULES Location
1. Determine number of machines
2. Pick machines providing max possible LB, DL and FT guarantees
3. Spread CAPSULES across machines
t_0
t_1
Runtime CAPSULES Location
1. Detect machines at risk of overload
2. Detect hot CAPSULES on these machines
3. Close these CAPSULES and pick machines to spread them

Figure 4: Data organization flow when a stage starts generating data.

4.1 Capsule: A Unit of Data in WHIZ

WHIZ groups intermediate data based on keys into groups called *capsules*. A stage's intermediate data is organized into some large number N capsules; crucially N is *late-bound* as described below, which helps meet our goals above. Intermediate data key range is split N-ways, and each capsule stores all <k, v> data from a given range. WHIZ strives to materialize all capsule data on one machine; rarely (e.g., when there is less space left on a machine), a capsule may be spread across a small number of machines. This materialization property of capsules forms the basis for consumer task data locality.

Furthermore, WHIZ capsule key ranges are *late-bound*: we first determine the set of machines on which capsules from a stage are to be stored; machines are chosen to maximally support isolation, load balance, locality and fault tolerance; the choice of machines then determines the number N for a stage's capsules (§4.2).

Given these machines and N, as the stage produces data at runtime, N capsules are materialized, and dynamically allocated to right-sized tasks; this enables the ES to preserve data-local processing, lower skew, and optimally use compute resources (§6).

4.2 Fast Capsule Allocation

We consider how to place multiple jobs' capsules on machines to avoid hotspots, ensure data locality and minimize job runtime impact on data loss. We formulate an ILP to this end (see Table 7 in Appendix. B). However, solving this ILP at scale can take several tens of seconds delaying capsule placement. WHIZ instead uses a practical approach for the capsule placement problem. First, instead of jointly optimizing global placement decisions for all the capsules, WHIZ solves a "local" problem of placing capsules for each stage independently while still considering inter-stage dependencies; when new stages arrive, or when existing capsules may exceed job quota on a machine, new locations for some of these capsules are determined (see Figure 4). Second, instead of solving a multi-objective optimization, WHIZ uses a linear-time rule-based heuristic to place capsules; the heuristic prioritizes load and locality (in that order) in case machines satisfying all objectives cannot be found. Isolation is always enforced.

Capsule location for new stages (Figure4): When a job j is

h1	*// Q_j: max storage quota per job j and machine m.* Based on fairness considerations across all runnable jobs J.
h2	*// M_v: number machines (out of M) to organize data that //generated by v of j.* a. Count number machines M_{j75} where j is using $< 75\%$ of Q_j; b. $M_v = max(2, M_{j75} \times \frac{M\_M_{j75}}{M})$.
h3	*// Given M_v, compute list of machines $\overrightarrow{M_v}$.* Considers only machines where j is using $< 75\%$ of Q_j; a. Pick machines that provide load balance (LB), data locality (DL) and maximum possible fault tolerance (FT); b. If $\lvert\overrightarrow{M_v}\rvert < M_v$, relax FT guarantees and pick machines that provide LB and DL; c. If $\lvert\overrightarrow{M_v}\rvert$ is still $< M_v$, pick machines that just provide LB.
h4	*// Given M_v, compute total capsules N.* $N = G$ X M_v, where G = capsules per machine
h5	*// Which machines are at risk of violating Q_j?* $\overrightarrow{M_j}$: machines which store data of j and j is using $\geq 75\%$ of Q_j.
h6	*// Which capsules are hot on $\overrightarrow{M_j}$?* Significantly larger in size or have a higher increasing rate than others.

Table 1: Heuristics employed in data organization.

ready to run, **DS-M** invokes an admin-provided heuristic h1 (Table 1) that assigns job j a quota Q_j per machine. Setting up quotas helps ensure isolation across jobs.

When a stage v of job j starts to generate intermediate data, **DS-M** invokes h2 to determine the number of machines M_v for organizing v's data. h2 picks M_v between 2 and a fraction of the total machines which are $\leq 75\%$ of the quota Q_j for job j. $M_v \geq 2$ ensures opportunities for data parallel processing; a bounded M_v (Table 1) controls the ES task launch overhead (§6.2).

Given M_v, **DS-M** invokes h3 to generate a list of machines $\overrightarrow{M_v}$ to materialize data on. It starts by creating three sub-lists: (1) For load balancing (LB), machines are sorted lightest-load-first, and only ones which have $\leq 75\%$ quota usage for the corresponding job are considered. (2) For data locality (DL), we prefer machines which already materialize other capsules for this stage v, or capsules from other stages whose output will be consumed by same downstream stage as v (e.g., two map stages in Figure 2a). (3) For fault tolerance (FT), we strive to place dependent capsules on different machines to minimize failure recovery time. We pick machines where there are no capsules from any of v's k upstream stages in the job, sorted in descending order of k. Thus, for the largest value of k, we have all machines that do not store data from *any* of v's ancestors; for $k = 1$ we have nodes that store data from the immediate parent of v.

We pick machines from the sub-lists to maximally meet our objectives in 3 steps: (1) Pick the least loaded machines that are data local and offer as *high fault tolerance as possible* (machines present in all three sub-lists). Note that as we go down the fault tolerance list in search of a total of M_v machines, we trade-off fault tolerance. (2) If despite reaching the minimum possible fault tolerance, i.e., reaching the bottom of the fault tolerance sub-list – the number of machines picked falls below M_v, we completely trade-off fault tolerance and pick the least loaded machines that are data local. (3) If still the number of machines picked falls below M_v, we simply pick the least-loaded machines and trade-off data locality too.

Finally, given $\overrightarrow{M_v}$, **DS-M** invokes h4 and instantiates a fixed number (G) of capsules per machine leading to total capsules per-stage (N) to be $G \times M_v$. While a large G would aid us in better handling of skew and computation as the capsules can be processed in parallel, it comes at the cost of significant scheduling and storage overheads. We empirically study the sensitivity to G (in §9.4); based on this, our prototype uses $G = 24$.

New locations for existing capsules: Data generation patterns can vary across different stages, and jobs, due to heterogeneous compute logics and data skew. Thus a job j may run out of its Q_j on machine m, leaving no room to grow already-materialized capsules of job j on m. **DS-M** reacts to such dynamics by determining, $\forall j$: machines where job j is using $\geq 75\%$ Q_j (h5), closing capsules that are significantly larger or have a higher growth rate than others on such machines (h6), and invokes heuristic h3 to compute the machines to spread these capsules. This focuses on capsules that contribute most load to machines at risk of being overloaded and thus bounds the number of capsules that will spread out.

5 Data Visibility

We now describe how WHIZ offers programmable data monitoring via the DS (§5.1), and how it initiates data-driven computations using execution predicates (§5.2).

5.1 Data Monitoring

Given that intermediate data properties form the basis of data-driven computation, native support for data monitoring is extremely crucial. The DS through its data organization simplifies monitoring as it consolidates a capsule at one or a few locations rather than it being spread across the cluster (§4.1). WHIZ achieves scalable monitoring via per-job masters **DS-JMs** which track light-weight properties related to their capsules.

WHIZ supports *built-in* and *custom* monitors that gather properties per capsule. They are periodically sent to the relevant **DS-JM**. Built-in monitors constantly collect *coarse-grained* properties such as current capsule size, total or number of unique (k,v) pairs, location(s) and rate of growth; apart from being used for data-driven computation, these are used in runtime data organization (§4.2).

Custom monitors are UDFs (user defined functions) that are used to get *fine-grained* data properties per the job specification. We restrict UDFs to those that can execute in linear time and O(1) state, such as (a) number of entries s.t. values are $<, =,$ or $>$ than a threshold; (b) min, max, avg. of keys; and (c) whether data is sorted or not.

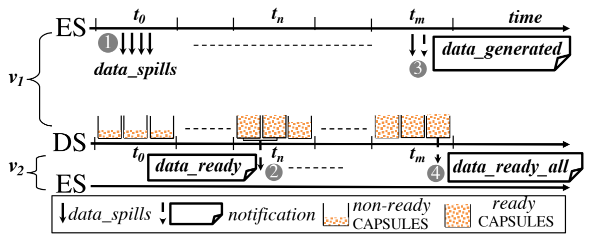


Figure 5: Data-driven computation facilitated by notifications. (1) Intermediate data (v_1) batches sent from ES to DS. (2) DS detects that 2 capsules are ready and sends data_ready notification from DS to ES leading to downstream computation (v_2). (3) ES sends data_generated notification to DS when entire output of v_1 pushed to DS. (4) DS sends data_ready_all notification to ES indicating that all data_ready notifications have been sent.

Getting visibility into intermediate data through monitors enables data-driven computation as we describe next.

5.2 Indicating Data Readiness

The DS is responsible for initiating data-driven computation. The DS achieves this via two key abstractions: *notifications* and *execution predicates*. The decoupled DS and ES interact via notifications which enable, and track progress of, data-driven computation. Execution predicates enable the **DS-JM** to decide when capsules can be deemed ready for corresponding computation to be run on them.

Notifications: WHIZ introduces 3 types of notifications: (1) A *data_ready* notification is sent by the **DS-JM** to the ES whenever a capsule becomes ready (as per the execution predicate) to trigger corresponding computation. (2) A *data_generated* notification is sent by the ES to the **DS-JM** when a stage finishes generating all its intermediate output. This notification is required because the **DS-JM** is unaware of the number of tasks that the ES launches corresponding to a stage, and thus cannot determine when a stage is completed. (3) A *data_ready_all* notification is sent by the **DS-JM** to the ES when a stage has received all its input data (occurs when *data_ready* notifications regarding all ready input capsules are sent). This notification is required because the ES is unaware of the total number of capsules that the DS deems ready.

The use of these notifications is exemplified in Figure 5. Here: ❶ when a stage v_1 generates a batch of intermediate data, a *data_spill* containing the data is sent to the data store, which accumulates it into capsules (t_0 through t_m). ❷ Whenever the **DS-JM** determines that a collection of v_1's capsules (2 capsules in Figure 5 at t_n) are ready for further processing, it sends a *data_ready* notification per capsule to the ES; the ES launches tasks of a consumer stage v_2 to process such capsules. This notification carries per-capsule information such as: a list of machine(s) on which each capsule is spread, and a list of statistics collected by the data monitors. ❸ Finally, a *data_generated* notification – from the ES, generated upon v_1 computation completion – notifies the **DS-JM** that v_1 finished generating *data_spills*. ❹ Subsequently, **DS-JM** notifies the ES via the *data_ready_all* event, that all capsules corresponding to v_1 have sent their data_ready events (at t_m). This enables the ES to determine when the immediate downstream stage v_2, that is reading the data generated by v_1, has received all of its input data.

Execution predicates: The interaction between ES and DS via notifications is initiated by execution predicates whose logic is based on the properties collected by the monitors. Each job stage is typically associated with an execution predicate as indicated by the input program, which is transferred to the **DS-JM** by the WHIZ client. If not, default analytics-specific predicates are applied. WHIZ supports diverse execution predicates such as:

1. Data Generated: This predicate deems capsules ready when the computation generating them is done; this is the default predicate for batch and graph analytics in WHIZ; akin to a barrier in batch systems today and bulk synchronous execution in graph analytics.

2. Record Count $\geq X$: The vanilla version of this predicate deems a capsule ready when it has $\geq X$ records from producers tasks; this is the default predicate for streaming systems in WHIZ; akin to micro-batching in existing streaming systems [58], with the crucial difference that the micro-batch is not wall clock time-based, but is based on the more natural intermediate data count.

This predicate can be extended to support pipelining via *ephemeral compute*, i.e, compute is launched once there is partial data and just for the processing duration. The ability to launch compute ephemerally is particularly useful under heavy resource contention. Ephemeral compute can be used to speed up jobs across analytics if they contain commutative+associative operations (§9).

For example, consider the partial execution of a batch (or graph) analytics job, consisting of the first two logical stages (likewise, first two iterations) $v_1 \rightarrow v_2$. If the processing logic in v_2 contains commutative+associative operations, it can start processing its input before all of it is in place. Using this predicate, a capsule generated by v_1 is ready whenever the number of records in it reaches a threshold X. This enables the ES to overlap v_2's computation with v_1's data generation as follows: (1) Upon receiving a *data_ready* notification from the **DS-JM** for capsules which have $\geq X$ records, the ES launches ephemeral tasks of v_2. (2) Tasks read the current data, compute the associative+commutative function on the (k,v) data read, push the result back to data store (in the same capsules advertised through the received *data_ready* notification) and immediately quit. (3) The **DS-JM** waits for each capsule to grow back beyond threshold X for generating subsequent *data_ready* notifications leading to ephemeral tasks being launched again. (4) Finally, when a *data_generated* notification is received from v_1, the **DS-JM** triggers a final *data_ready* notification for all the capsules generated by v_1, and a subsequent *data_ready_all* notification, to enable v_2's final output to be written in capsules and fully consumed by a downstream stage, say v_3 (similar to Figure 5).

This predicate can be further extended to *across* capsules, i.e., the **DS-JM** could deem all capsules ready when the number of entries generated across all capsules cross a threshold. In streaming, such predicates help improve efficiency and performance as ephemeral tasks are launched only when the required input records have streamed into the system and quit post processing (§9.1.3). On the other hand, systems today lack support for ephemeral compute and are forced to deploy long-standing tasks.

3. Special Records: This predicate deems all output capsules of a stage ready on observing a special record in any one capsule. Stream processing systems often rely on "low watermark" records to ensure event-time processing [19, 40], and to support temporal joins [40]. Such predicates can be used to launch, *on demand*, temporal operators whenever a low watermark record is observed at any of a stage's output capsules. In contrast, systems today have the operators always running and this leads to compute idling when there are no records to process.

6 Execution Service

While the DS initiates data-driven computation by notifying when data is ready for processing, the ES carries out all other data-driven execution aspects by *incrementally generating the physical graph*. It does so via a per-job master **ES-JM** that given ready capsules, and available resources[1]: (a) determines the appropriate processing logic to use (§6.1); (b) determines optimal parallelism and deploys tasks to minimize skew and shuffle; (c) maps capsules to tasks in a resource-aware fashion (§6.2).

6.1 Selecting Compute Logic

Upon detecting a ready capsule, the **DS-JM** sends the *data_ready* notification, with capsule properties (including fine-grained ones) piggybacked, to the **ES-JM**. The **ES-JM** then uses modification predicates associated with this stage to determine the exact processing logic.

Modification predicates give the ability to decide processing logic at runtime based on the received data properties and available resources. Importantly, WHIZ also provides jobs with the flexibility to use *different* processing logic for different input capsules of the same stage. For e.g., consider a batch analytics job that involves joining two tables.[2] In such

[1]Similar to existing frameworks, a cluster-wide Resource Manager decides available resources as per cross-job fairness.

[2]DS via per-job quotas ensures that the input tables use the same # of capsules and because both tables use the same key, i.e., the join key, while

h7	*// $\overrightarrow{C}$: subsets of unprocessed capsules.* a. $CaMax = 2 \times \lvert c \rvert$, c is largest capsule $\in C$; b. Group all capsules $\in C$ into subsets in strict order: i. data local capsules together; ii. each spread capsule, along data-local capsules together; iii. any remaining capsules together; subject to: iv. each subset size $\leq CaMax$; v. conflicting capsules don't group together; vi. troublesome capsules always group together.
h8	*// $\overrightarrow{M}$: preferred machines to process each subset $\in \overrightarrow{C}$.* c. no machine preference for troublesome subsets $\in \overrightarrow{C}$ d. for every other subset $\in \overrightarrow{C}$ pick machine m such that: i. all capsules in the subset are only materialized at m; ii. otherwise m contains the largest materialization of the subset.
h9	Compute $\overrightarrow{R}$: resources to execute each subset $\in \overrightarrow{C}$: e. $\overrightarrow{A}$ = available resources for j on machines $\overrightarrow{M}$; f. $F = \min\left(\frac{\overrightarrow{A}[m]}{\text{total size of capsules allocated to } m}, \text{ for all } m \in \overrightarrow{M}\right)$; g. for each subset $i \in \overrightarrow{C}$: $\overrightarrow{R}[i] = F \times$ total size of capsules allocated to $\overrightarrow{C}[i]$.

Table 2: Heuristics to group capsules and assign them to tasks.

a scenario, the SQL framework running atop sets the modification predicates of the join stage to choose the appropriate join algorithm between, say, sort-merge join[3] and hash join[4] as follows: (a) if both capsules are already sorted, and the max value in the first capsule is less than the min value in the other capsule (no intersection), then skip unnecessarily launching a task to do the join; (b) if the size of one capsule is significantly smaller than the other one (and data is not sorted), then use hash join (as it is typically less expensive to create a hash table of the smaller capsule, than sorting the large one); and (c) if data is sorted or (a)–(b) don't satisfy, then default to sort-merge join.

Crucially, such predicates enable stream jobs submitted to WHIZ, to change their processing logic *over time*, as opposed to being early bound to processing logic (status-quo today). For e.g., predicates allow a job involving temporal join to change its join algorithm over time and choose the appropriate one, from the three choices (a)–(c) above, based on data properties and available resources.

6.2 Task Parallelism, Placement, and Sizing

Given a set of ready capsules (C) for a stage, the ES-JM needs to map capsules to tasks, and determine their location (across machines M) and sizes (resources) so as to minimize cross-task skew and shuffle while taking available resources into account. To do so, we propose an *iterative procedure* that applies a set of heuristics (Table 2) repeatedly until tasks for all ready capsules are allocated, and their locations and sizes determined.

The iterative procedure consists of 3 steps: (a) generate optimal subsets of capsules to minimize cross-subset skew (using h7), (b) decide on which machine should a subset be processed to minimize shuffle (h8), and (c) determine resources required to process each subset (h9).

First, we group capsules C into a collection of *subsets* $\overrightarrow{C}$ using h7. We then try to assign each group to a task. Our grouping into subsets attempts to ensure that data in a subset is spread on just one or a few machines (lines (b.i-b.iii)), which minimizes shuffle, and that the data is spread roughly evenly across subsets (line (b.iv)) making cross-task performance uniform. We place a bound $CaMax$, equaling twice the size of the largest capsule, on the total size of a subset (see line (a)). This ensures that multiple (at least 2) capsules are present in each subset and allows mitigating stragglers by *only* assigning the yet-to-be-processed capsules to the speculative task.[5]

Second, we determine a preferred machine to process each subset using h8; this is a machine where most if not all capsules in the subset are materialized (line (d)). Choosing a machine in this manner minimizes shuffle.

Finally, given available resources across the preferred machines (from the cluster-wide resource manager [52]) we need to allocate tasks to process subsets. But some machines may not have resource availability. For the rest of this iteration, we ignore such machines and the subsets of capsules that prefer such machines.

Given machines with resources $\overrightarrow{A}$, we assign a task for each subset of capsules which can be processed, and allocate task resources *altruistically* using h9. That is, we first compute the minimum resource available to process unit data (F; line (f)). Then, for each task, the resource allocated (line (g)) is F times the total data in the subset of capsules allocated to the task ($\lvert\overrightarrow{C}[i]\rvert$).

Allocating task resources proportional to input size ensures that tasks have similar finish times. Allocating resources corresponding to the minimum available helps further: if a job gets more resources than what is available for the most constrained subset, then it does not help the job's completion time (which is determined by the most constrained subset's processing). Altruistically "giving back" such resources speeds up other jobs/stages.

The above 3 steps repeat whenever new capsules are ready, or existing ones can't be scheduled. Similar to delay scheduling [56], we attempt several tries to execute a group which couldn't be scheduled on its preferred machine due to resource unavailability, before marking capsules *conflicting*. These are

writing to the store, key-range split for both tables is the same.

[3]This (a) sorts the two input capsules on the join key and (b) merges them by comparing the records.

[4]This (a) builds a hash table on the join key using the smaller capsule and (b) probes for matches using the other capsule.

[5]Speculative tasks today [12–14, 38, 59] reprocess the entire input leading to duplicate work and resource wastage.

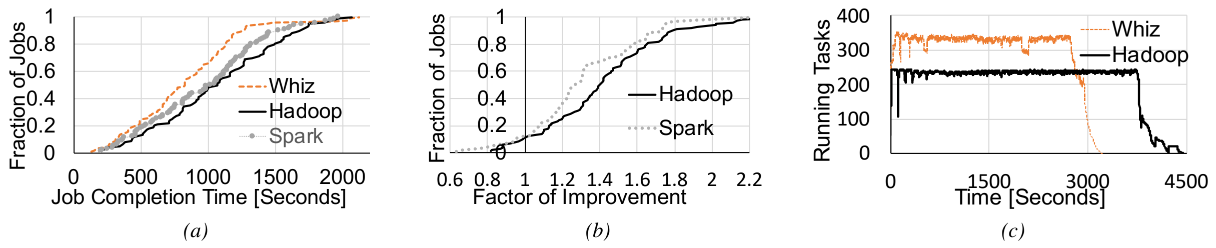


Figure 6: [Batch Analytics] (a) CDF of JCT; (b) CDF of factors of improvement of individual jobs using Whiz *w.r.t. baselines; (c) Snapshot of running tasks during one of the experiments. Gains are lower w.r.t. Spark due to our Hadoop-based implementation, and thus using a non-optimized in-memory store.*

re-grouped in the next iteration (line (b.v)). Finally, capsules that cannot be executed under any grouping are marked troublesome (line (b.vi)) and processed on any machine (line (d.ii)).

7 Fault Tolerance

Task Failure. When a task fails due to a machine failure, only the failed tasks need to be re-executed if the input capsules are not lost. But, this will result in duplicate data in all capsules for the stage leading to data inconsistencies. To address this, we use checksums at the consumer task-side Whiz library to suppress duplicate data.

However, if the failed machine also contains the input capsules of the failed task, then the **ES-JM** triggers the execution of the upstream stage(s) to regenerate the input capsules of the failed task. Recall that Whiz's fault tolerance-aware capsule storage (§4) helps control the number of upstream (ancestor) stages that need to be re-executed in case of data loss.

DS-M/DS-JM/ES-JM. Whiz maintains replicas of DS-M/DS-JM daemons using Apache Zookeeper [28], and fails over to a standby. Given that Whiz generates the physical graph of a job at runtime in a data-driven manner, upon ES-JM failure, we simply need to restart it so that it can resume handling notifications from the DS. During this time already launched tasks continue to run.

8 Implementation

We prototyped Whiz by modifying Tez [5] and leveraging YARN [52]. The DS, implemented from scratch, has three kinds of daemons (managed via YARN): cluster-wide master **DS-M**, per-job masters **DS-JM** and workers **DS-W**. DS-M does data organization across DS-Ws. DS-JMs collect statistics and notify ES-JM when execution predicates are met. DS-Ws run on cluster machines and do node-level management: (a) store data received from ES/other DS-Ws in local in-memory file system (`tmpfs` [48]) and transfer data to other DS-Ws per DS-M directives; (b) report statistics to DS-M/DS-JMs via heartbeats; and (c) provide ACK to tasks for data written.

The ES was implemented by modifying Tez. It consists of per-job masters **ES-JM** which are responsible for generating the physical graph at runtime. ES tasks are modified Tez tasks that have an interface to the local DS-W as opposed to local disk or cluster-wide storage. The Whiz client is a standalone process per-job.

All communication (asynchronous) between DS, ES and client is through RPCs in YARN using Protobuf [8]. We also use RPCs between the YARN Resource Manager (RM) and ES-JM to propagate resource allocations (§6).

9 Evaluation

We evaluated Whiz on a 50-machine cluster deployed on CloudLab [6] using publicly available benchmarks – *batch* TPC-DS jobs, PageRank for *graph analytics*, and synthetic *streaming* jobs. Unless otherwise specified, we set Whiz to use default execution predicates, equal storage quota ($Q_j = 2.5GB$) and 24 capsules per machine.

9.1 Experiment Setup

Workloads: We consider a mix of jobs, all from TPC-DS (**batch**), or all from PageRank (**graph**). For **streaming**, we use a variety of different queries described in detail later. In each experiment, jobs are randomly chosen and follow a Poisson arrival distribution with average inter-arrival time of 20s. Each job lasts up to 10s of minutes, and takes as input tens of GBs of data. We run each experiment thrice and present the median.

Cluster, baseline, metrics: Machines have 8 cores, 64GB memory, 256GB storage, and a 10Gbps NIC. We compare Whiz as follows: (1) **Batch**: *vs.* Tez [5] running atop YARN [52], for which we use the shorthand "Hadoop" or "CC"; and *vs.* SparkSQL [15]; (2) **Graph**: *vs.* Giraph (i.e., open source Pregel [42]); and *vs.* GraphX [24]; (3) **Streaming**: *vs.* SparkStreaming [58].

For a fair comparison, we ensure Hadoop/Giraph use `tmpfs`. We study the relative improvement in the average job completion time (JCT), or $JCT_{CC}/JCT_{\text{WHIZ}}$. We measure efficiency using *makespan*.

9.1.1 Batch Analytics

Performance and efficiency: Figure 6a shows the JCT distributions of Whiz, Hadoop, and Spark for the TPC-DS work-

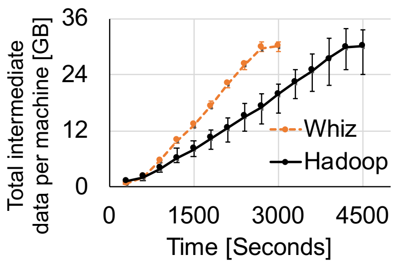


Figure 7: Cross-job average, min *and* max *intermediate data per machine during one of our batch analytics experiments.*

load. Only 0.4 (1.2) highest percentile jobs are worse off by $\leq 1.06\times$ ($\leq 1.03\times$) than Hadoop (Spark). WHIZ speeds up jobs by **1.4×** (**1.27×**) on average, and **2.02×** (**1.75×**) at 95*th* percentile w.r.t. Hadoop (Spark). Also, WHIZ improves makespan by **1.32×** (**1.2×**).

Figure 6b presents improvement for individual jobs. For more than 88% jobs, WHIZ outperforms Hadoop and Spark. **Only 12**% jobs slow down to $\leq 0.81\times$ ($0.63\times$) using WHIZ. Gains are $> 1.5\times$ for $> 35\%$ jobs.

Sources of improvements: We observe that *more rapid processing* due to *data-driven execution*, and *better data management* contribute most to benefits.

First, we snapshot the number of running tasks across all the jobs in one of our experiments when running WHIZ and Hadoop (Figure 6c). WHIZ has 1.45× more tasks scheduled over time which translates to jobs finishing 1.37× faster. It has **1.38×** better cluster efficiency than Hadoop. Similar observations hold for Spark (omitted).

The main reasons for rapid processing/high efficiency are: (1) The DS ensures that most tasks are data local (**76**% in our expts). This improves average *consumer* task completion time by 1.59×. Resources thus freed can be used by other jobs' tasks. (2) Based on DS-provided properties, ES's data-driven actions provide similar input sizes for tasks in a stage – within 14.4% of the mean.

Second, Figure 7 shows the size of the cross-job total intermediate data per machine. We see that Hadoop generates heavily imbalanced load spread across machines. This creates many storage hotspots and slows down tasks competing on those machines. Spark is similar. WHIZ mitigates hotspots (§4) improving overall performance.

We observe jobs generating less intermediate data are more prone to performance losses in WHIZ, especially under ample resource availability as WHIZ strives for capsule-local task execution (§6.2). If resources are unavailable, WHIZ will assign the task to a data-remote node, or get penalized waiting for data-local placement.

9.1.2 Graph Processing

We run multiple PageRank (40 iterations) jobs on the Twitter Graph [17, 18]. In each iteration, vertices run the processing logic and exchange their output as messages with each other. WHIZ groups messages into capsules based on vertex ID. We use an *execution predicate* that deems a capsule ready when ≥ 1000 messages are present.

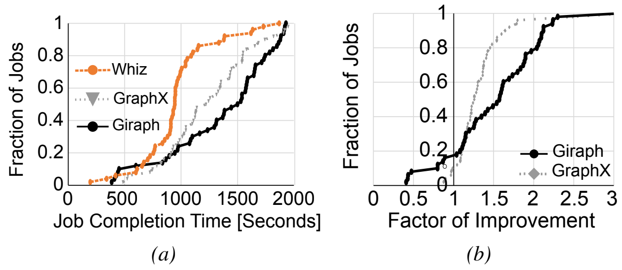


Figure 8: [Graph Analytics] (a) CDF of JCT using WHIZ, *GraphX and Giraph; (b) CDF of factors of improvement of individual jobs using* WHIZ *w.r.t. GraphX and Giraph.*

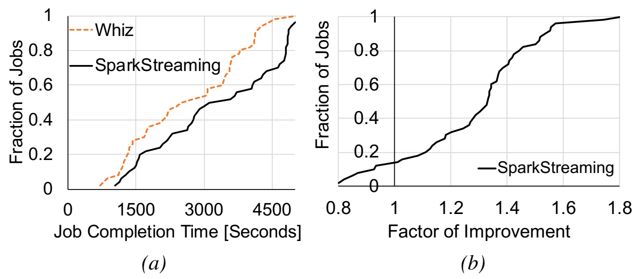


Figure 9: [Stream Analytics] (a) CDF of JCT using WHIZ *and SparkStreaming; (b) CDF of factors of improvement of individual jobs using* WHIZ *w.r.t. SparkStreaming.*

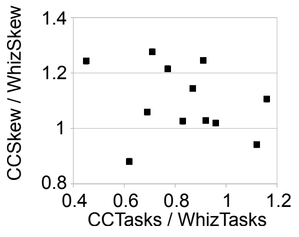
Figure 8a shows the JCT distribution of WHIZ, GraphX and Giraph. WHIZ speeds up jobs by **1.33×** (**1.57×**) on average and **1.57×** (**2.24×**) at the 95*th* percentile w.r.t. GraphX (Giraph) (Figure 8b). Gains are lower w.r.t. GraphX, due to its efficient implementation atop Spark. However, **< 10**% jobs are slowed down by $\leq 1.13\times$.

Improvements arise for two reasons. First, WHIZ is able to *deploy appropriate number of ephemeral tasks*: execution predicates immediately indicate data availability, and runtime parallelism (§6.2) allows messages to high-degree vertices [24] to be processed by more than one task. Also, WHIZ has 1.53× more tasks (each runs multiple vertex programs) scheduled over time; rapid processing and runtime adaptation to data directly leads to jobs finishing faster. Second, because of ephemeral compute, WHIZ doesn't hold resources for a task if not needed, resulting in **1.25×** better cluster efficiency.

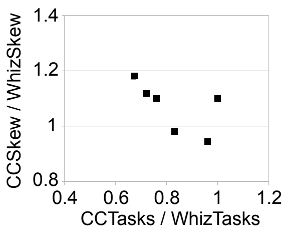
9.1.3 Stream Processing

We run multiple stream jobs, each calculating top 5 common words for every 100 distinct words from synthetic streams replaying GBs of text data from HDFS.

Spark Streaming discretizes the records stream into time-based micro-batches and processes every micro-batch duration. We configure the micro-batch interval to 1 minute. With WHIZ, given the semantics of the processing logic, we use an



(a)



(b)

Figure 10: Hadoop w.r.t. WHIZ *fraction of tasks allocated vs. the fraction of skew in a given stage: (a) for a job with* 12 *stages where* WHIZ *improves JCT by* $1.6\times$*; (b) for a job with* 6 *stages, where* WHIZ *improves JCT by* $1.2\times$. $\frac{CCSkew}{WhizSkew} > 1$ *means* WHIZ *has less skew;* $\frac{CCTasks}{WhizTasks} < 1$ *means Hadoop under-parallelizes.*

execution predicate to enable computation whenever ≥ 100 distinct records are present.

Figures 9a, 9b show our results. WHIZ speeds up jobs by **1.33×** on average and **1.55×** at the 95*th* %-ile. Also, 15% of the jobs are slowed down to around **0.8×**.

The gains are due to *data-driven computation* via execution predicates; WHIZ does not have to delay execution till the next micro-batch if data can be processed now. A Spark Streaming task has to wait as it has no data visibility. In our experiments, more than **73%** executions happen at less than 40*s* time intervals with WHIZ.

Additionally, we evaluate the role of modification predicates in streaming in §9.2.

9.1.4 WHIZ Overheads

CPU, memory overhead: We find that DS-W (§8) processes inflate the memory and CPU usage by a negligible amount even when managing data close to storage capacity. DS-M and DS-JM have similar resource profiles.

Latency: We compute the average time to process heartbeats from various ES/DS daemons, and WHIZ client. For 5000 heartbeats, the time to process each is $2-5ms$. We implemented the WHIZ client and ES-JM logic atop Tez AM. Our changes inflate AM decision logic by $\leq 14ms$ per request with negligible increase in AM memory/CPU.

Network overhead from events/heartbeats is negligible.

9.2 Benefits of Data-driven Computation

The overall benefits above included the effects of execution predicates and incrementally generating the physical graph. We now delve deeper to further shed light into late-binding benefits.

Skew and parallelism: Figure 10 shows fractions of skew and parallelism as generated by Hadoop w.r.t. WHIZ for two TPC-DS jobs from one of our runs. WHIZ's ability to dynamically change parallelism at runtime, driven by the number of capsules for each vertex, leads to significantly less data skew than Hadoop. When Hadoop is under-parallelizing, the skew is significantly higher than WHIZ (up to 1.43×). Over-parallelizing does not help either; Hadoop incurs up to 1.15× larger skew, due to its rigid data partitioning and tasks allocation schemes. Even when WHIZ incurs more skew (up to 1.26×), corresponding tasks will get allocated more resources to alleviate this overhead (§6.2).

% Skew	Improvement Factor
10%	1.1
30%	1.47
50%	1.87
70%	2.48
90%	2.67

Table 3: Improvement in cumulative time using modification predicates w.r.t no predicates. Predicate chooses hash join if the ratio of input capsules' sizes is ≥ 3.

Modification predicates: To evaluate the benefits enabled by WHIZ's ability to late-bind processing logic, we pick a query from our TPC-DS workload which has a join and run it with and without modification predicates while varying the skew between the input tables. Modification predicates allow the job to pick the join algorithm between sort-merge join and hash join (see §6.1) for the different tasks. Table 3 shows the relative improvement in cumulative time (summation over duration of all tasks of the job) with and without predicates (sticks to sort-merge join). We see that predicates improve the cumulative time **1.1×–2.8×** as the skew in capsule sizes increases. This is because with modification predicates, WHIZ chooses to use the hash join when skew between the task inputs exists as building a hashmap on the smaller input is typically cheaper than sorting the other input (occurs when sort-merge join is used instead). Gains increase with skew as the join performance difference also increases.

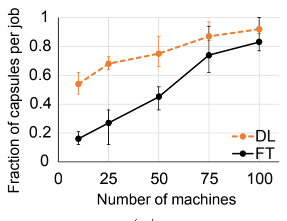
We also quantify the benefits of modification predicates for stream processing. We run a stream query that performs event-time temporal join over 3 minute intervals (execution predicate indicates to wait for watermark) with and without the above modification predicates. We change skew-% randomly (from 10%, 20%,..., 90%) between the two input sources over the same time interval and observe that using modification predicates leads to 1.7× average improvement in cumulative time.

Additionally, we run microbenchmarks to delve further into WHIZ's data-driven benefits (results in Appendix C).

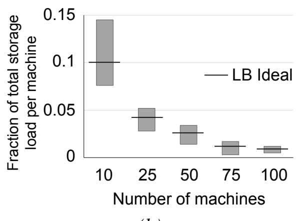
9.3 Load Balancing, Locality, Fault Tolerance

To evaluate DS load balancing (LB), data locality (DL) and fault tolerance (FT), we stressed the data organization under different cluster load. We used job arrivals and all stages' capsule sizes from one of our TPC-DS runs.

Figure 11 shows that: (1) WHIZ prioritizes load balancing and data locality over fault tolerance across cluster loads (§4.2); (2) when the available resources are scarce (5× higher load than initial), all three metrics suffer. However, the maxi-



(a)



(b)

Figure 11: (a) Average, min and max fraction of capsules which are data local (DL) respectively fault tolerant (FT) across all the jobs for different cluster load; (b) Max, min and ideal storage load balance (LB) on every machine for different cluster load.

% Machines Failed	JCT [Seconds] Avg	Min	Max
None	725	215	2100
10%	740	250	2320
25%	820	310	2360
50%	1025	350	2710
75%	1600	410	3300

Table 4: JCT under random machine failures.

mum load imbalance per machine is $< 1.5\times$ than the ideal, while for any job, $\geq 47\%$ of the capsules are data local. Also, on average 16% of the capsules per job are fault tolerant; (3) less cluster load ($0.6\times$ lower than initial) enables more opportunities for DS to maximize all of the objectives: $\geq 84\%$ of the per-job capsules are data local, 71% are fault tolerant, with at most $1.17\times$ load imbalance per machine than the ideal.

Failures: Using the same workload, we also evaluated the performance impact in the presence of machine failures (Table 4). We observe that WHIZ does not degrade job performance by more than $1.13\times$ even when 25% of the machines fail. This is mainly due to DS's ability to organize capsules to be fault tolerant across ancestor stages and avoid data recomputations. Even when 75% of the machines fail, the maximum JCT does not degrade by more than $1.57\times$, mainly due to capsules belonging to some ancestor stages still being available, which leads to fast recomputation for corresponding downstream vertices.

9.4 Sensitivity Analysis

Impact of Contention: We vary storage load, and hence resource contention, by changing the number of machines while keeping the workload constant; half as many servers lead to twice as much load. We see that at $1\times$ cluster load, WHIZ improves over Hadoop by $1.39\times$ ($1.32\times$) on average in terms of JCT (makespan). Even *at high contention* (up to $4\times$), WHIZ's gains keep increasing $1.83\times$ ($1.42\times$). This is because of data-driven execution and better data management which minimizes resource wastage, time spent in shuffling, and leads to few hotspots.

Multiple of Original Load	# Capsules 8	16	20	24	28	32	36	40
1	1.07	1.33	1.46	1.52	1.57	1.63	1.54	1.46
2	1.10	1.16	1.53	1.58	1.56	1.61	1.47	1.31
4	0.85	1.12	1.34	1.39	1.32	1.16	0.95	0.74

Table 5: Factors of improvement w.r.t. Hadoop for different number of capsules per machine and cluster load.

Impact of G (number of capsules per machine): We now provide the rationale for picking $G = 24$. Table 5 shows the factors of improvements w.r.t. Hadoop for different values of G and levels of contention.

The main takeaways are as follows: for $G = 8$ the performance gap between WHIZ and Hadoop is low ($< 1.1\times$). This is expected because small number of capsules results in less data locality (each capsule is more likely to be spread). Further, the gap decreases at high resource contention. In fact, at $4\times$ the cluster load, Hadoop performs better ($0.85\times$). At larger values of G the performance gap increases. For example, at $G = 24$, WHIZ gains are the most (between $1.39\times$ and $1.58\times$). This is because larger G implies (1) more flexibility for WHIZ to balance the load across machines; (2) more likely that few capsules are spread out; (3) lesser data skew and more predictable per task performance. However, a very large G does not necessarily improve performance, as it can lead to massive task parallelism. The resulting scheduling overhead degrades performance, especially at high load.

Altruism: Assigning resources altruistically is beneficial as it improves median (95*th* %-ile) JCT by $1.48\times$ ($4.8\times$) for our TPC-DS runs w.r.t a greedy approach where tasks use all of their available resources. Only 16% jobs are slowed down by $\leq 0.6\times$.

10 Related Work

We now discuss the various related efforts to overcome the various limitations of compute-centricity. WHIZ, with its clean separation of compute and intermediate data, overcomes the various limitations of compute-centricity in a unified manner while prior related efforts propose point-fixes to a subset of the limitations that plague compute-centric execution engines.

Data opacity: Almost all database and bigdata SQL systems [10, 12, 54] use statistics computed ahead of time to optimize execution. Adaptive query optimizers (QOs) [22] use dynamically collected statistics and re-invoke the QO to re-plan queries top-down. In contrast, WHIZ alters the query plans on-the-fly at the *execution layer* based on *run-time* data properties, thereby circumventing additional expensive calls to the QO. Tukwila [30] reformulates queries by using run-time visibility in a limited fashion to fix poor statistics maintenance in QOs. WHIZ instead enables much richer visibility and supports a richer set of actions that enable true data-centric behavior. RoPE [12] leverages historical statistics from prior plan executions in order to tune future executions. WHIZ, instead uses runtime properties.

CIEL [45] is an execution engine that provides support for data-dependent iterative or recursive algorithms by dynamically deciding the execution graph as tasks execute. However, low-level execution aspects such as per-stage parallelism are decided beforehand. Optimus [33] extends frameworks such as CIEL [45] and Dryad [29] to enable runtime logic rewriting and parallelism selection by using streaming-based algorithms to collect aggregated statistics on intermediate data. RIOS [39] is an optimizer for Spark that solely focuses on optimizing joins by deciding the join order and stage-level join implementation using the approximate statistics collected at runtime. While Optimus and RIOS attempt to provide data visibility, neither of them does a clean separation of compute and data; this limits data-local processing and imposes I/O interference as intermediate data organization is determined by the compute structure. Moreover, both still resort to compute-driven scheduling and do not use data visibility to decide if/when tasks should be scheduled. Further, RIOS adopts static per-stage parallelism, and cannot make fine-grained logic changes (e.g., task-level) as table-level statistics are aggregated by a separate Spark job and then sent to the Spark driver which is responsible for making runtime changes. Overall, WHIZ is a general approach to data-driven computation that subsumes all prior efforts, and enables new data-driven execution benefits; its clean separation of data enables data-locality and I/O isolation management.

Skew and parallelism: Some parallel databases [25, 34, 55] and big data systems [38] dynamically adapt to data skew for single large joins. In contrast, WHIZ holistically solves data skew for all joins across multiple jobs. [25, 38] deal with skew in MapReduce by dynamically splitting data for slow tasks into smaller partitions and processing them in parallel. But, they can cause additional data movement from already slow machines leading to poor performance. Hurricane [16] mitigates skew via an adaptive task partitioning scheme by cloning slow tasks at runtime and performing data organization such that all tasks, be it the primary task or its clones, can access data that requires processing. However, Hurricane can lead to additional processing overheads as it does not take data locality into account while organizing data (data corresponding to the same key can be spread across multiple machines) and also involves an additional merge step that combines the partial outputs of the clones using the end-user provided merging logic. Moreover, Hurricane does not have fine-grained visibility into intermediate data and thus cannot do fine-grained task logic changes and still adopts compute-driven scheduling. Henge [32] supports multi-tenant streaming by deciding parallelism based on SLOs. However, it still adopts compute-driven scheduling.

Decoupling: Naiad [44] and StreamScope [53] also decouple intermediate data. They tag intermediate data with vector clocks which are used to trigger compute in the correct order. Both support ordering driven computation, orthogonal to data-driven computation in WHIZ. Also, StreamScope is not applicable to batch/graph analytics. Crail [49] decouples intermediate data from compute so that various execution engines can easily leverage modern storage hardware (including tiered storage) to perform intermediate data management. However, the compute structure still decides the number of partitions across which data is organized. Additionally, it adopts a similar data storage abstraction as well as data placement policy to Hurricane and thus incurs additional overheads as it does not take data locality into account. Moreover, Crail recommends replicating data in case fault tolerance is required which can further lead to additional overheads. Instead, WHIZ provides fault tolerance by intelligent placement of intermediate data so as to minimize recovery time. Also, similar to Hurricane, it does not have fine-grained data visibility to drive all aspects of execution.

Storage inefficiencies: For batch analytics, [31,46] addresses storage inefficiencies by pushing intermediate data to the appropriate external data services (like Amazon S3 [1], Redis [9]) while remaining cost efficient and running on serverless platforms. Similarly, [35] is an elastic data store used to store intermediate data of serverless applications. However, since this data is still opaque, and compute and storage are managed in isolation, these systems cannot support data-driven computation or achieve data locality and load balancing simultaneously.

11 Summary

The compute-centric nature of existing data analytics frameworks hurts flexibility, performance, efficiency, and job isolation. With WHIZ, analytics undergo data-driven execution aided by a clean separation of compute from intermediate data. WHIZ enables monitoring of data properties and using these properties to decide all aspects of execution - what to launch, where to launch, and how many tasks to launch, while ensuring isolation. Our evaluation using batch, stream and graph workloads shows that WHIZ significantly outperforms state-of-the-art.

Acknowledgments

We would like to thank our shepherd, Chris Rossbach, the anonymous reviewers of NSDI'21 and the members of WISR Lab for their insightful comments and suggestions. This research was supported by NSF Grants CNS-1565277, CNS-1719336, CNS-1763810, CNS-1838733 and by gifts from Google and VMware.

References

[1] Amazon S3. https://aws.amazon.com/s3/.

[2] Apache Hadoop. http://hadoop.apache.org.

[3] Apache Hive. http://hive.apache.org.

[4] Apache Samza. http://samza.apache.org.

[5] Apache Tez. http://tez.apache.org.

[6] Cloudlab. https://cloudlab.us.

[7] Presto | Distributed SQL Query Engine for Big Data. prestodb.io.

[8] Protocol Buffers. https://bit.ly/1mISy49.

[9] Redis. https://redis.io/.

[10] Spark SQL. https://spark.apache.org/sql.

[11] Storm: Distributed and fault-tolerant realtime computation. http://storm-project.net.

[12] S. Agarwal, S. Kandula, N. Burno, M.-C. Wu, I. Stoica, and J. Zhou. Re-optimizing data parallel computing. In *NSDI*, 2012.

[13] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective Straggler Mitigation: Attack of the Clones. In *NSDI*, 2013.

[14] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in mapreduce clusters using Mantri. In *OSDI*, 2010.

[15] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational data processing in Spark. In *SIGMOD*, 2015.

[16] L. Bindschaedler, J. Malicevic, N. Schiper, A. Goel, and W. Zwaenepoel. Rock you like a hurricane: Taming skew in large scale analytics. In *EuroSys*, 2018.

[17] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *WWW*, 2011.

[18] P. Boldi and S. Vigna. The webgraph framework i: Compression techniques. In *WWW*, 2004.

[19] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *IEEE Computer Society TCDE Bulletin*, 36(4), 2015.

[20] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing data transfers in computer clusters with Orchestra. In *SIGCOMM*, 2011.

[21] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[22] A. Deshpande, Z. Ives, and V. Raman. Adaptive query processing. *Found. Trends databases*, 1(1):1–140, Jan. 2007.

[23] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant Resource Fairness: Fair allocation of multiple resource types. In *NSDI*, 2011.

[24] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, 2014.

[25] K. A. Hua and C. Lee. Handling data skew in multiprocessor database computers using partition tuning. In *VLDB*, 1991.

[26] B. Huang, N. W. Jarrett, S. Babu, S. Mukherjee, and J. Yang. Cumulon: Matrix-based data analytics in the cloud with spot instances. *PVLDB*, 9(3):156–167, 2015.

[27] B. Huang and J. Yang. Cumulon-d: Data analytics in a dynamic spot market. *PVLDB*, 10(8):865–876, 2017.

[28] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *ATC*, 2010.

[29] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[30] Z. G. Ives, D. Florescu, M. Friedman, A. Levy, and D. S. Weld. An adaptive query execution system for data integration. *ACM SIGMOD Record*, 1999.

[31] E. Jonas, Q. Pu, S. Venkataraman, I. Stoice, and B. Recht. Occupy the cloud: Distributed computing for the 99%. In *SOCC*, 2017.

[32] F. Kalim, L. Xu, S. Bathey, R. Meherwal, and I. Gupta. Henge: Intent-driven multi-tenant stream processing. In *SoCC*, 2018.

[33] Q. Ke, M. Isard, and Y. Yu. Optimus: A dynamic rewriting framework for data-parallel execution plans. In *EuroSys*, 2013.

[34] M. Kitsuregawa and Y. Ogawa. Bucket spreading parallel hash: A new, robust, parallel hash join method for data skew in the super database computer (sdc). In *VLDB*, 1990.

[35] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *OSDI*, 2018.

[36] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, I. Pandis, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-Milne, and M. Yoder. Impala: A modern, open-source SQL engine for Hadoop. In *CIDR*, 2015.

[37] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter heron: Stream processing at scale. In *SIGMOD*, 2015.

[38] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skewtune: Mitigating skew in mapreduce applications. In *SIGMOD*, 2012.

[39] Y. Li, M. Li, L. Ding, and M. Interlandi. Rios: Runtime integrated optimizer for spark. In *SoCC*, 2018.

[40] W. Lin, Z. Qian, J. Xu, S. Yang, J. Zhou, and L. Zhou. Streamscope: continuous reliable distributed processing of big data streams. In *NSDI*, 2016.

[41] K. Mahajan, M. Chowdhury, A. Akella, and S. Chawla. Dynamic query re-planning using QOOP. In *OSDI*, 2018.

[42] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD*, 2010.

[43] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, et al. Mllib: Machine learning in apache spark. *JMLR*, 17(1):1235–1241, 2016.

[44] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *SOSP*, 2013.

[45] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. Ciel: A Universal Execution Engine for Distributed Data-Flow Computing. In *NSDI*, 2011.

[46] Q. Pu, S. Venkataraman, and I. Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *NSDI*, 2019.

[47] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino. Apache tez: A unifying framework for modeling and building data processing applications. In *SIGMOD*, 2015.

[48] P. Snyder. tmpfs: A virtual memory file system. In *EUUG Conference*, 1990.

[49] P. Stuedi, A. Trivedi, J. Pfefferle, A. Klimovic, A. Schuepbach, and B. Metzler. Unification of temporary storage in the nodekernel architecture. In *ATC*, 2019.

[50] A. Thusoo, R. Murthy, J. S. Sarma, Z. Shao, N. Jain, P. Chakka, S. Anthony, H. Liu, and N. Zhang. Hive – a petabyte scale data warehouse using Hadoop. In *ICDE*, 2010.

[51] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. Storm@twitter. In *SIGMOD*, 2014.

[52] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: Yet another resource negotiator. In *SoCC*, 2013.

[53] L. Wei, Q. Zhengping, X. Junwei, Y. Sen, Z. Jingren, and Z. Lidong. Streamscope: Continuous reliable distributed processing of big data streams. In *NSDI*, 2016.

[54] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and rich analytics at scale. In *SIGMOD*, 2013.

[55] Y. Xu, P. Kostamaa, X. Zhou, and L. Chen. Handling data skew in parallel joins in shared-nothing systems. In *SIGMOD*, 2008.

[56] 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 *EuroSys*, 2010.

[57] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[58] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant stream computation at scale. In *SOSP*, 2013.

[59] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *OSDI*, 2008.

A Whiz APIs

Table 6 shows the various APIs exposed by Whiz that are used by frameworks running atop Whiz to submit programs. Similar to APIs today, (1)–(3) APIs are job composition APIs to create logical directed graphs (Whiz has similar APIs to load a graph). Crucially, Whiz *does not require* the frameworks to specify low-level details like parallelism and partitioning strategy. The *addCustomMonitor* API is used by the framework to submit UDFs to collect custom statistics (apart from the built-in monitors that Whiz supports).

During the data embellishment phase, the framework annotates the logical graph with execution and modification predicates using *addExecutionPredicate* API and *addModificationPredicate* API respectively. The framework provides the predicates based on data properties via a UDF.

We now show how to write execution and modification predicates for a number of applications used in our testbed experiments (§9).

Figure 12a shows the execution predicate used to run the PageRank algorithm. Specifically, the UDF specifies that as soon as 1000 messages corresponding to a vertex are received, the data is ready to be processed. Similarly, Figure 12b shows the execution predicate used to run the streaming job that returns the top 5 words when we see 100 distinct words. This predicate indicates that as soon as 100 unique key records globally have been received, the data is ready to be processed.

Lastly, Figure 12c shows how to specify modification predicates to decide the join algorithm on the fly for a batch SQL query as well as a streaming query involving a temporal join. This predicate takes as input the properties collected of the two capsules (from the two input tables) and chooses the join algorithm based on the amount of skew. If the amount of skew is less than the threshold specified by the framework, then sort-merge join is used; otherwise hash join is chosen.

B Allocating Capsules to Machines ILP

We consider how to place multiple jobs' capsules to avoid hotspots, reduce per-capsule spread (for data locality) and minimize job runtime impact on data loss. We formulate a binary integer linear program (see Table 7) to this end. The indicator decision variables, x_i^k, denote that all future data to capsule g_k is materialized at machine M_i. The ILP finds the best x_i^k's that minimizes a multi-part weighted objective function, one part each for the three objectives mentioned above.

The first part (O_1) represents the maximum amount of data stored across all machines across all capsules. Minimizing this ensures load balance and avoids hotspots. The second part (O_2) represents the sum of *data-spread penalty* across all capsules. Here, for each capsule, we define the primary location as the machine with the largest volume of data for that capsule. The total volume of data in non-primary locations is the data-spread penalty, incurred from shuffling the data prior to processing it. The third part (O_3) is the sum of fault-tolerance penalties across capsules. Say a machine m storing intermediate for current stage s fails; then we have to re-execute s to regenerate the data. If the machine also holds data for ancestor stages of s then multiple stages have to be re-executed. If we ensure that data from parent and child stages are stored on different machines, then, upon child data failure only the child stage has to be executed. We model this by imposing a penalty whenever a capsule in the current stage is materialized on the same machine as the parent stage. Penalties O_2, O_3 need to be minimized.

Finally, we impose isolation constraint (C_1) requiring the total data for a job to not exceed an administrator set quota Q_j. Quotas help ensure isolation across jobs.

However, solving this ILP at scale can take several tens of seconds delaying capsule placement. Thus, Whiz uses a linear-time rule-based heuristic to place capsules (as described in §4).

C Whiz Microbenchmarks

Apart from the experiments on the 50-machine cluster (§9), we also ran several microbenchmarks to delve deeper into Whiz's data-driven benefits. The microbenchmarks were run on a 5 machine cluster and the workloads consists of the following jobs: $\mathbb{J}_1$ ($v_1 \rightarrow v_2$) and $\mathbb{J}_2$ ($v_1 \rightarrow v_2 \rightarrow v_3$). These patterns typically occur in TPC- DS queries.

Skew and parallelism: Figure 13a shows the execution of one of the $\mathbb{J}_2$ queries from our workload when running Whiz and CC. Whiz improves JCT by 2.67× over CC. CC decides stage parallelism tied to the number of data partitions. That means stage v_1 generates 2 intermediate partitions as configured by the user and 2 tasks of v_2 will process them. However, execution of v_1 leads to data skew among the 2 partitions ($1GB$ and $4GB$).On the other hand, Whiz ends up generating capsules that are approximately equal in size and decides at runtime a max. input size per task of $1GB$ (twice the largest capsule). This leads to running 5 tasks of v_2 with equal input size and 2.1× faster completion time of v_2 than CC.

Over-parallelizing execution does not help. With CC, v_2 generates 12 partitions processed by 12 v_3 tasks. Under resource crunch, tasks get scheduled in multiple waves (at 570s in Figure 13a) and completion time for v_3 suffers (85s). In contrast, Whiz assigns at runtime only 5 tasks of v_3 which can run in a single wave; v_3 finishes 1.23× faster.

Straggler mitigation: We run an instance of $\mathbb{J}_1$ with 1 task of v_1 and 1 task of v_2 with an input size of $1GB$. A slowdown happens at the v_2 task, which was assigned 2 capsules by Whiz.

In CC (Figure 13b), once a straggler is detected (v_2 task at 203s), it is allowed to continue, and a speculative task v_2' is launched that duplicates v_2's work. The work completes when v_2 *or* v_2' finishes (at 326s). In Whiz, upon straggler detection, the straggler (v_2) is notified to finish processing the current capsule; a task v_2' is launched and assigned data

	API	Description
1	createJob(name:Str, type:Type)	Creates a new job which can be of type BATCH, STREAM or GRAPH.
2	createStage(j:Job, name:Str, impl:StageImpl, prop:StageProperties)	Adds a logical stage of to a Job with a default processing logic. StageProperties specifies properties of the logic (e.g., if it is commutative+associative).
3	addDependency(j: Job, s1: Stage, s2: Stage)	Adds a starts before relationship between stages s1 and s2.
4	addCustomMonitor(j:Job, s:Stage, impl:DataMonitorImpl)	Adds a custom data monitor to compute statistics over data generated by stage *s*. DataMonitorImpl is a UDF.
5	addExecutionPredicate(j:Job, s:Stage, predicates:ExecutionPred)	Decides when downstream stages can consume current stage's data based on the predicates specified by ExecutionPred. ExecutionPred is used by DS to decide when data is ready for processing.
6	addModificationPredicate(j:Job, s:Stage, predicates:ModifyPred)	Decides processing logic for the input capsules that are ready based on the predicates specified by ModifyPred. ModifyPred is used by the ES to decide which processing logic to use based on the data properties from the DS.

Table 6: WHIZ *APIs - Used by the frameworks running atop* WHIZ *to translate the high-level job submitted by end users to* WHIZ*-compatible data-driven logical graphs.*

```
def ExecutionPredicate():
  for key in keys:
    if (DS.monitor.num_entries(key) >= 1000):
      return true
  return false
```

```
def ExecutionPredicate():
  if (DS.monitor.global_unique_entries >= 100):
       return true
  return false
```

```
def ModificationPredicates(capA, capB):
  // THRESHOLD is set by the framework
  sizeRatio = max(capA.size, capB.size)/ min(capA.size, capB.size)
  if (sizeRatio >= THRESHOLD):
     return HashJoinImpl //Refers to hash join
  return SortMergeJoinImpl //Refers to sort-merge join
```

Figure 12: Examples of predicates. (a) Execution predicate for the PageRank algorithm - deem capsule ready when it has 1000 messages, (b) Execution predicate for the streaming application - deem capsules ready when we see 100 unique entries and (c) Modification predicate for changing join algorithm on the fly.

Objectives (to be minimized):

O_1	$\max_i \left(\sum_k (b_i^k + x_i^k e^k)\right)$
O_2	$\sum_k \left(P^k - \left(\sum_{i\in I^k_-} x_i^k\right) b^k_{\hat{i}(k)} + \sum_{i\in I^k_+} x_i^k \left(b_i^k + e^k\right)\right)$
O_3	$\sum_k \left((1-f^k)\sum_{i\in I^\circ} x_i^k\right)$

Constraints:

C_1	$\sum_{k:J(g_k)=j} \left(b_i^k + x_i^k e^k\right) \leq Q_j, \quad \forall j,i$

Variables:

x_i^k	Binary indicator denoting capsule g_k is placed on machine i

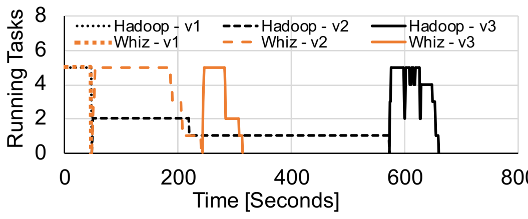
Parameters:

b_i^k	Existing number of bytes of capsule g_k in machine i
e^k	Expected number of remaining bytes for capsule, g_k
P^k	$e^k + \sum_i b_i^k$
$J(g_k)$	The job ID for job g_k
$\hat{i}(k)$	$\text{argmax}_i \quad b_i^k$
I^k_-, I^k_+	$\{i : b_i^k \leq b^k_{\hat{i}(k)} - e^k\}$, $\{i : b_i^k > b^k_{\hat{i}(k)} - e^k\}$
f^k	Binary parameter indicating that capsules for same stage as g_k share locations with capsules for preceding stages
I°	Set of machines where capsules of preceding stages are stored
Q_j	Administrative storage quota for job, j.

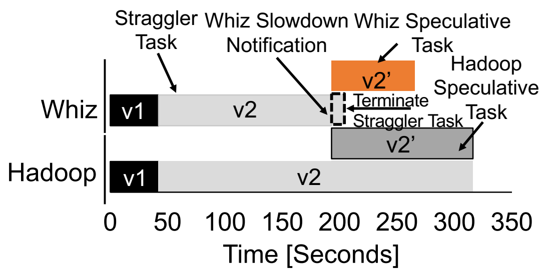
Table 7: Binary ILP formulation for capsule placement.

from v_2's unprocessed capsule. v_2 finishes processing the first capsule at 202s; v_2' processes the other capsule and finishes 1.7× faster than v_2' in CC.

Modification Predicates: We consider a job which processes words and, for words with < 100 occurrences, sorts them by frequency. The program structure is $v_1 \to v_2 \to v_3$, where v_1 processes input words, v_2 computes word occurrences, and v_3 sorts the ones with < 100 occurrences. In CC, v_1 generates $17GB$ of data organized in 17 partitions; v_2 generates $8GB$ organized in 8 partitions. Given this, 17 v_2 tasks and 8 v_3 tasks execute, leading to a CC JCT of 220s. Here, the entire data generated by v_2 has to be analyzed by v_3. In contrast, WHIZ uses modification predicates for v_3 as follows - (a) if all the # entries of all keys in the capsule is > 100, then we unnecessarily don't launch a task; (b) otherwise we launch the task to do the sort. We observe that WHIZ ignores processing two capsules at runtime, and 6 tasks of v_3 (instead of 8) are executed; JCT is 165s (1.4× better).



(a)



(b)

Figure 13: (a) Controlling task parallelism significantly improves WHIZ*'s performance over CC. (b) Straggler mitigation with* WHIZ *and CC.*

Pushing the Physical Limits of IoT Devices with Programmable Metasurfaces

Lili Chen[2,1,*], Wenjun Hu[4], Kyle Jamieson[3], Xiaojiang Chen[2], Dingyi Fang[2] and Jeremy Gummeson[1]

[1]Univ. of Massachusetts Amherst, [2]Northwest Univ. (China), [3]Princeton Univ., [4]Yale Univ.

Abstract

Small, low-cost IoT devices are typically equipped with only a single, low-quality antenna, significantly limiting communication range and link quality. In particular, these antennas are typically linearly polarized and therefore susceptible to polarization mismatch, which can easily cause $10-15$ dB of link loss when communicating with such devices. In this work, we highlight this under-appreciated issue and propose the augmentation of IoT deployment environments with programmable, RF-sensitive surfaces made of metamaterials. Our smart metasurface mitigates polarization mismatch by rotating the polarization of signals that pass through or reflect from the surface. We integrate our metasurface into an IoT network as LLAMA, a Low-power Lattice of Actuated Metasurface Antennas, designed for the pervasively used 2.4 GHz ISM band. We optimize LLAMA's metasurface design for both low transmission loss and low cost, to facilitate deployment at scale. We then build an end-to-end system that actuates the metasurface structure to optimize for link performance in real time. An empirical evaluation demonstrates gains in link power of up to 15 dB, and wireless capacity improvements of 100 and 180 Kbit/s/Hz in through-surface and surface-reflective scenarios, respectively, attributable to the polarization rotation properties of LLAMA's metasurface.

1 Introduction

Internet of Things (IoT) devices have achieved widespread adoption due to shrinking hardware costs and software management tools that ease installation by the end user. In recent years, a wide range of IoT devices have resulted in diverse systems including mobile devices such as smartwatches [31] and health trackers or statically deployed devices including sensors, cameras, voice assistants, and other appliance automation [14, 15]. One key property these devices share is low-cost hardware, in particular low-cost radios, allowing for a minimal consumer price point. Such devices are typically

[*]Work conducted on internship at Univ. of Massachusetts Amherst.

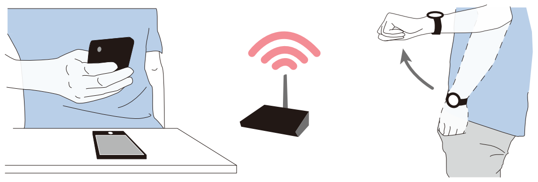
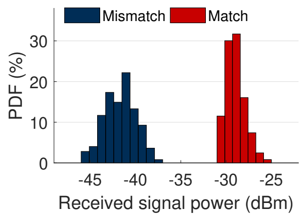


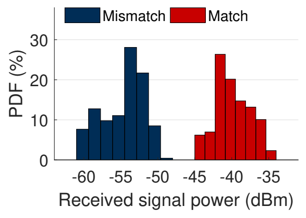
Figure 1: Low-cost IoT devices and wearables suffer from polarization mismatch when their antennas become oriented perpendicularly with respect to an AP's antenna.

deployed by non-experts who understand neither their home's wireless environment nor the deployment considerations that govern wireless performance. Devices are typically deployed in a configuration that is well-suited for a particular application or use-case, but may not be the ideal placement in terms of communications performance. This combination of cheap hardware and non-ideal network topology results in significant opportunities to improve wireless performance.

One source of performance degradation in such deployments is a significant power loss caused by a *polarization mismatch* [13, 32] between a low-cost dipole antenna on an IoT device and antennas on a Wi-Fi access point (AP)—in higher performance devices (*i.e.*, mobile handsets) this loss is usually mitigated through the use of circularly polarized or dynamically-switched linearly polarized antennas in different orientations. Low-cost IoT devices instead use one cheap, linearly polarized dipole antenna that results in weak, fragile links between transmitters and receivers. In addition to misaligned stationary devices, mobile devices such as wearables can suffer from dynamic antenna misalignment as a user swings their arm, for example, as Figure 1 illustrates. This effect can be significant: microbenchmark experiments show that moving between orthogonal and aligned relative antenna polarization results in ≈ 10 dB of power variation at the receiver, for both the low power Wi-Fi link between an Arduino equipped with an ESP8266 module [19] and an 802.11g Wi-Fi AP [5] shown in Figure 2 (a), and for a Bluetooth link between a smartwatch [1] and a Raspberry Pi 3 [36]



(a) Wi-Fi communication (802.11g) between an AP [5] and a cheap ESP8266-based Arduino [19].



(b) Bluetooth communication between a Huawei Watch [1] and a Raspberry Pi 3 [36].

Figure 2: *Impact of polarization mismatch.* Received signal power distributions for matching and mismatching antenna orientations between IoT transmitter-receiver pairs. Polarization mismatch significantly reduces link signal power.

shown in Figure 2 (b).

In this work, we specifically investigate how to change the effective relative orientation of antennas at the communication endpoints, without hardware modifications to the endpoints themselves. Achieving this objective would allow us to maintain a low bill of materials cost for IoT wireless communications components, and significantly improve the performance for existing off-the-shelf devices. Our approach to changing effective polarization alignment hinges instead, on changing the radio propagation environment itself, with a ***Low-Power Lattice of Actuated Metasurface Antennas*** (*LLAMA*), a tunable smart surface made with inexpensive metamaterials [39, 41]. LLAMA is deployed in the radio environment near the IoT endpoints, and is able to change the polarization of incident waves as they travel from sender to receiver. As we show in this paper, the LLAMA substrate can be programmed dynamically to effect just the right amount of polarization rotation needed to help the ongoing communication between nearby endpoints. LLAMA follows the PRESS [43] approach, which was the first to outline a vision of programmable radio environments along with a preliminary experimental setup that validates the feasibility of change the perceived channel at the communication endpoints.

Designing a metasurface in the 2.4 GHz ISM band requires us to overcome significant challenges. First, longer wavelengths in the 2.4 GHz band require larger and thicker metasurface substrates, which can attenuate the incident signal significantly. While an inefficient structure could rotate polarization, losses would dominate and the structure would attenuate the incident signal, hampering communication. Therefore, the metasurface structure needs to be optimized for low transmission loss. Second, since we aim to realize pervasive deployments of these structures, we need to develop materials that are low cost, avoiding high performance but relatively expensive RF materials commonly used in other implementations of metasurfaces. Overcoming both of these challenges results in a pervasively deployable substrate that can compensate for losses between different endpoint pairs.

Indeed, naively replacing a high performance substrate (*i.e.*, Rogers 5880 [38]) with a low-cost substrate (*i.e.*, FR4 [20]) results in higher transmission loss due to FR4's inherent physical properties; this in turn significantly attenuates power at the receiver, reducing link throughput and communication distance. To deal with this problem, we optimize the metasurface structure to ensure the overall system has both low transmission loss, as well as a scalable price point. Specifically, we choose a cheap material (FR4) as the substrate, use a minimum number of substrate layers for the required bandwidth, and minimize the thickness of each layer to significantly reduce the losses associated with FR4.

To enable real-time polarization optimization, a receiver must report received power to a controller which in turn rotates polarization by modifying a pair of bias voltages. We provide a novel method to estimate the polarization rotation angle induced by the metasurface which can vary with link distance—understanding this mapping can enable rotation sensing and tracking.

Contributions. To summarize, LLAMA is the first system that leverages an inexpensive RF substrate to optimize the radio environment in real time, thereby avoiding signal losses caused by polarization mismatch, and thus enables higher quality communication links between IoT devices. In this work, we optimize a metasurface structure based on microwave attenuation theory and achieve comparable polarization tunability to a similar system that uses relatively expensive materials. We validate a proof-of-concept implementation of LLAMA for both communication and sensing with comprehensive experiments. Our results show that LLAMA enables polarization rotation within $3^\circ - 45^\circ$, improves the signal strength by 15 dB (transmission) and 17 dB (reflection) with respect to mismatched antenna polarizations. LLAMA also holds great potential to enhance sensing applications, as demonstrated in § 5.2.2.

2 A Polarizing View of Wireless

Electromagnetic polarization describes the parametric trajectory of the electric and magnetic field vectors of a planar electromagnetic wave as it propagates through space. The polarization of RF wave propagation is a fundamental characteristic of wireless communication, but it has not received as much attention as issues like multipath fading and interference. An antenna constrains outgoing or incoming RF propagation to a particular plane. Therefore, communication is only possible if the signal propagation planes at both the transmitter and the receiver are well aligned.

Polarization loss. One challenge in mobile wireless communication links is the significant power loss due to polarization mismatch. Three examples of various transmitter antennas and their associated far-field electric fields are depicted in

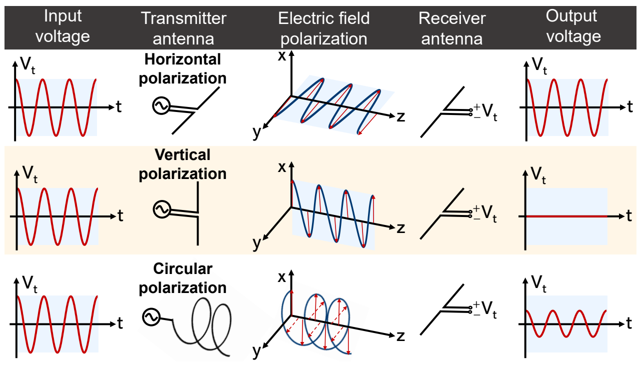


Figure 3: Signal transmission loss under different antenna polarization combinations between transmitter and receiver [13].

Figure 3. If the signal is being received by a horizontally polarized receiver antenna, it will be polarization matched to the transmitter antenna with horizontal linear polarization and the power received will primarily be a function of the transmit power and free space path loss. If the transmitter antenna is rotated in space, the received signal will continue to degrade due to polarization mismatch to the point where the very little signal is received when the antennas are completely mismatched with orthogonal polarizations. The signal loss is less when one of the antennas is circularly polarized [1].

As shown in Figure 2, polarization mismatch can be debilitating for IoT devices. Higher performance devices such as mobile phones use switched antennas or circular polarized antennas to mitigate polarization mismatch, but low-cost IoT devices like smartwatches typically have a single low-quality antenna.

Correcting polarization mismatch. Intuitively, polarization mismatch can be corrected by rotating the polarization of the signal before it arrives at the receiver. Here we show the mathematical foundation of polarization rotation.

In general, the polarization state of radio waves can be described by a 2×1 Jones vector J. Consider a plane perpendicular to the direction of signal propagation. Any polarization state can be represented by two orthogonal components in that plane (*i.e.*, projected onto the X and Y axes) with different amplitude and phase. The *Jones vector* is [28]:

$$J = \left[\begin{array}{c} a^x \\ a^y e^{j\pi/2} \end{array} \right], \quad (1)$$

where a^x and a^y represent the X and Y polarized signal components respectively.

When a manipulation surface is aligned with the x-y coordinate axis, the *Jones matrix* is defined as [28]:

$$M = e^{j\alpha} \left[\begin{array}{cc} 1 & 0 \\ 0 & e^{j\pi/2} \end{array} \right], \quad (2)$$

[1] A theoretical 3 dB degradation in coupling due to polarization mismatch will also occur when one of the antennas is circularly polarized while the other is linearly polarized.

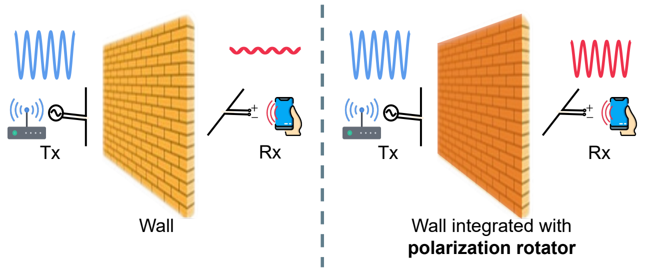


Figure 4: Wireless communication system without/with polarization rotator between mismatched transmitter (Tx) and receiver (Rx).

where α is a phase delay between the X and Y axes. If the surface is rotated counterclockwise by a degree of θ, the Jones matrix becomes [28]:

$$M_\theta = R(\theta) M R^T(\theta), \qquad R(\theta) = \left[\begin{array}{cc} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{array} \right], \quad (3)$$

where $R(\theta)$ is a rotation matrix.

In systems with multiple layers of polarization manipulation surface between the incident wave and outgoing wave, the outgoing Jones vector J_{out} is obtained by multiplying the Jones vector of incident wave with the Jones matrix of each surface layer [28]:

$$J_{out} = M_N ... M_2 M_1 J_{in}, \quad (4)$$

where M_N is a 2×2 Jones matrix, representing the N^{th} surface.

3 System Design

In this section, we introduce the LLAMA architecture (§ 3.1) and illustrate the properties of the metasurface hardware with HFSS simulation results (§ 3.2). Next, we illustrate our approach towards actuating polarization angle in real-time by manipulating bias voltages for the polarization rotator (in § 3.3). While the primary goal of LLAMA is to enhance signal quality between two endpoints, it can also be used for orientation sensing, which requires estimating relative polarization rotation between endpoint; § 3.4 describes a technique for this purpose.

3.1 System Overview

LLAMA is a low power system that is designed to reduce significant wireless signal loss caused by polarization mismatch between the transmitter and receiver. As shown in Figure 4, the signal from a mismatched transmitter arrives at the receiver with a lower loss when the intermediate wall includes a polarization rotator. LLAMA has the ability to improve the communication quality and extend the sensing range in the widely used ISM frequency band.

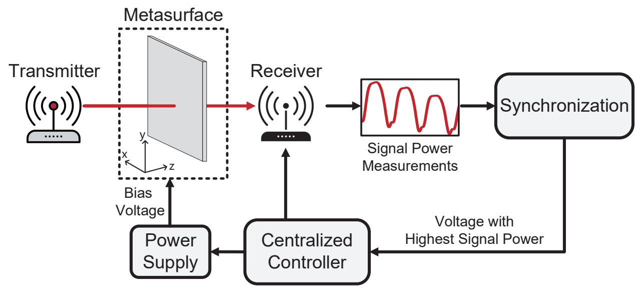


Figure 5: LLAMA system architecture, showing transmitter and receiver endpoints, the LLAMA substrate deployed in the environment, and control signal flow.

An overview of our system architecture is depicted in Figure 5 and consists of these four elements:

Metasurface. The metasurface used in LLAMA is a polarization rotator implemented using a low-cost FR4 substrate; the polarization rotator is tunable and uses biasing voltages within phase shifters in both the X and Y axes to define a rotation angle. The metasurface is deployed in a structural element (*i.e.*, wall) and influences wireless signals that reflect from or propagate through the metasurface.

Centralized controller. A centralized controller observes the power measured at a receiver and uses a search algorithm to determine a set of bias voltages that maximize received signal power by finding the optimal rotation angle that achieves a polarization match between the antennas at the endpoints.

Power supply. The bias voltages used to tune the metasurface are set with a programmable DC power supply. By synchronizing the power supply output with the receiver, we can manipulate the polarization rotator with an optimal rotation angle in real time, with a rotation that maximizes signal power. Two bias voltages are needed for the phase shifters in the X and Y axes; while bias voltages as high as 30 V are needed, the metasurface draws only 15 nA of current. In future implementations, a circuit that generates these bias voltages could be integrated directly on the metasurface.

Endpoints. The endpoint receiver reports its received signal strength to the controller, which then determines how to actuate the metasurface by manipulating the two bias voltages.

3.2 Metasurface Architecture

Tunable metasurfaces are implemented as layered structures that consist of copper patterns printed on controlled dielectric substrates; these layers perform different functions that reflect, bias, or guide EM waves. A transmissive metasurface uses a biasing network sandwiched between or adjacent to waveguide layer(s) to modify the transmissive signal properties in a controlled manner. In contrast, a reflective metasurface uses a metallic plane as one of its layers, where the signal passes through a wave guide layer and a biasing layer, and reflects from the metallic plane in the same relative direction with a different angle of departure. Depending on the bias voltages used LLAMA can operate in either a transmissive or reflective mode.

Cost-effective metasurface design. Our design was inspired by a 10 GHz design [45], and we calculate the correct geometries of circuit elements for 2.4 GHz instead based on the impedance matching.

To achieve polarization rotation for both x-polarized and y-polarized waves, we construct a polarization rotator consisting of a tunable birefringent structure (BFS) placed between two quarter wave plates (QWP) [17]. The QWPs are rotated by $+45^\circ$ and -45° with respect to the BFS, which causes the phase delays for two orthogonal polarizations differ by 90°. The Jones matrices of the two QWPs can be expressed as:

$$Q_{+45^\circ} = e^{j\alpha} R(+45^\circ) \begin{bmatrix} 1 & 0 \\ 0 & e^{j\pi/2} \end{bmatrix} R^T(+45^\circ), \tag{5}$$

$$Q_{-45^\circ} = e^{j\alpha} R(-45^\circ) \begin{bmatrix} 1 & 0 \\ 0 & e^{j\pi/2} \end{bmatrix} R^T(-45^\circ). \tag{6}$$

The tunable BFS is a transmissive metasurface that can rotate the polarization of the X and Y axes, independently. The Jones matrix of the BFS is:

$$B = e^{j\beta} \begin{bmatrix} 1 & 0 \\ 0 & e^{j\delta} \end{bmatrix}, \tag{7}$$

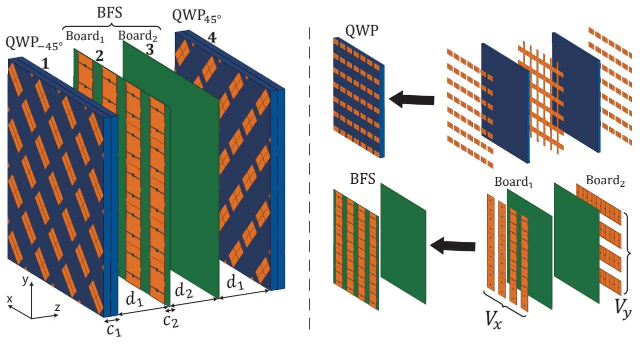
where β is the transmission phase as the signal passes through the BFS, irrespective of initial polarization orientation, while δ represents the transmission phase difference between the X and Y polarizations, which can be adjusted by manipulating the biasing voltages of the X and Y axes as shown in Figure 5. The entire Jones matrix of the polarization rotator is:

$$\begin{aligned} P &= Q_{+45^\circ} B Q_{-45^\circ} \\ &= e^{j(\alpha+(\pi/2)+\beta+(\delta/2))} \begin{bmatrix} \cos(\delta/2) & -\sin(\delta/2) \\ \sin(\delta/2) & \cos(\delta/2) \end{bmatrix}. \end{aligned} \tag{8}$$

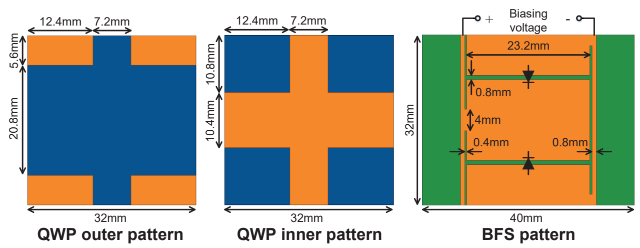
In summary, the proposed structure can rotate the polarization of a wave by $\delta/2$ rotation degrees, according to the rotation matrix presented in Equation (3).

In Figure 6 (a) we show the microstrip geometries used in our metasurface design. The metasurface consists of a tunable BFS (layers 2 and 3 in Fig. 6 (a)) placed between two QWPs (layers 1 and 4 in Fig. 6 (a)) rotated 45° with respect to the BFS. The BFS includes two birefringent boards rotated by 90° with respect to each other, each board acts as a phase shifter that supports different polarization rotations by manipulating the bias voltages (V_x and V_y) of integrated varactor diodes (black) along X and Y axes. The metallic patterns (orange) plated on the substrate boards (green and blue) act as admittance components.

We next consider the transmission efficiency (S_{21}) of X and Y axis polarized signals, as that is among the most important performance metric of the overall metasurface design. Higher



(a) Metasurface topology of LLAMA.



(b) Unit dimension of metasurface in LLAMA. The black elements in BFS represent the varactor diodes.

Figure 6: Polarization rotator structure designed for the 2.4 GHz ISM band. The distances between adjacent boards are $d_1 = 7$ mm and $d_2 = 11$ mm, the FR4 substrate thicknesses of QWP and BFS are $c_1 = 12$ mm and $c_2 = 0.6$ mm.

transmission efficiency indicates better performance. For a two-port network as shown in Figure 7, the amplitude (normalized voltage) of incoming waves (a_1 and a_2) and the outgoing waves (b_1 and b_2) are given by [35]:

$$\begin{cases} a_1 = \frac{V_1+Z_0I_1}{2\sqrt{Z_0}} \\ a_2 = \frac{V_2+Z_0I_2}{2\sqrt{Z_0}} \end{cases}, \quad \begin{cases} b_1 = \frac{V_1-Z_0I_1}{2\sqrt{Z_0}} \\ b_2 = \frac{V_2-Z_0I_2}{2\sqrt{Z_0}} \end{cases}, \tag{9}$$

where V_1 and V_2 are the normalized voltage of port 1 and port 2, I_1 and I_2 are the normalized current of port 1 and port 2, Z_0 is the matched impedance. The scattering matrix S relates the incoming waves to the outgoing waves as [35]:

$$\begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} S_{11} & S_{12} \\ S_{21} & S_{22} \end{bmatrix} \times \begin{bmatrix} a_1 \\ a_2 \end{bmatrix}, \; S_{ij} = \frac{b_i}{a_j}|a_k = 0 \,\forall\, k \neq j. \tag{10}$$

S_{11} and S_{22} are reflection coefficients, S_{21} and S_{12} are transmission coefficients. Then the transmission efficiency can be calculated according to the following equation [35]:

$$eff = \begin{cases} |S_{21}^{xx}|^2 + |S_{21}^{yx}|^2 \,, & for \;\; x-polarized \;\; wave \\ |S_{21}^{xy}|^2 + |S_{21}^{yy}|^2 \,, & for \;\; y-polarized \;\; wave \end{cases}, \tag{11}$$

where S_{21}^{yx} is obtained from the x-polarized component of incoming wave a_1^x and the y-polarized component of outgoing wave b_2^y.

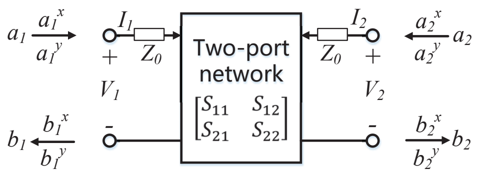


Figure 7: The scattering parameters of metasurface can be measured according to the matched impedance of the input and output ports [35].

We can potentially obtain high transmission efficiency (see Figure 8) by directly scaling the circuit geometry of an existing 10 GHz design [45] to 2.4 GHz, but a key limitation is its use of an expensive, low-loss dielectric substrate (Rogers 5880) [45]. While this material achieves high transmission efficiency, it is cost prohibitive at scale. Instead, we choose a commodity FR4 substrate and characterize the behavior of the FR4 structure using an HFSS simulation environment to analyze critical parameters in the 2.4 GHz ISM band. The key problem is that FR4 (0.02 dielectric loss tangent) causes much larger signal loss than Rogers 5880 (0.0009 dielectric loss tangent), and thus severely decreases the transmission efficiency, as shown in Figure 9.

To reduce transmission loss, we simplify the structure of the tunable phase shifter layers, and decrease the thickness of FR4 by replacing it with an air gap since the dielectric loss tangent of air is 0. By comparing Figure 10 and Figure 8, we can see that our optimized structure made of cheap FR4 can achieve comparable transmission efficiency to more complex structures and expensive materials. We use fewer (*i.e.*, two) phase shifting layers made with thinner substrate; since the supported bandwidth of a phase shifter changes approximately linearly with the transmission line length, that is, the thickness of the substrate. Suppose the thickness of substrate is λ/m, the bandwidth can be represented as below [35]:

$$\Delta f = f_0(2 - \frac{m}{\pi} arccos[\frac{\Gamma}{\sqrt{1-\Gamma^2}} \frac{2\sqrt{Z_I Z_L}}{|Z_L - Z_I|}]), \tag{12}$$

where f_0 is the design center frequency of phase shifter, Γ is the maximum tolerable reflection coefficient, Z_I and Z_L are input impedance and load impedance, respectively. Our design achieves (150 MHz of bandwidth with efficiency $>$ -5 dB), which is wider than the target ISM frequency band that has less than 100 MHz of bandwidth.

Estimating Polarization Efficiency. A voltage controlled capacitance is used to actuate the tuning of the X and Y planes — here we use an x-polarized incident wave as an example to show the polarization rotation results. The transmission efficiencies of the simulated frequencies under various voltage combinations are shown in Figure 11, which are always higher than -8 dB in the $2.4 - 2.5$ GHz ISM frequency band. The other set of measurements looks at how the polarization angle can be controlled by adjusting the lumped tuning capacitance used for the X and Y axis biasing layers. Varying

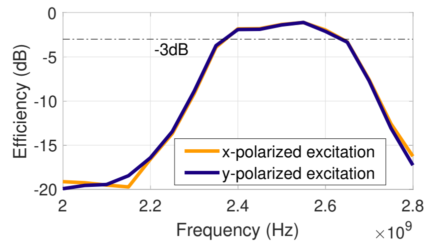


Figure 8: S_{21} efficiency of cascaded polarization rotator layers using Rogers 5880 substrate (loss tangent is 0.0009).

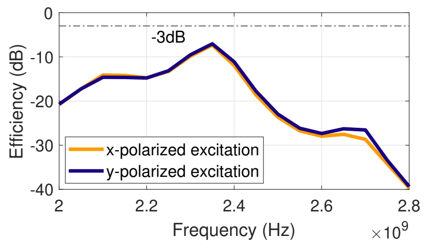


Figure 9: S_{21} efficiency of cascaded polarization rotator layers using FR4 substrate (loss tangent is 0.02).

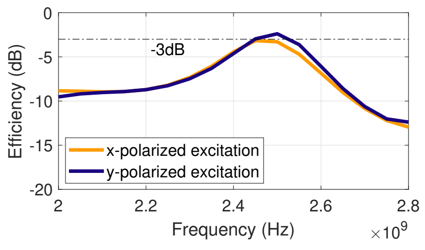


Figure 10: S_{21} efficiency of **optimized** cascaded polarization rotator layers using FR4 substrate (loss tangent is 0.02).

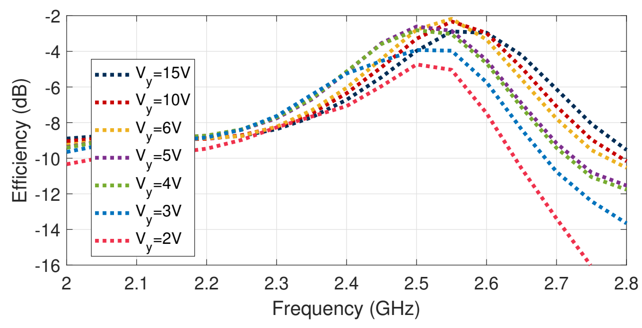


Figure 11: S_{21} efficiencies under different voltage combinations of X and Y axes. The results show polarization can be controlled by changing the biasing voltages of phase shifter.

Table 1: Simulated rotation degrees (θ_r).

θ_r (°)		$V_x(V)$						
		2	3	4	5	6	10	15
$V_y(V)$	2	11.6	26.1	36.8	41.0	44.3	48.3	48.7
	3	6.5	12.4	26.6	32.2	35.2	38.6	39.2
	4	23.0	4.9	10.9	17.3	20.8	25.0	25.6
	5	27.0	9.3	7.4	14.0	18.0	22.6	23.2
	6	41.8	25.0	7.9	2.1	4.2	10.2	10.7
	10	45.8	30.0	13.7	7.9	2.8	5.1	5.6
	15	48.2	33.1	18.2	12.9	7.3	1.9	2.0

this capacitance from 0.84 pF to 2.41 pF for both the X and Y axes resulted in a polarization rotation angle that varied between 1.9° and 48.7° (see Table 1). We have also simulated the polarization rotator structure in the 900 MHz band used for RFID and found comparable performance after additional scaling.

3.3 Metasurface Control

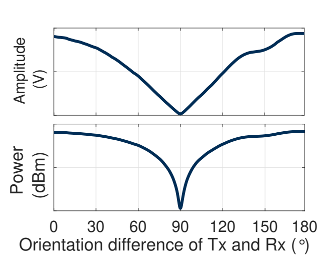
To enable polarization rotation control, we need to change the capacitance of the X and Y axis phase shifters; in our design this is accomplished by changing the bias voltage of the integrated varactor diodes (SMV1233) in the X and Y polarities, which in turn changes their capacitance and thus phase. All diodes in a given polarization are controlled using the same bias voltage; we use a programmable power supply

Algorithm 1: Biasing Voltage Sweep

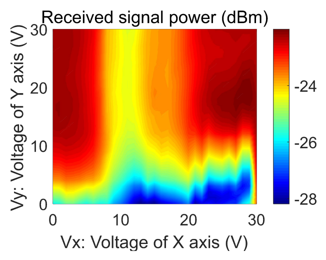
```
Input: Number of iterations: $N$; Number of voltage
  tuning steps for $X$ and $Y$ axes per iteration: $T$
Initialization: Voltage sweep range of $X$ and $Y$ axes
  in first iteration ($n = 1$): $[V^{min}_{x,1}, V^{max}_{x,1}] = [0, 30]$,
  $[V^{min}_{y,1}, V^{max}_{y,1}] = [0, 30]$
for $n = 1, ..., N$ do
    for $\tau_x = 1, ..., T$ do
        $V_{x,n,\tau_x} = V^{min}_{x,1} + (\tau_x - 1)(V^{max}_{x,n} - V^{min}_{x,n})/T$
        for $\tau_y = 1, ..., T$ do
            $V_{y,n,\tau_y} = V^{min}_{y,1} + (\tau_y - 1)(V^{max}_{y,n} - V^{min}_{y,n})/T$
        end
    end
    if Received signal power at voltage combination
      $(V_{x,n,\tau_x}, V_{y,n,\tau_y})$ is strongest then
        $V^{min}_{x,n+1} = V_{x,n,\tau_x} - (V^{max}_{x,n} - V^{min}_{x,n})/T$,
        $V^{max}_{x,n+1} = V_{x,n,\tau_x}$,
        $V^{min}_{y,n+1} = V_{y,n,\tau_y} - (V^{max}_{y,n} - V^{min}_{y,n})/T$,
        $V^{max}_{y,n+1} = V_{y,n,\tau_y}$
    end
end
Output: Optimal voltage combination:
  $(V_{x,N,\tau_x}, V_{y,N,\tau_y})$
```

for this purpose and these voltages can be as high as 30 V to account for errors induced during fabrication and assembly, hence we set $0-30$ V as the voltage sweep range of the X and Y axes. The power supply is connected to a desktop computer through a USB interface, and is controlled by a Python script that uses the Virtual Instrument Software Architecture (VISA) standard with a maximum voltage switching frequency of 50 Hz. With a voltage step of 1 V, the full scan across both X and Y axes takes $\sim$ 30 seconds, which prevents real-time applications.

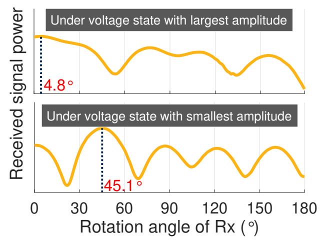
To reduce the sweep time, we start with a coarse-grained voltage sweep then increase the resolution of control as summarized in Algorithm 1. Specifically, we define N as the number of iterations, and T as the number of voltage adjust-



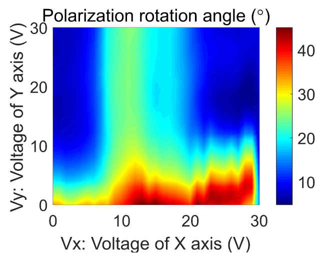
(a) Signal power as a function of Tx rotation without metasurface.



(b) Signal power over voltage combination of metasurface.



(c) Minimum and maximum polarization rotation angles computation.



(d) Estimated polarization rotation angle over voltage combination.

Figure 12: Polarization rotation degree estimation according to the received signal power. The acceptable "band" around the optimal value shown in (c) is chosen by considering multipath interference.

ments per iteration. The total time cost of N iterations is $0.02 \times N \times T^2$. We empirically set T to 5 and N to 2 , according to the switching speed and the voltage resolution of the programmable power supply. After N iterations across the X and Y axes, we can determine the optimal voltage combination to yield the strongest received signal.

Synchronization between rotator and receiver. For real-time polarization correction and communication link optimization, it is necessary to correlate the currently received sample with the bias voltage state, so that we can determine an optimal voltage combination that enables the strongest received signal power. In our prototype, for simplicity we directly connect the receive antenna to the voltage supply, which allows us to assume the voltage switch speed and receiver sampling rate have a constant relationship over time. This allows us to relate received signal samples with the voltage state of LLAMA at a given time instance. A full implementation can have the receiver explicitly send channel state information to the controller, as in previous work [18, 29].

One important aspect of the design we want to highlight is that the leaking current of our metasurface is as low as 15 nA, which means the metasurface does not need a large battery or significant power from the AC mains to keep it powered; it can maintain operation with a modestly sized capacitor.

3.4 Polarization Rotation Degree Estimation

Besides increasing SNR, we can also sense the relative orientation of the two endpoints with LLAMA. Here we present our approach for rotation angle estimation according to the received signal power reported by the endpoint receiver, no matter the transmitter-receiver distance; as distances between the transmitter and receiver become comparable to the size of the metasurface (See Figure 4), a fraction of the signal can bypass the metasurface without polarization rotation, and this results in less overall perceived rotation at the receiver. According to our benchmark experimental result plotted in Figure 12 (a), we observed that the received signal power (before the dBm conversion) can be approximated as a linear change with the orientation difference between transmitter and receiver. When we perform measurements across a full voltage sweep, we can get the maximum potential improvement to signal power. To obtain the polarization rotation angle for an unknown transmitter-receiver distance (*i.e.*, the potential power improvement over an orientation sweep is unknown), the key is determining the minimum and maximum polarization rotation angles. We take the following steps.

Step 1: Fix the receiver at the same orientation with the transmitter, by rotating the receiver to find an orientation θ_0 where the received power is largest.

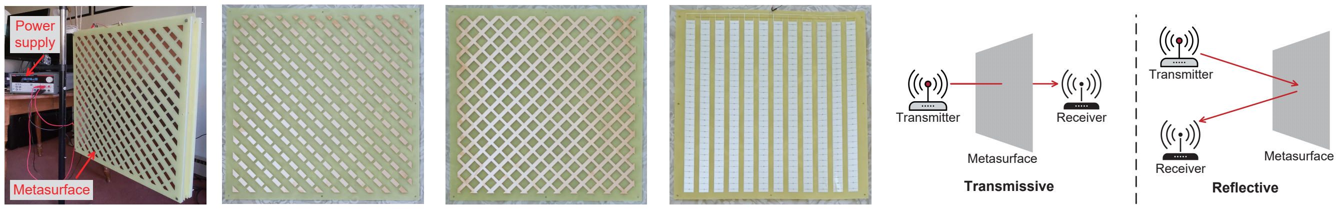
Step 2: Sweep across voltage combinations V_{min} and V_{max} corresponding to min and max powers, respectively (*i.e.*, parallel and orthogonal polarizations).

Step 3: Set the voltage state to the two searched combinations, respectively. At each voltage state, rotate the receiver by 180° to find the new orientation where the power is strongest. The two new orientations of V_{min} and V_{max} can be defined as θ_{max} and θ_{min} as shown in Figure 12 (c). The differences of the receiver's initial orientation and two new orientations $|\theta_0 - \theta_{min}|$ and $|\theta_0 - \theta_{max}|$ correspond to the minimum and maximum polarization rotation angles, respectively.

The antenna that needs to be rotated is fixed on a turntable and rotated via remote control [2]. From the experimental results of the match setup shown in Figure 12 (b-d), we can see that the polarization rotation angle varies between $5° - 45°$ during the voltage sweep.

4 Implementation and Experimental Setup

Metasurface. We fabricated the metasurface with a total surface area of $48 \times 48\ cm^2$ and a thickness of 5 cm, including 180 functional units (Figure 13 (a-d)). The biasing voltages of the metasurface are provided by a programmable power supply (TektronixSeries 2230G [6]) through two DC channels, as shown in Figure 13 (a). LLAMA utilizes 720 varactor diodes (SMV1233), costing $\sim$ \$0.50 each. The total cost of LLAMA for all PCB layers is $\sim$ \$540, resulting in a total cost of $\sim$ \$900. Given economies of scale, the unit cost can be



(a) Prototype. (b) QWP outer layer. (c) QWP inner layer. (d) BFS layer. (e) Evaluation setups.

Figure 13: LLAMA prototype and evaluation setups. (a-d) show the PCBs with diodes embedded in the BFS layer. (e) presents the two experimental setups used in the evaluation. The first is a transmissive configuration where endpoints are placed on either side of LLAMA; the second is a reflective configuration where both endpoints are placed on the same side of LLAMA.

reduced to $2 when there are more than 3000 units per PCB.

Experimental setup. For controlled experiments, we utilize one USRP N210 software-defined radio with a UBX-40 daughterboard as the ISM signal transceiver, operating at a default center frequency of 2.44 GHz. The transmitter and receiver antennas are separated by a specified distance. We experiment with both directional [9] and omni-directional antennas [3]. We configure and control the USRP using the GNU radio software development toolkit [22] run on a PC. The transmitter continuously transmits a cosine waveform over 500 KHz, while the sampling rate of the receiver is 1 MHz. We also evaluate LLAMA using low-cost Wi-Fi and Bluetooth devices (the same setup as benchmark experiments as shown in Figure 2). Additionally, we perform an experiment with a pair of GIGABYTE mini-PCs [4] with Intel 5300 wireless cards, to evaluate performance over a larger frequency range (*i.e.*, 20 MHz, including 52 OFDM subcarriers).

We perform both through-surface (*transmissive*) and surface-reflection (*reflective*) experiments as shown in Figure 13 (e). In transmissive experiments, the metasurface is placed between the transmitter and receiver. In reflective experiments, the transmitter and receiver are placed on the same side of the metasurface. In each experiment, the baseline received signal power without the metasurface is measured by averaging 30 seconds of received samples, and the maximum signal power with the metasurface is obtained after a fast sweep of voltages as detailed in § 3.3. To avoid multipath effects confounding the performance behavior of LLAMA, we cover the test area with RF absorbing material, and use directional antennas by default in USRP-based experiments.

5 Evaluation

In this section, we conduct extensive experiments to evaluate the performance of LLAMA. We first answer how metasurface improves the transmissive signal power in polarization mismatch setup. Then we analyze the relationship between signal enhancement induced by the metasurface across a number of parameters including transmitted power, multipath effect, antenna directionality and operating frequency. We also evaluate LLAMA's performance for practical low-cost IoT communication links. In addition, we validate LLAMA's ability to enhance a reflected signal, and demonstrate the influence of the proposed metasurface structure for sensing.

Performance metrics. We measure signal strength at the receiver as our performance metric, since this directly characterizes the benefit of polarization rotation. An increase in the received power usually translates to a throughput improvement. While it is common to measure link throughput directly, the size limit of our current prototype makes it challenging to characterize link throughput in diverse settings.

5.1 Transmissive Operation

5.1.1 Transmissive Signal Enhancement

To verify LLAMA's ability to rotate the polarization of transmissive signals, we conduct experiments with the metasurface under different transmitter-receiver (Tx-Rx) distances (from 24 cm to 60 cm by half wavelength steps of 6 cm). The transmitter and receiver are placed orthogonally such that they are in a mismatched polarization configuration. In each experiment, we measure the received signal power across a full sweep of voltage combinations (both V_x and V_y vary from $0-30$ V). Figure 14 (a-g) show how the received signal power changes with different voltage combinations at each Tx-Rx distance and how the maximum achieved rotation angle diminishes as the distance becomes comparable to the size of the surface. The signal power changes significantly with changes in biasing voltage. We also find the mapping between these voltages as the rotation shifts gradually with respect to Tx-Rx distance. Figure 14 (h) shows the polarization rotation degree measured by the proposed method in § 3.4. We find that the metasurface can rotate the polarization over a range of $3^\circ-45^\circ$, which allows the metasurface to correct for a significant amount of mismatch. To understand the signal improvements provided by the metasurface, we also measure the signal power in mismatch configuration with no metasurface present as a baseline. By comparing the results with and without the metasurface as depicted in Figure 15, we can see that the metasurface enhances the transmissive signal power by up

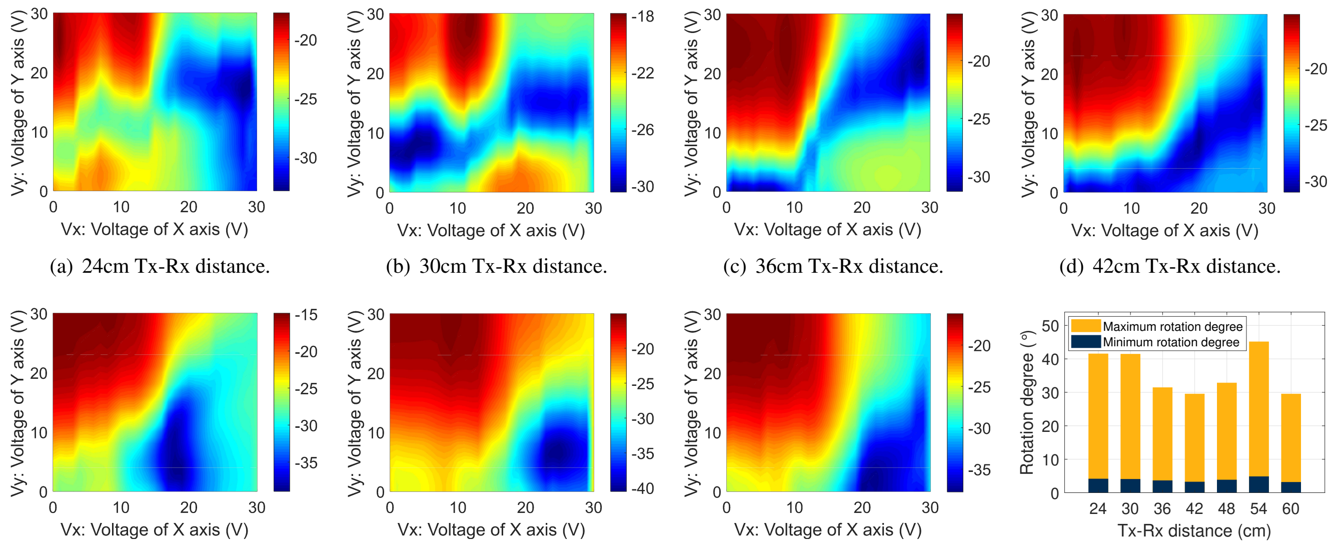

(a) 24cm Tx-Rx distance. (b) 30cm Tx-Rx distance. (c) 36cm Tx-Rx distance. (d) 42cm Tx-Rx distance. (e) 48cm Tx-Rx distance. (f) 54cm Tx-Rx distance. (g) 60cm Tx-Rx distance. (h) Polarization rotation degree.

Figure 14: Measurements with metasurface under polarization mismatch setup. (a-g) show the received signal power heatmap with different voltage combinations. (h) presents the maximum polarization rotation degree caused by metasurface.

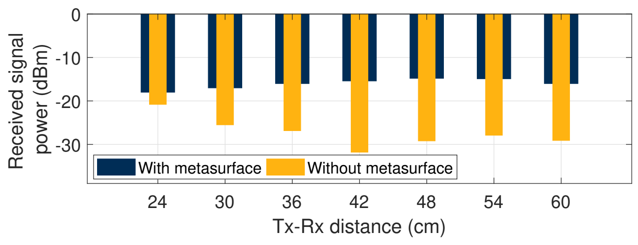


Figure 15: Received signal power with/without metasurface in polarization mismatch setup.

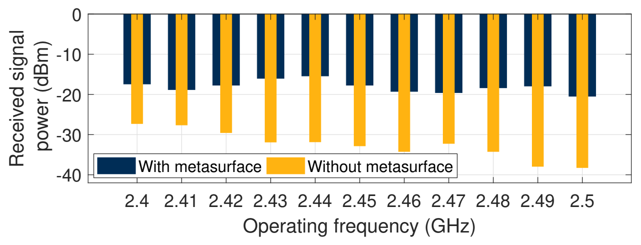


Figure 16: Power improvement VS. operating frequency in polarization mismatch setup.

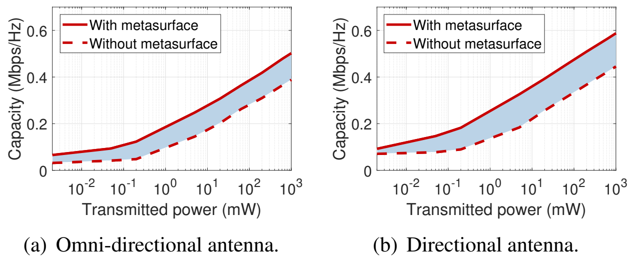
to 15 dB, which extends the potential transmission distance by up to 5.6× according to the Friis equation [21].

5.1.2 Performance Benchmarks

Validating operational bandwidth. In order to evaluate LLAMA's performance over the entire ISM frequency band, we conduct experiments that vary the operating frequency from 2.4 GHz to 2.5 GHz by steps of 0.01 GHz. We measure the maximum signal power with and without the metasurface. From the results shown in Figure 16, we can see that LLAMA enables $>$ 10 dB signal enhancement across the entire ISM frequency band, when compensating for polarization mismatch (orthogonal antenna orientation). This indicates that LLAMA has potential for optimizing IoT communication links with protocols including Wi-Fi, Bluetooth and Zigbee.

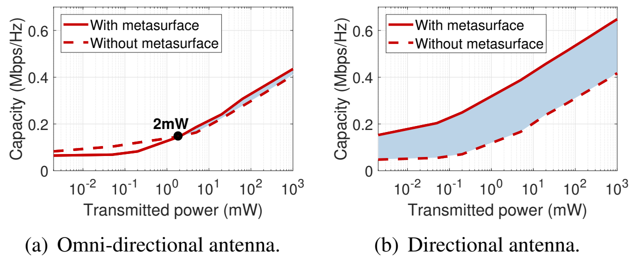
Impact of incident power on performance gains. In this experiment, we sweep through transmit power settings to understand how incident power affects the performance improvement (measured in terms of channel capacity enhancement) provided by the metasurface — lower transmit powers could potentially be dominated by loss within the metasurface. The capacity is calculated according to the SNR measurement and channel bandwidth. We perform experiments with both directional [9] and omnidirectional [3] antennas on the transceiver. Figure 17 shows that the capacity initially increases slowly with transmitting power for both the directional and omnidirectional antennas.

Impact of multipath. In these experiments we seek to understand the impact of multipath propagation on LLAMA's performance. We perform experiments in an indoor lab environment without absorber material. We also measure the channel capacity by using two types of antennas at different transmit power. By comparing the results of Figure 18 with Figure 17, we find that for a directional antenna, the metasurface can still contribute similar capacity improvements to without multipath. However, the results from omni-directional antennas are different – when the transmitted power is lower than 2 mW, the metasurface will no longer enhance the passing signal, and in fact degrade the channel capacity. Directional antennas



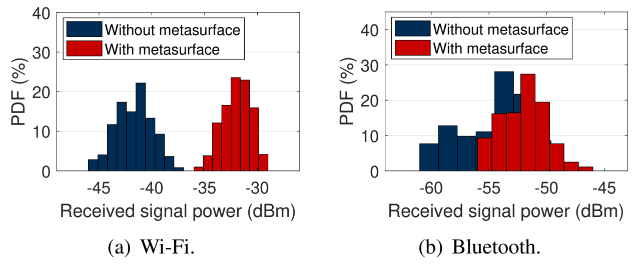
(a) Omni-directional antenna. (b) Directional antenna.

Figure 17: Channel capacity with varying incident power. We eliminate multipath by using RF absorbing material.



(a) Omni-directional antenna. (b) Directional antenna.

Figure 18: Experimental results in rich multipath environment (laboratory) without using RF absorbing material.



(a) Wi-Fi. (b) Bluetooth.

Figure 19: Experimental results of low-cost IoT devices in polarization mismatch setup.

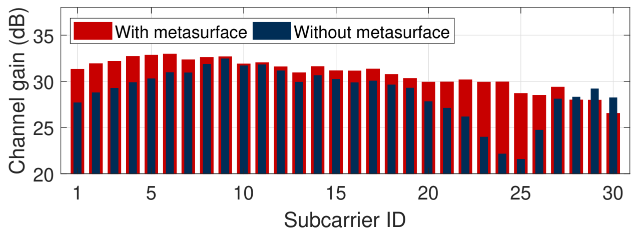


Figure 20: Experimental results of OFDM frequency bins in polarization mismatch setup.

"concentrate" the signals through the metasurface, so the incident power is higher and the metasurface can let more power through; omni-directional antennas do not send as much incident power to the metasurface, so the enhancement from polarization matching may not compensate for the loss through the surface. For that reason, the metasurface can effectively block out some weaker multipath components, and therefore the endpoint receiver will not get constructive interference from these multipath components. In that sense, it is possible the metasurface can reduce performance.

5.1.3 Experiments with Low-cost IoT Devices

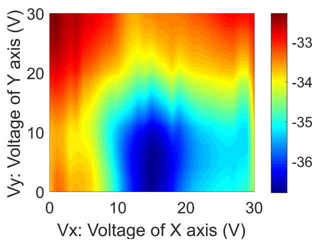
Finally, we evaluate LLAMA's performance with low-cost IoT devices. We perform tests with conventional Wi-Fi and Bluetooth links. The Wi-Fi link is between a Wi-Fi router and an Arduino with a low-cost ESP8266 module, and the Bluetooth link is between a Huawei Watch and a Raspberry Pi 3. From the signal power distributions shown in Figure 19, we find that for the Wi-Fi link, LLAMA creates around 10 dB signal power improvement in a mismatched polarization setup, which looks similar to the matched configuration depicted in Figure 2 (a). While the improvement for the Bluetooth link is lower, this is expected according to the result shown in Figure 18 (a), which indicates the amount of signal quality change that can be affected by LLAMA depends on the incident power level at the metasurface. Nevertheless, we believe LLAMA could still help Bluetooth *receivers* when the transmitter is a higher-power device, such as a mobile handset.

We next study how LLAMA performs when operating across a wider band channel. Most Wi-Fi transmissions today leverage OFDM over $20-40$ MHz. To answer this question, we leverage GIGABYTE mini-PCs equipped with conventional off-the-shelf Intel 5300 wireless cards as transceivers to conduct an experiment – the center frequency and bandwidth are 2.47 GHz and 20 MHz, respectively. Figure 20 shows the channel gain measurements with and without the metasurface in a mismatched configuration. The results show that the overall channel gain is improved, but the enhancement of individual subcarriers are different given the specific bias voltages used for the metasurface; this is consistent with the simulation results shown in Figure 11. The subcarrier channel gains with the metasurface are more consistent over frequency than without. We believe the presence of LLAMA blocks weak multipath signal components that traverse longer paths and tend to exhibit frequency-selectivity more, and the remaining components are aligned through polarization rotation.

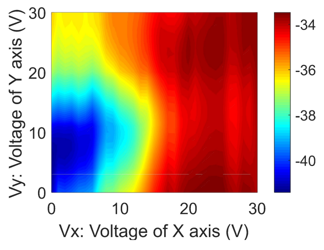
5.2 Reflective Operation

5.2.1 Reflective Signal Enhancement

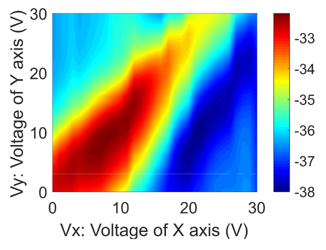
In addition to evaluating transmissive configurations, we also look at LLAMA's effect on reflected signals. We place the transmitter and receiver on the same side of the metasurface, and separate the transmitter and receiver by 70 cm. We perform experiments at different Tx-metasurface distances by moving the metasurface along the vertical line of the transceiver pair. Figure 22 shows the maximum received sig-



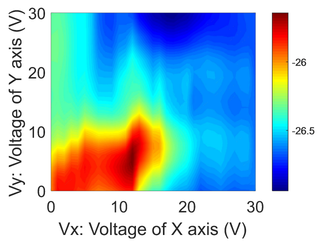
(a) 24cm Tx-Metasurface distance.



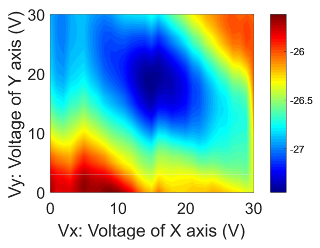
(b) 30cm Tx-Metasurface distance.



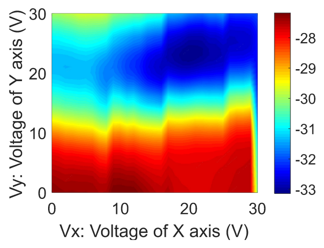
(c) 36cm Tx-Metasurface distance.



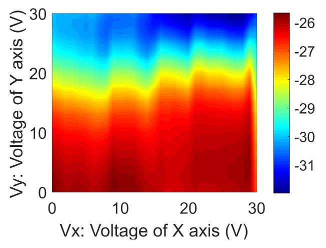
(d) 42cm Tx-Metasurface distance.



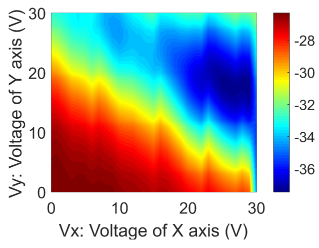
(e) 48cm Tx-Metasurface distance.



(f) 54cm Tx-Metasurface distance.



(g) 60cm Tx-Metasurface distance.



(h) 66cm Tx-Metasurface distance.

Figure 21: Experimental results in reflection scenarios. We find that LLAMA also changes the reflective signal power.

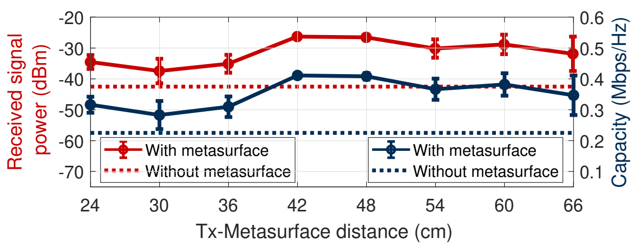


Figure 22: LLAMA provides improvements to channel capacity and power in a reflective configuration.

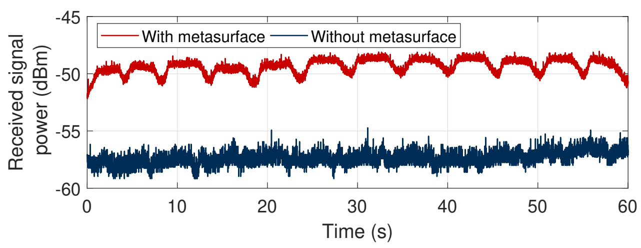


Figure 23: Human respiration sensing results with/without metasurface at low transmitting power of 5 mW.

nal power and channel capacity with the metasurface , as well as the baseline measurements without the metasurface in a mismatched configuration. These results show that LLAMA also has a positive impact on the reflection scenario—the signal power and capacity can be improved with respect to mismatch by as much as 17 dB and 180 kbps/Hz, respectively. However, the signal power difference over voltage combinations (see Figure 21) is much smaller than that in the transmission scenario. We believe this is because the rotation will be cancelled after reflection.

5.2.2 Employing LLAMA for Sensing

Based on the reflective configuration, we conjecture that LLAMA can be utilized to enhance sensing. To validate this, we consider human respiration detection as a case study to test LLAMA's potential. In this experiment, the metasurface is placed 2 m away from the center of the transceiver pair, the human subject is located on the side between the transmitter and the metasurface. First, we remove the metasurface and reduce the transmitting power to where the human subject's respiration can no longer be detected from the received signal. Then we introduce the metasurface at the predetermined location, and measure the received signal strength. The detection results with and without the metasurface are plotted in Figure 23. It is clear that the metasurface can enhance the reflected signal and allow the target's respiration rate to be detectable under a low transmit power configuration (5 mW). We believe that LLAMA can also be extended to other low SNR sensing applications [23, 50].

6 Discussion and Future Work

Scaling to a dense IoT deployment. This work marks the first step towards mitigating polarization issues for individual communication links with a LLAMA prototype. Next, we plan to scale up the size of the metasurface for a larger deployment and explore more challenging multi-link scenarios. When there are multiple IoT devices in different polarization orientations, tuning the signal polarization can lead to a new form of polarization reuse or access control and improve the network throughput of a dense IoT deployment.

Adapting to device mobility. The current search time for optimal voltage is limited by the switching speed of the commodity power supply we used, hence there is still a latency issue in mobility scenarios. In the future, we will look at methods that can speed this up once the relative antenna orientations are determined and then track the changes.

7 Related Work

Broadly speaking, our work is related to work in three areas:
Endpoint optimizations. Most efforts for improving communication quality focuses on controlling the endpoints themselves. For instance, Multiple Input, Multiple Output (MIMO) links leverage multiple antennas to exploit spatial diversity at a sub-link level, while Multi-User MIMO exploits spatial diversity at an inter-link level [24, 25]. Massive MIMO introduces many more antennas at an access point than both radio chains and users, so that the AP may search for a set of antennas that forms a well-conditioned MIMO channel to those users [34, 40, 47]. However, these approaches are fundamentally limited if the cause of performance loss is antenna polarization mismatch between endpoints. At the endpoints, the only directly relevant mitigation strategies are to use either circularly polarized antennas or multiple linearly polarized antennas. Once the antennas are fixed, not much can be done about polarization match at the endpoints. Using an antenna array like massive MIMO can enhance the received signal power, but without directly addressing the polarization issue.

In contrast, an approach that changes the radio environment itself (e.g., deploying low-cost reflectors) offers the possibilities to increase the number of degrees of freedom.
Environment-based optimizations. Previous work on radio environment optimizations fall into two categories: phase-based and amplitude-based. Initial attempts of phase based approaches such as leveraging static mirrors [52] or programmable phased-array reflectors [7, 8, 42] are in the ability of generating constructive propagation paths. These methods focus on millimeter wave links on high frequency bands (*i.e.*, 10 GHz and above). More generally, several proposals argue for dynamically reconfiguring the radio environments [11, 12, 30, 37, 43, 44, 48]. Specifically, recent prototypes manipulate the signal propagation behavior in the 2.4 GHz band, by using a large array of inexpensive antennas [18, 29, 43] or conductive surfaces [16] as phase shifting elements. These systems align phase elements according to a channel decomposition. Amplitude-based designs, such as RFocus [10], sidestep the difficulty in measuring phase. Based on the signal amplitude measurements from the receiver, RFocus configures the signal to either pass through or reflect from the surface element by setting the “on” or “off” state of each element, so that the transmitted signal is focused at the intended receiver.

Orthogonal to prior work that aligns multiple paths to achieve beamforming effects or improve spatial multiplexing efficiency, LLAMA optimizes low-cost IoT communication links by specifically overcoming the pervasive issue of polarization mismatch that affects both single and multi-path communication.
Metamaterials. Metamaterials are an earlier, more general form of metasurfaces that are constructed in 3D rather than 2D. These are artificially constructed with special properties. Recent work in the applied physics community has developed metamaterials that can directly alter existing signals in the environment itself, such as creating materials with a negative refraction index [27] and engineering complex beam patterns [33]. Other work has verified the feasibility of leveraging metamaterials to change the signal polarization [26, 45, 46, 49, 51]. With a biasing network, different voltages are provided to diodes integrated on the metasurface for rotating the polarization of a transmitted wave. While these designs have shown great promise in controlled experiments that quantify performance in a higher frequency band (*i.e.*, > 5 GHz), they were constructed using expensive, low loss substrate materials such as Rogers or F4B. Furthermore, they have not been integrated into an end-to-end system that optimizes signal paths in real time.

In contrast, we present an end-to-end system incorporating the structure of a metasurface design for the 2.4 GHz ISM band using cheaper, but higher loss FR4 material, and specifically control the structure’s polarization rotation to optimize the communication link between a pair of devices.

8 Conclusion

This paper highlights the under-appreciated issue of polarization mismatch for low-cost IoT devices that are physically limited to employing a single low-quality antenna. We present LLAMA, a system designed to mitigate the polarization mismatch without hardware modifications to the endpoints. LLAMA is capable of manipulating the polarization state of the signal arriving at the receiver with a tunable metasurface structure made with cheap material. It can optimize the communication quality in real time, and enhance the performance of sensing applications.

Acknowledgement

We thank our shepherd Fadel Adib, and the anonymous reviewers for their helpful feedback in improving the paper. This work is supported by the National Science Foundation under Grant Nos. CNS-1617161, CNS-1763212 and CNS-1763309, the National Natural Science Foundation of China under Grant Nos. 61672428 and 62061146001, the ShaanXi Science and Technology Innovation Team Support Project under Grant No. 2018TD-026. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] Huawei watch. Website, 2015.

[2] Comxim 360 degree photography turntable. Website, 2020.

[3] Highfine 2 x 2.4 GHz 6 dBi indoor omni-directional Wi-Fi antenna. Website, 2020.

[4] Mini-pc barebone (brix). Website, 2020.

[5] Netgear n300 wifi cable modem wireless router. Website, 2020.

[6] Series 2230G High Power, 3-Channel Programmable Power Supplies. Website, 2020.

[7] O. Abari, D. Bharadia, A. Duffield, D. Katabi. Cutting the cord in virtual reality. *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, 162–168, 2016.

[8] ——. Enabling high-quality untethered virtual reality. *NSDI*, 531–544, 2017.

[9] Alfa Network APA-M25 2.4+5GHz Dual Band Indoor Antenna. Website, 2020.

[10] V. Arun, H. Balakrishnan. RFocus: Practical beamforming for small devices. *NSDI*, 1–12, 2020.

[11] E. Basar, M. Di Renzo, J. De Rosny, M. Debbah, M. Alouini, R. Zhang. Wireless Communications Through Reconfigurable Intelligent Surfaces. *IEEE Access*, **7**, 116,753–116,773, 2019.

[12] E. Björnson, L. Sanguinetti, H. Wymeersch, J. Hoydis, T. L. Marzetta. Massive MIMO is a Reality – What is Next? Five Promising Research Directions for Antenna Arrays. *Digital Signal Processing*, **94**, 2019.

[13] S. M. Bowers, A. Safaripour, A. Hajimiri. Dynamic polarization control. *IEEE Journal of Solid-State Circuits*, **50**(5), 1224–1236, 2015.

[14] J. Chan, S. Raju, R. Nandakumar, R. Bly, S. Gollakota. Detecting middle ear fluid using smartphones. *Science translational medicine*, **11**(492), eaav1102, 2019.

[15] J. Chan, T. Rea, S. Gollakota, J. E. Sunshine. Contactless cardiac arrest detection using smart devices. *NPJ digital medicine*, **2**(1), 1–8, 2019.

[16] J. Chan, A. Wang, V. Iyer, S. Gollakota. Surface mimo: Using conductive surfaces for mimo between small devices. *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, 3–18, 2018.

[17] J. A. Davis, D. E. McNamara, D. M. Cottrell, T. Sonehara. Two-dimensional polarization encoding with a phase-only liquid-crystal spatial light modulator. *Applied Optics*, **39**(10), 1549–1554, 2000.

[18] M. Dunna, C. Zhang, D. Sievenpiper, D. Bharadia. Scattermimo: enabling virtual mimo with smart surfaces. *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, 1–14, 2020.

[19] ESP8266 Wi-Fi Main Board. Website, 2020.

[20] Standard FR4 TG135 Datasheet. Website, 2020.

[21] Friis transmission equation. Website, 2020.

[22] GNURadio. Website, 2020.

[23] U. Ha, J. Leng, A. Khaddaj, F. Adib. Food and liquid sensing in practical environments using rfids. *NSDI*, 1083–1100, 2020.

[24] E. Hamed, H. Rahul, M. A. Abdelghany, D. Katabi. Real-time distributed MIMO systems. *SIGCOMM*, 412–425, 2016.

[25] E. Hamed, H. Rahul, B. Partov. Chorus: Truly distributed distributed-MIMO. *SIGCOMM*, 461–475, 2018.

[26] J. Hao, Y. Yuan, L. Ran, T. Jiang, J. A. Kong, C. Chan, L. Zhou. Manipulating electromagnetic wave polarizations by anisotropic metamaterials. *Physical review letters*, **99**(6), 063,908, 2007.

[27] I. Kourakis, P. Shukla. Nonlinear propagation of electromagnetic waves in negative-refraction-index composite materials. *Physical Review E*, **72**(1), 016,626, 2005.

[28] B. Kress, P. Meyrueis. *Digital diffractive optics: An introduction to planar diffractive optics and related technology*, 2000.

[29] Z. Li, Y. Xie, L. Shangguan, R. I. Zelaya, J. Gummeson, W. Hu, K. Jamieson. Towards programming the radio environment with large arrays of inexpensive antennas. *NSDI*, 285–300, 2019.

[30] C. Liaskos, S. Nie, A. Tsioliaridou, A. Pitsillides, S. Ioannidis, I. Akyildiz. A New Wireless Communication Paradigm through Software-Controlled Metasurfaces. *IEEE Communications Magazine*, **56**(9), 162–169, 2018.

[31] R. Nandakumar, V. Iyer, D. Tan, S. Gollakota. Fingerio: Using active sonar for fine-grained finger tracking. *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, 1515–1525, 2016.

[32] M. Okatan, J. Mantese, S. Alpay. Polarization coupling in ferroelectric multilayers. *Physical Review B*, **79**(17), 174,113, 2009.

[33] A. A. Orlov, P. M. Voroshilov, P. A. Belov, Y. S. Kivshar. Engineered optical nonlocality in nanostructured metamaterials. *Physical Review B*, **84**(4), 045,424, 2011.

[34] H. S. Rahul, S. Kumar, D. Katabi. JMB: Scaling wireless capacity with user demands. *SIGCOMM CCR*, **42**(4), 235–246, 2012.

[35] R. S. Rao. *Microwave engineering*. PHI Learning Pvt. Ltd., 2015.

[36] Raspberry Pi 3 Model B+. Website, 2020.

[37] M. D. Renzo, M. Debbah, D.-T. Phan-Huy, A. Zappone, M.-S. Alouini, C. Yuen, V. Sciancalepore, G. C. Alexandropoulos, J. Hoydis, H. Gacanin, J. de Rosny, A. Bounceur, G. Lerosey, M. Fink. Smart radio environments empowered by reconfigurable AI meta-surfaces: An idea whose time has come.

EURASIP Journal on Wireless Communications and Networking volume, 2019.

[38] Rogers corporation. Website, 2017.

[39] N.-H. Shen, M. Kafesaki, T. Koschny, L. Zhang, E. N. Economou, C. M. Soukoulis. Broadband blueshift tunable metamaterials and dual-band switches. *Physical Review B*, **79**(16), 161,102, 2009.

[40] C. Shepard, H. Yu, N. Anand, E. Li, T. Marzetta, R. Yang, L. Zhong. Argos: Practical many-antenna base stations. *MobiCom*, 53–64, 2012.

[41] D. Shrekenhamer, W.-C. Chen, W. J. Padilla. Liquid crystal tunable metamaterial absorber. *Physical review letters*, **110**(17), 177,403, 2013.

[42] X. Tan, Z. Sun, J. M. Jornet, D. Pados. Increasing indoor spectrum sharing capacity using smart reflect-array. *ICC*, 1–6. IEEE, 2016.

[43] A. Welkie, L. Shangguan, J. Gummeson, W. Hu, K. Jamieson. Programmable radio environments for smart spaces. *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, 36–42, 2017.

[44] Q. Wu, R. Zhang. Towards Smart and Reconfigurable Environment: Intelligent Reflecting Surface Aided Wireless Network. *IEEE Communications Magazine*, **58**(1), 106–112, 2020.

[45] Z. Wu, Y. Ra'di, A. Grbic. Tunable metasurfaces: A polarization rotator design. *Physical Review X*, **9**(1), 011,036, 2019.

[46] H. Yang, X. Cao, F. Yang, J. Gao, S. Xu, M. Li, X. Chen, Y. Zhao, Y. Zheng, S. Li. A programmable metasurface with dynamic polarization, scattering and focusing control. *Scientific reports*, **6**, 35,692, 2016.

[47] Q. Yang, X. Li, H. Yao, J. Fang, K. Tan, W. Hu, J. Zhang, Y. Zhang. BigStation: Enabling scalable real-time signal processing in large MU-MIMO systems. *SIGCOMM CCR*, **43**(4), 399–410, 2013.

[48] Q. Zhang, Y.-C. Liang, H. V. Poor. Large Intelligent Surface/Antennas (LISA) Assisted Symbiotic Radio for IoT Communications, 2020.

[49] X. G. Zhang, Q. Yu, W. X. Jiang, Y. L. Sun, L. Bai, Q. Wang, C.-W. Qiu, T. J. Cui. Polarization-controlled dual-programmable metasurfaces. *Advanced Science*, 1903382, 2020.

[50] M. Zhao, F. Adib, D. Katabi. Emotion recognition using wireless signals. *MobiCom*, 95–108, 2016.

[51] Y. Zheng, Y. Zhou, J. Gao, X. Cao, H. Yang, S. Li, L. Xu, J. Lan, L. Jidi. Ultra-wideband polarization conversion metasurface and its application cases for antenna radiation enhancement and scattering suppression. *Scientific reports*, **7**(1), 1–12, 2017.

[52] X. Zhou, Z. Zhang, Y. Zhu, Y. Li, S. Kumar, A. Vahdat, B. Y. Zhao, H. Zheng. Mirror mirror on the ceiling: Flexible wireless links for data centers. *SIGCOMM CCR*, **42**(4), 443–454, 2012.

Bootstrapping Battery-free Wireless Networks: Efficient Neighbor Discovery and Synchronization in the Face of Intermittency

Kai Geissdoerfer
TU Dresden

Marco Zimmerling
TU Dresden

Abstract

Due to their favorable size, cost, and sustainability, battery-free devices are preferable in various applications. However, battery-free devices operate only intermittently since ambient energy sources, such as light and radio-frequency signals, are often too weak to continuously power the devices. This paper addresses the unsolved problem of efficient device-to-device communication in the face of intermittency. We present Find, the first neighbor discovery protocol for battery-free wireless networks that uses randomized waiting to minimize discovery latency. We also introduce Flync, a new hardware/software solution that synchronizes indoor light harvesting nodes to powerline-induced brightness variations of widely used lamps, which we exploit to further speed up neighbor discovery. Experiments with an open-source prototype built from off-the-shelf hardware components show that our techniques reduce the discovery latency by 4.3× (median) and 34.4× (99th percentile) compared with a baseline approach without waiting.

1 Introduction

Despite technological advances, the maintenance costs and environmental impact of batteries remain a major threat to the vision of a truly ubiquitous Internet of Things [3, 11]. *Battery-free devices* that store energy harvested from light, vibrations, radio-frequency (RF) signals, and other ambient sources in a capacitor are one of the most viable alternatives today [45]. Capacitors store electrical energy in an electrical field rather than in the form of chemical energy, and thus have negligible aging effects and are sustainable [1, 6]. Moreover, their favorable size, weight, and cost points enable new applications where batteries would be inconvenient or infeasible [30].

Challenge. The power that can be harvested from ambient energy sources can vary significantly across time and space [15], and is often too weak to directly power a battery-free node, such as a smart sensor [32]. Thus, as illustrated in Fig. 1 and further discussed in detail in Sec. 7, a battery-free device first needs to buffer sufficient energy in its capacitor before it can operate for a short period of time; then the device turns off

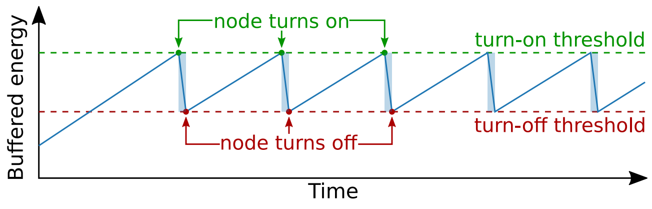


Figure 1: Because ambient power is often weak, a battery-free node must buffer energy before it can wake up and operate for a short time period. This is known as intermittent operation.

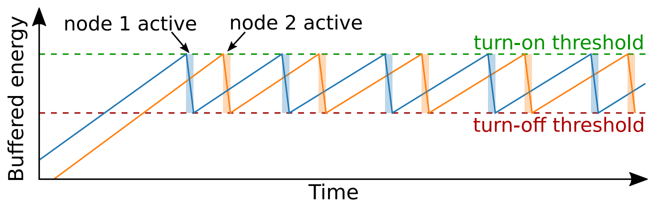
until the capacitor is sufficiently charged again. As a result, battery-free devices operate *intermittently*.

Intermittency is in stark contrast to conventional duty cycling. While duty cycling is intentionally introduced to save energy and thus predictable, intermittency is mainly dictated by uncontrollable environmental factors and thus impacts the device operation in unpredictable ways. The resulting challenges in terms of, for example, reliable time keeping [12, 18] or ensuring application progress and data consistency [8, 34] have been widely studied in the recent literature.

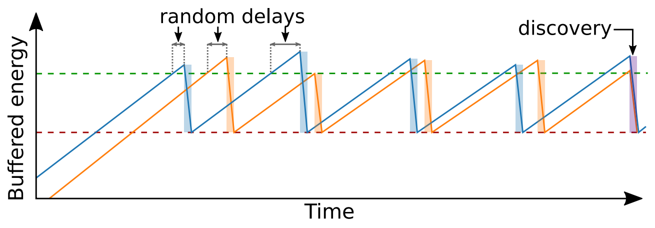
The impact of intermittency on wireless networking has instead received little attention. Just like in conventional battery-supported networks, direct communication between battery-free devices is desirable, for example, to increase the availability of the system [36], to enable novel applications [20, 32], and to reduce infrastructure costs [35]. However, to communicate with one another, sender and receiver must be active simultaneously for at least the airtime of one complete packet. This is challenging in battery-free networks for three reasons:

1. Battery-free nodes can only become active when they have accumulated sufficient energy in their capacitors.
2. They may only be active for a short period, which renders excessive sampling of the wireless channel infeasible.
3. Their duty cycles are often low and may change unpredictably due to varying availability of ambient energy.

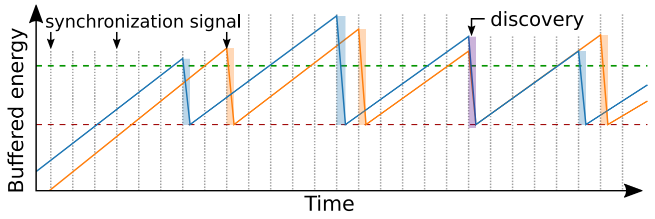
For example, our prototype battery-free node needs to charge its capacitor for hundreds of milliseconds to sustain 1 ms of activity when harvesting from indoor light. Because the short



(a) Battery-free nodes may need a long time to discover each other due to low duty cycles and the interleaving of short activity phases.



(b) Using Find, nodes randomly delay their wake-ups to avoid interleaving, thereby discovering each other faster and more efficiently.



(c) Using Find + Flync, nodes implicitly align their wake-ups to an external synchronization signal, further accelerating discovery.

Figure 2: Illustration of the battery-free neighbor discovery challenge in (a) and of our proposed mechanisms to address it in (b) and (c).

activity phases of different nodes are generally interleaved, as shown in Fig. 2a, it takes a long time until nodes encounter each other. And this is not a one-time endeavor: While nodes may attempt to synchronize their activity phases at the first encounter, they lose track of time during extended periods without energy [12, 18], which forces them to re-synchronize.

This challenge is fundamental and pertains to battery-free networks regardless of the type of wireless communication: While backscatter communication can lower the energy costs compared to active radio communication, sender and receiver still need to have sufficient energy at the same time. Prior work on backscatter has primarily focused on pushing the envelope of communication range and throughput, avoiding intermittency by evaluating the designs under high ambient energy availability [20, 32] or by powering the devices via USB or batteries to not disturb the measurements [35]. To our knowledge, direct radio communication between real battery-free devices has not been explored so far, as the overhead due to intermittency is considered too demanding [36].

Contribution. We set out to bootstrap battery-free wireless networks by presenting two mechanisms that enable battery-free nodes to discover each other quickly and efficiently.

The first mechanism, Find, is a neighbor discovery protocol. As illustrated in Fig. 2b, the key idea behind Find is to address the interleaving problem by introducing random delays after the devices have sufficiently charged their capacitors before becoming active. We develop analytical models to determine an optimized delay distribution that minimizes discovery latency. At runtime, each Find node dynamically adapts the delay distribution to changes in its energy availability.

The second mechanism, Flync, is a hardware/software solution that further speeds up the discovery process. Flync phase-synchronizes solar energy harvesting devices to powerline-induced flicker of state-of-the-art lamps; the proposed circuit draws only 5 µW of power. As shown in Fig. 2c, using Find together with Flync, nodes can implicitly align their activity phases to this external synchronization signal, dramatically increasing their chances to be active at the same time.

We prototype our mechanisms on a custom-designed ultra low-power battery-free node. It is based on a state-of-the-art microcontroller (MCU) with a 2.4 GHz Bluetooth Low Energy (BLE) radio, and buffers energy harvested via three small solar panels in a tiny 47 µF ceramic capacitor.

We use 6 of our prototype battery-free nodes to conduct extensive experiments and a contact-tracing case study. We summarize our key findings as follows:

- Find provides shorter discovery latencies than greedy and naïve random node activations. Find + Flync improves on greedy by 4.3× in terms of the median latency (141 s vs. 604 s); the 99th percentile improvement is 34.4×.
- Our hardware prototype works with 14 out of 19 fluorescent, halogen, and light emitting diode (LED) lamps we tested, demonstrating that Flync is broadly applicable in indoor environments. Flync provides a stable clock signal when nodes are deployed across different rooms, carried around, or exposed to temporary shadowing.
- We conduct a contact-tracing case study in an open-air pub with Find and in an office kitchen using Find + Flync. The median time between consecutive encounters of the same two nodes is 1.5 s and 7.5 s in the outdoor and indoor environment, respectively. This shows the potential of our battery-free designs for real-world applications.

Overall, this paper makes the following contributions:

- Find, the first neighbor discovery protocol for battery-free networks. Find is agnostic to the energy harvesting modality and the type of wireless communication.
- Flync, the first solution extracting a stable clock from solar harvesting current, whose amplitude changes due to powerline-induced flicker of state-of-the-art lamps. While we use Flync in tandem with Find to speed up discovery in indoor scenarios, Flync is useful for other purposes and also applicable to battery-supported nodes.
- A novel battery-free node design including an implementation of an efficient intermittent runtime.
- Empirical evidence that the proposed techniques work well under a diverse set of real-world conditions.

2 Battery-free Neighbor Discovery

This section presents the design of Find, the first neighbor discovery protocol for battery-free wireless networks. Find empowers battery-free nodes to quickly discover each other's presence despite intermittent operation and varying ambient energy availability. It is agnostic as to how the nodes harvest energy (from solar, vibrations, RF, etc.) and as to whether they communicate using backscatter or radio communication.

The design of Find is based on the observation that the only way battery-free nodes can reliably avoid interleaving is to not wake up and become active immediately after reaching the minimum energy level required to do so. We refer to this as the greedy approach. Instead, Find delays each wake-up for a random time. A crucial question is how to choose this random delay to ensure fast and energy-efficient discovery.

To answer this question, we devise a model that captures the impact of key parameters, such as the charging time needed to reach the minimum energy level and the random delay, on the discovery latency (Sec. 2.1). Using this model, we then determine an optimized delay distribution that minimizes the discovery latency (Sec. 2.2). Finally, we describe how these considerations materialize in the practical design of the Find protocol and its runtime operation (Sec. 2.3).

2.1 Modeling Discovery Latency

Suppose that a node needs to charge for c slots until it reaches the minimum energy level required to be active for one slot. Let k_0 denote the first slot in which a node reaches the minimum energy level. Using Find, a node waits for a random delay x in units of slots before it wakes up and becomes active. We model x as a discrete random variable X with probability mass function (pmf) $p_X(x)$. During an active slot, a node fully depletes its energy storage. The probability that a node becomes active for the first time in slot k is given by

$$p_{wk,0}(k) = p_X(k - k_0) \tag{1}$$

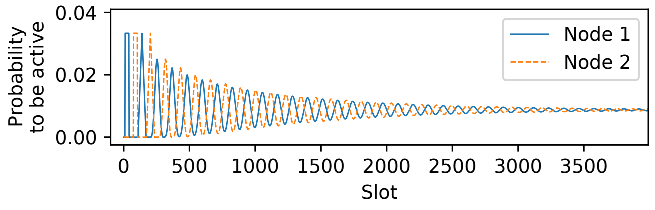
Afterward, a node needs to recharge for c slots before it can become active again. The time of the second wake-up is the sum of the time of the first wake-up, the charging time, and the second random delay. The same reasoning applies recursively to all future wake-up times. Because the random delay is independently chosen across all wake-ups, we can use a recursive convolution to determine the probability that a node wakes up for the n-th time in the k-th slot

$$p_{wk,n}(k) = (p_{wk,n-1} * p_X)(k - c) \tag{2}$$

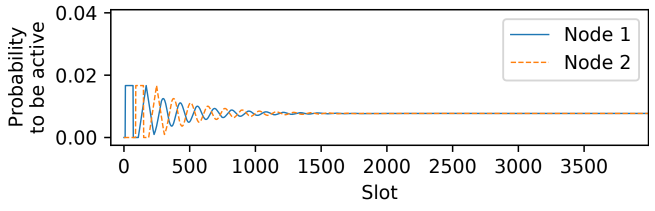
By summing over $n \to \infty$ we obtain the probability that a node is active in slot k

$$p_a(k) = \sum_{n=0}^{\infty} p_{wk,n}(k) \tag{3}$$

To model discovery latency, we consider a fully connected network of N nodes (i.e., a clique of size N). Using a suitable



(a) Random delay drawn from $X \sim U[0,30]$.



(b) Random delay drawn from $X \sim U[0,60]$.

Figure 3: Probability of being active in a slot for two nodes with identical charging times but an initial offset in their wake-ups. The more wide-spread the random delay, the faster nodes break up their interleaved wake-up pattern at the cost of a lower average duty cycle.

sequence of message exchanges in active slots (see Sec. 2.3), one of the $M = N(N-1)/2$ bi-directional links $i \leftrightarrow j$ is discovered if nodes i and j are active in the same slot while all other nodes in the network are inactive. Otherwise, a collision occurs and no link is discovered, a typical assumption in neighbor discovery protocols [24]. The probability that link $i \leftrightarrow j$ is discovered within k slots is the complement of the probability that the link is not discovered in slots $0, \ldots, k$:

$$c_{i \leftrightarrow j}(k) = 1 - \prod_{\kappa=0}^{k} \left(1 - p_{a,i}(\kappa) \cdot p_{a,j}(\kappa) \cdot \prod_{l \neq i,j} (1 - p_{a,l}(\kappa)) \right) \tag{4}$$

$c_{i \leftrightarrow j}(k)$ can be regarded as the cumulative distribution function (cdf) of the discrete random variable describing the slot in which link $i \leftrightarrow j$ is discovered. With $p_{i \leftrightarrow j}(k)$ denoting the corresponding pmf, we compute the expected fraction of links discovered up to slot k by averaging $p_{i \leftrightarrow j}(k)$ over all M links

$$d(k) = \frac{1}{M} \sum_{i \leftrightarrow j} p_{i \leftrightarrow j}(k) \tag{5}$$

If the nodes' charging times are finite, $d(k)$ is a valid cdf, and we define the discovery latency as

$$T_{nd} = \sum_{k=0}^{\infty} (1 - d(k)) \tag{6}$$

2.2 Optimized Delay Distribution

With the above model we are able to get a better understanding of how nodes should delay their wake-ups to help discovery.

Example. Suppose two nodes i and j with the same charging time of $c = 100$ slots, but different slots k_0 in which they reach

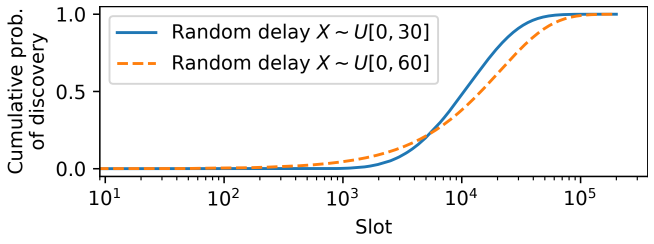


Figure 4: Cumulative distribution function of the slot in which two nodes discover each other, for the two delay distributions in Fig. 3. A more wide-spread delay performs better initially, but leads to lower performance in the long run due to a lower average duty cycle.

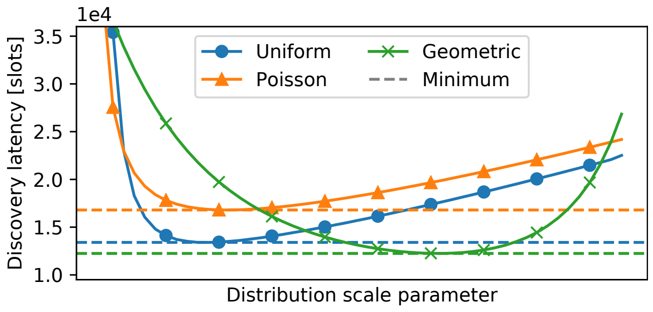


Figure 5: Discovery latency against scale parameter for three different probability distributions. The geometric distribution performs best as it yields delays with high randomness and low mean.

the minimum energy level for the first time (i.e., initial offset). Using (3) we plot in Fig. 3a for both nodes the probability of being active in a slot when they pick random delays from the discrete uniform distribution $X \sim U[0,30]$. We see that in the first thousand slots there is hardly any overlap in the activity of the nodes: Due to the initial offset, node i is likely active when node j is powered off, and vice versa. The probability of being active smears out over time and converges to an average duty cycle of $1/(c+\mathrm{E}[X]) \approx 0.0087$. Fig. 3b plots the same when the two nodes pick random delays from $X \sim U[0,60]$. Compared to Fig. 3a we find that the probability of being active smears out sooner as nodes tend to choose more wide-spread delays. However, as nodes also tend to pick longer delays, they have a lower average duty cycle of 0.0077.

Fig. 4 directly compares the two delay distributions by plotting the cdf of the slot in which nodes i and j discover each other according to (4). We observe that the more wide-spread delay induced by the second distribution $X \sim U[0,60]$ initially provides a higher probability of discovery. In the long run, however, the higher average duty cycle of the first distribution $X \sim U[0,30]$ leads to a higher probability of discovery.

Choosing a distribution. The above example suggests that a non-negative delay distribution with *high randomness and low mean* is preferable. Entropy is a commonly used measure of randomness. Maximizing the entropy of a general non-negative distribution with a given mean yields the exponential distribution [38]. Thus, in Find, we draw random delays from the geometric distribution, the discrete analogue of the exponential distribution, with scale parameter $1/r$ and pmf $(1-r)^k r$ for $k \in \{0,1,2,\dots\}$.

To confirm our reasoning, we compare the geometric distribution against other well-known distributions, namely the discrete uniform distribution and the Poisson distribution. We sweep the scale parameter of the three distributions and compute the discovery latency using (6) for the two-node case, where nodes i and j have equal charging times (25, 100, 500, or 1000 slots). We find that the geometric distribution achieves the lowest discovery latency across all charging times. Fig. 5 shows the resulting curves for a charging time of 100 slots. The differences in the minimum discovery latencies are relatively small. One reason for this is that, according to the central limit theorem, the probability that a node wakes up for the n-th time in slot k converges to a normal distribution for large n, irrespective of the underlying delay distribution.

Determining optimized distribution parameters. Having chosen a suitable delay distribution, we now turn to the problem of determining the scale parameter that minimizes the discovery latency. To formally state the optimization problem, we consider the worst case in terms of discovery latency: all N nodes have the same charging time c, and their initial wake-up times $k_{0,i}$ are all interleaved as in Fig. 3, that is,

$$k_{0,i} = i \cdot \frac{c + 2\,\mathrm{E}[X]}{N} \tag{7}$$

where i is the node index and $\mathrm{E}[X]$ is the expected delay. For specific N and c, we minimize the discovery latency given by (6) and the initial offsets given by (7)

$$\min_r T_{nd}(N,c) \tag{8}$$

Numerical evaluation suggests that $T_{nd}(N,c)$ is convex (see Fig. 5) and hence straightforward to optimize. We use Brent's method [9] to approximate the scale parameter $1/r^*$ that minimizes the discovery latency. The next section explains how we adapt the scale parameter at runtime on a real node.

2.3 Practical Protocol Design

The above analysis makes a number of simplifying assumptions that do not hold in practice. For example, the charging times are generally different across nodes and vary over time. A node typically only knows its own charging time c and is unaware of the total number N of nodes in the network.

Nevertheless, prior work has shown that neighboring nodes have similar energy availability because they harvest energy from the same ambient source(s) [4, 15]. Thus, *in the absence of any prior information*, a reasonable approach for a node is to assume that its neighbors harvest the same amount of energy and thus have the same charging time c like itself.

Moreover, we found that knowledge of the number of nodes N is often not required: optimizing for the case of a two-node

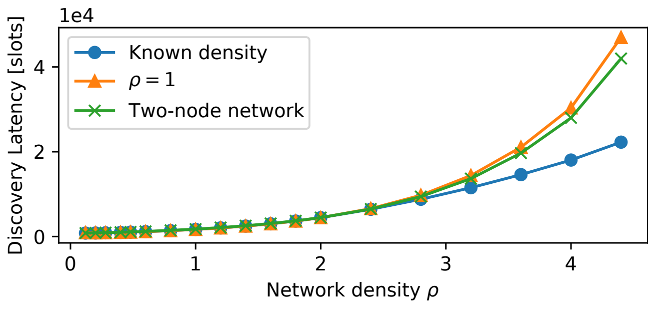


Figure 6: Discovery latency against network density ρ when optimizing for the known density, for a fixed density of ρ = 1, and for a two-node network. For ρ ≤ 2.5, all approaches perform similarly.

network yields competitive performance across a wide range of network densities. In other words, in practice it is often sufficient for a node to assume that it is has only one neighbor (although over time it may discover that it has many more). To understand why, we plot in Fig. 6 the discovery latency for a charging time of 25 slots when optimizing for (*i*) the known network density $\rho = N/c$, (*ii*) a fixed network density of $\rho = 1$, and (*iii*) a two-node network. We can see that for a network density of $\rho \leq 2.5$ the three approaches achieve almost the same performance. For realistic charging times, the network density rarely exceeds this threshold. For example, based on the charging times and beacon length in our real-world case study (see Sec. 6), a network density of $\rho = 2$ would require a network of around 4000 fully connected nodes.

Runtime operation. Prior to each wake-up, a Find node samples a geometric distribution to determine the random delay. A node dynamically adapts the scale parameter of the distribution to changes in its charging time, under the assumption that it has one neighbor with the same charging time, as explained above. To achieve an efficient runtime operation, we store a look-up table of optimized scale parameters in non-volatile memory and use inverse transform sampling to convert samples from a uniform pseudo-random number generator to the optimized, geometric distribution.

Frame structure. Taking inspiration from existing neighbor discovery protocols for battery-powered sensor nodes [5, 13], we adopt the frame structure shown in Fig. 7. During each active slot, a node first transmits a beacon, then listens for potential beacons from neighboring nodes, and finally transmits another beacon at the end of the slot. The second beacon ensures that nodes can discover bi-directional links in one common active slot. Specifically, if the slot offset $\mathcal{T}$ between two nodes (see Fig. 7) is uniformly distributed between $-T_{slot}/2$ and $T_{slot}/2$, where T_{slot} is the slot length, the probability that two nodes successfully discover each other's presence is

$$p = 1 - \frac{2 \cdot (T_{ta} + T_{tx})}{2 \cdot (T_{ta} + T_{tx}) + T_{rx}} \quad (9)$$

Here, as depicted in Fig. 7, T_{tx}, T_{rx}, and T_{ta} denote the times

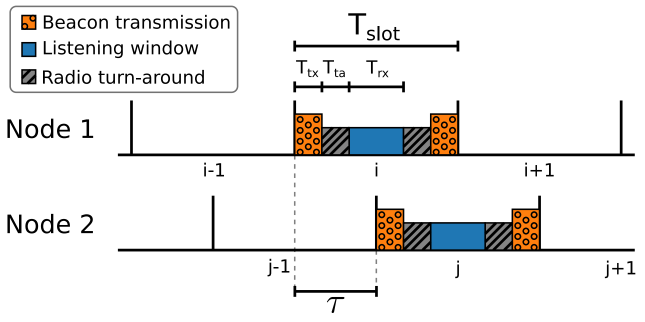


Figure 7: Find's frame structure specifying the sequence of beacon transmissions and the intermediate listening window during an active slot. Using our prototype implementation, nodes can successfully discover each other if the slot offset $\mathcal{T}$ is between 88 µs and 848 µs.

needed to transmit a beacon, to listen for potential beacons, and to switch from receive to transmit mode (or vice versa). In order to maximize the success probability according to (9), Find keeps the beacon transmission time T_{tx} as short as possible to maximize the listening window T_{rx}.

3 Further Accelerating Neighbor Discovery

Find provides fast and energy-efficient neighbor discovery in battery-free networks. Nevertheless, if the ambient energy availability is low, discovery may still take a long time due to the low duty cycles. For example, according to our model, under dim indoor light conditions it takes on average 8 min until two of our prototype battery-free nodes (see Sec. 4) discover each other. Similar observations are to be expected when nodes harvest from weak RF signals or miniature vibrations [7]. The discovery latencies in those challenging energy environments can be prohibitively long for many applications.

This section introduces an approach that facilitates, according to our model, a 10× speed-up in the above-mentioned scenario, allowing two nodes to discover each other in 45 s on average instead of 8 min at an additional cost of only 5 µW. The underlying idea is that neighboring nodes harvest energy from the same ambient source(s) and may therefore have access to a common energy signal that can be used as a time reference. In combination with Find, nodes can exploit this common time reference to align their wake-ups, thereby increasing the chances that nodes are active in the same slot.

To assess the potential of this idea, we focus in this work on harvesting energy from indoor light. While this is a popular method for powering battery-free nodes due to the ubiquity of interior lamps, the energy density of indoor light is significantly lower than that of sunlight. As such, it represents both a challenging environment for battery-free neighbor discovery and a highly relevant setting for real applications. In the following, we provide answers to three key questions:

1. What common energy signal can nodes use? (Sec. 3.1)
2. How to efficiently extract a time reference? (Sec. 3.2)
3. How to exploit this for faster discovery? (Sec. 3.3)

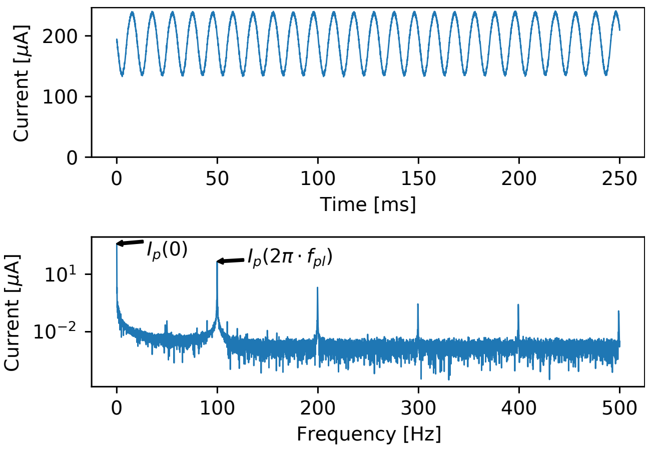


Figure 8: Time and frequency domain of solar panel current when harvesting energy from light emitted by a UP-PL30120-45W LED panel. The current varies with double the powerline frequency.

3.1 Powerline Flicker in Solar Current

When harvesting energy from indoor light, we observed that the solar panel current varies with double the powerline frequency (50 or 60 Hz depending on the region). As an example, Fig. 8 shows the solar panel current when harvesting energy from an LED panel light found in a typical office space.

Practically all indoor lamps are connected to mains power, which induces phase-synchronized brightness variations (*powerline flicker*) of the lamps through different effects. Despite their relatively high inertia, the alternating current through the filament of incandescent and halogen lamps causes temperature and, as a result, brightness variations. A similar effect occurs in gas-discharge lamps like the ubiquitous fluorescent lamps, where the alternating current through the gas modulates the brightness. Due to the exponential relation between forward voltage and brightness, voltage-controlled LEDs are also sensitive to residual ripple of the rectified supply voltage. Because the power available from a solar panel is proportional to the brightness of the incident light, it also varies with double the powerline frequency, as visible in Fig. 8.

To assess the potential of using powerline flicker as a common energy signal, we characterize the magnitude of powerline frequency induced fluctuations of the solar panel current for a wide variety of lamps. To compare lamps across diverse average brightness levels, we define the *flicker index FI* as the ratio of the amplitude of the powerline frequency component and the DC component of the solar panel current i_p

$$FI = \frac{I_p(2\pi \cdot f_{pl})}{I_p(0)} \tag{10}$$

where $I_p(\omega) = \mathcal{F}\{i_p(t)\}$ is the Fourier transform of the solar panel current and f_{pl} is the powerline frequency.

We attach an IXYS SM141K06L solar panel to a Shepherd node [15] and record 15 s of solar panel current at a sampling

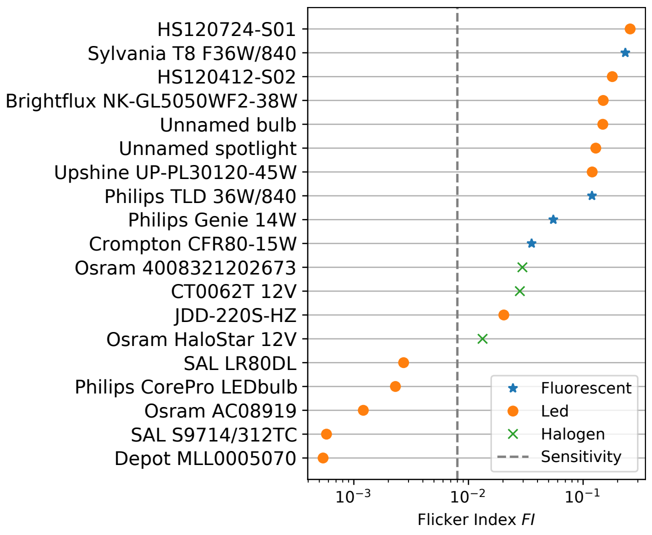


Figure 9: Flicker index for 19 tested lamps. The gray line marks the sensitivity of our Flync prototype. The proposed circuit works with all fluorescent and halogen lamps and the majority of LED lamps.

frequency of 100 kHz from each of the 19 lamps in Fig. 9. For each trace we compute the flicker index using (10). The results in Fig. 9 show that all lamps we tested exhibit varying levels of powerline flicker. We observe that all fluorescent and halogen lamps have a relatively large flicker index. The results for the tested LED lamps are more ambiguous. We suspect that highly integrated, bulb-shaped LED lamps tend to have high-quality current-controlled drivers with little flicker, whereas commercial panel-style LED lamps often rely on voltage-controlled drivers with significant levels of flicker.

We conclude that most types of lamps exhibit significant powerline flicker, which makes this an attractive common energy signal. Next, we present our design of Flync, a hardware/software solution that extracts a frequency- and phase-synchronized clock signal from this common energy signal on distributed battery-free nodes. The dashed line in Fig. 9 is the measured sensitivity (see Sec. 5.2) of our Flync prototype, showing that the proposed design works with all fluorescent and halogen lamps and the majority of tested LED lamps.

3.2 Extracting a Clock from Solar Current

To be viable, Flync needs to provide a stable clock signal while keeping the required energy costs as low as possible.

Hardware. We propose the circuit shown in Fig. 10, which converts the modulated current signal from the solar panel into a digital clock signal that can be connected to a general purpose input/output (GPIO) pin of a MCU. The current through shunt resistor R_S causes a voltage drop that is filtered with a narrow-band bandpass filter to extract and amplify the powerline frequency component. We tune the band-pass filter to a gain of 36 dB at a center frequency of exactly double the powerline frequency, taking into account the limited gain-

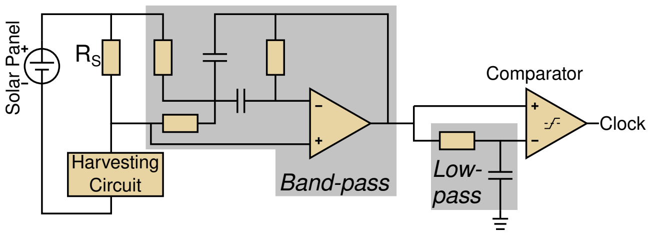


Figure 10: Flync circuit to extract a clock signal from the powerline-induced solar panel current variations (see Fig. 8 for an example).

bandwidth product of the low-power operational amplifier. The resulting signal is connected to a comparator directly and through a low-pass filter to convert it into a digital signal.

The TI TLV521 operational amplifier used in the band-pass filter has a typical current draw of 350 nA, and the TLV7031 comparator has a typical current draw of 315 nA. Including the losses over the 300 Ω shunt resistor, the Flync circuit draws a total of around 5 µW under typical harvesting conditions. This is orders of magnitude lower than the power draw of related approaches, using a light sensor and an analog-to-digital converter (ADC) (5.394 mW [31]) or an antenna to extract the signal from powerline radiation (300 µW [42]).

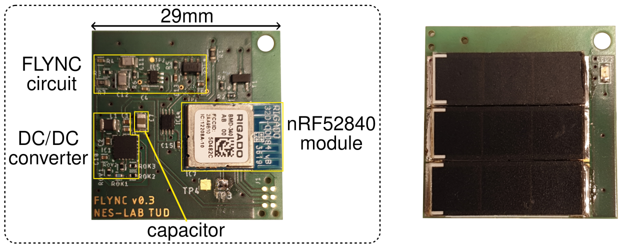
Software. To achieve a stable clock signal, we use a phase-locked loop (PLL) in combination with a proportional integral derivative (PID) controller to synchronize the MCU's real-time clock (RTC) to the powerline frequency signal extracted with our proposed circuit. In Sec. 4.2, we describe our software implementation of Flync in more detail.

3.3 Exploiting the Clock for Faster Discovery

Using Flync, neighboring battery-free nodes have access to a common clock. Nodes can use the phase information of this clock to implicitly agree on times at which they potentially become active. For the powerline flicker, this could be the rising edges of the solar panel current (see Fig. 8).

When using Find without Flync, we set the slot length to the duration of a node's active period. When using Find with Flync, we increase the slot length to $1/(2 \cdot f_{pl})$ and let nodes only become active at the beginning of a slot. This increases the probability that nodes become active in the same slot. For example, consider two nodes that randomly and uniformly wake up once within a 1 s time window. Using a slot length of 1 ms, the probability that both nodes wake up in the same slot is 1/1000. With a slot length of 10 ms, this probability is 10× higher, which speeds up the neighbor discovery process.

Flync exploits the well-behaved, widely available powerline flicker as sychronization source, but the concept applies to any phase-synchronized signal available on different nodes. Because the benefit in terms of a shorter discovery latency stems from increasing the effective slot length, the signal's period must be longer than the duration of a node's active period. The lower the frequency, the longer the slot length



(a) Front (b) Back

Figure 11: Prototype battery-free node based on the nRF52840 MCU. Solar panels on the back charge a tiny capacitor that powers the node.

and the greater the potential benefit. If the period is longer than the charging time of a node, it can be divided down to avoid nodes wasting energy while waiting for the next slot.

4 Prototype Implementation

This section describes the hardware and software components of our prototype implementation.

4.1 Hardware

We design a low-power battery-free node that integrates the circuit from Fig. 10. The node is based on a Nordic Semiconductor nRF52840 MCU, which features a 64 MHz ARM Cortex-M4F and a 2.4 GHz radio with support for Bluetooth 5.2 and IEEE 802.15.4. The node harvests energy using three 23 mm × 8 mm IXYS KXOB25-05X3F solar panels. A TI BQ25505 DC-DC boost converter steps up the voltage of the solar panels and charges a 2 mm × 1.25 mm × 1.25 mm 47 µF multilayer ceramic capacitor (MLCC). However, due to DC bias, the capacitor has only an effective capacitance of around 17 µF at 3.3 V. The BQ25505 implements a maximum power point tracking (MPPT) mechanism that aims to operate the solar panels close to their optimal voltage of around 80 % of the panels' open-circuit voltage. The MPPT circuit obtains a new reference voltage every 16 s by disabling the charger for 256 ms and sampling the panels' open-circuit voltage. Once the capacitor voltage reaches a hardware-programmable threshold of 3.3 V, the BQ25505 sets one of its pins high. This pin is connected to a TI TS5A23166 analog switch that connects the MCU to the capacitor-buffered supply voltage.

The two-layer printed circuit board (PCB) shown in Fig. 11 measures 29 mm × 29 mm. The total cost of all components is \$13.89, including \$8.11 for the relatively expensive, highly integrated nRF52840 module. Comparing our design to recently proposed battery-free platforms with similar capabilities in Table 1, we see that our prototype is indeed one of the first truly battery-free nodes in the sense that the energy storage is negligible in terms of cost, size, and environmental impact: The ceramic capacitor does not contain problematic materials, costs \$0.024, and takes up only 0.3 % of the PCB area.

Platform	Year	Capacitor	Communication
Pible [14]	2018	220 mF super-cap	BLE
luxBeacon [22]	2019	1.5 F super-cap	BLE
Sigrist et al. [46]	2020	520 µF MLCC	BLE
Botoks [12]	2020	100 µF MLCC	868 MHz
This work	2021	47 µF MLCC	BLE PHY

Table 1: Our battery-free prototype node has a sustainable ceramic capacitor that is significantly smaller and cheaper than the energy storage of other recently proposed battery-free platforms.

4.2 Software

Next, we describe our implementation of an efficient runtime for battery-free nodes. We also detail the PLL implementation of Flync and key configuration parameters of Find.

Efficient runtime. Many existing battery-free runtimes discharge the capacitor until the voltage drops below the minimum and the MCU is powered off [12, 17]. To avoid the high energy costs of frequent hardware resets, we implement a different approach that we call *soft intermittency*. During charging, the MCU enters the lowest possible sleep mode, periodically waking up to sample the capacitor voltage with the built-in ADC. In this mode, we measure a total average power draw of 15 µW, including the power for the Flync circuitry and software processing. When the capacitor voltage reaches a software-defined turn-on threshold, the node arms the power-fail comparator, a dedicated peripheral that raises an interrupt when the capacitor voltage drops below a software-defined turn-off threshold. Then the node executes protocol and application code until it is notified by the power-fail comparator upon which it immediately transitions to deep sleep, drastically reducing its power draw until it has again buffered enough energy. While this soft intermittency approach cannot prevent hard resets when there is no energy input for several hundreds of milliseconds, it greatly increases the average efficiency without using additional comparators and switches.

Flync PLL. The comparator at the output of the circuit in Fig. 10 has a relatively small hysteresis, occasionally causing flickering at signal transitions. Furthermore, while MPPT obtains a new reference value, the harvesting current approaches zero, causing the clock signal to pause for hundreds of milliseconds. To provide a stable clock signal despite these disturbances, we implement a PLL that synchronizes the MCU's RTC to the signal extracted with the Flync circuit. We configure the GPIO peripheral to generate an interrupt on a rising edge at the GPIO pin connected to the output of the comparator of the circuit. After a reset, we wait for the first GPIO interrupt. Upon this interrupt, we set up an RTC interrupt to reset the RTC counter after the nominal powerline frequency interval. Ideally, all following GPIO interrupts should coincide with that RTC interrupt. Thus, the counter value at the time of the GPIO interrupt can be interpreted as phase deviation between the external clock signal and the local timer.

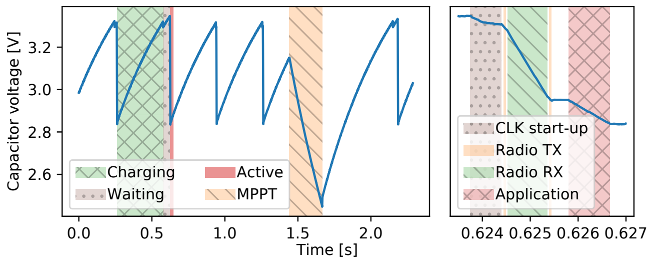


Figure 12: Example trace from a prototype node running Find.

We implement a control loop to continuously adjust the timer period in order to minimize the phase deviation. In this way, we obtain a highly stable interrupt that is phase-synchronized with the variations of the solar panel current and works even during the MPPT sampling or other disruptions.

Find settings. Each beacon in Find's frame structure shown in Fig. 7 consists of 2 B preamble, 3 B base address, 6 B payload, and 1 B cyclic redundancy check (CRC). When using the 2 Mbit BLE mode of the radio, this corresponds to a beacon transmission time of $T_{tx} = 48\,\mu s$. With 17 µF of capacitance, the time required to start the high-frequency oscillator, and a turn-around time of $T_{ta} = 40\,\mu s$, we can afford a maximum listening window of $T_{rx} = 800\,\mu s$. As a result, two nodes can successfully detect each other if they wake up with an offset $\mathcal{T}$ between 88 µs and 848 µs (see Fig. 7).

4.3 Example Real-world Trace

Fig. 12 shows capacitor voltage and activities over time while one of our prototype nodes runs Find. We see that the node charges its capacitor until reaching the turn-on threshold of 3.3 V. It wakes up and samples a random delay from Find's optimized distribution. The necessary computations cause a noticeable drop in the capacitor voltage when transitioning from charging to waiting. After the random delay, the node becomes active and quickly drains its capacitor below the turn-off threshold of 2.8 V. The overview on the left side of Fig. 12 also shows how the capacitor discharges during MPPT at around 1.5 s. The detailed view on the right side shows the individual stages while the node is active. We see that the node first starts the high-frequency clock required to run the radio. Then it sends the first beacon and starts to listen for potential beacons from other nodes. After listening for 800 µs, the node sends the trailing beacon. The remaining energy in the capacitor is assigned to the application that can run until the capacitor voltage hits the turn-off threshold.

5 Evaluation

We manufacture six prototype battery-free nodes to evaluate Find and Flync. We first look at their effectiveness in terms of discovery latency, followed by a detailed characterization of Flync's robustness and performance. Sec. 6 reports on the results of a contact tracing case study based on our techniques.

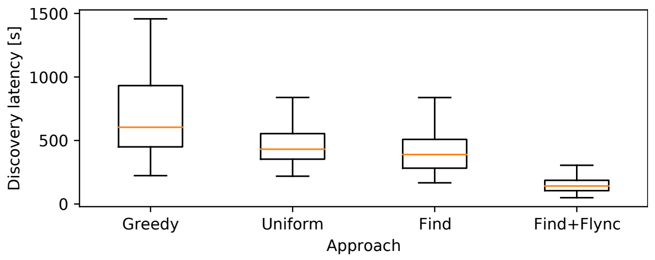


Figure 13: Discovery latency of four different approaches in a network of 6 battery-free nodes. Our techniques outperform the comparison approaches by up to 4.3× (median) and 34.4× (99th percentile).

5.1 Neighbor Discovery Performance

To fairly compare the neighbor discovery performance of our techniques against baseline approaches, we conduct experiments under controlled conditions. Sec. 6 reports on results when using Find and Flync in uncontrolled environments.

Setup. All experiments are conducted in a darkened room with a controllable light source. We place six prototype nodes next to each other on a flat surface. The nodes are programmed to output the ID of any discovered node over universal asynchronous receiver transmitter (UART), while a logic analyzer logs the output of every node. For each run, we let nodes wake up with a random initial delay, and consider the measured time until all 15 bi-directional links are discovered as the discovery latency. We compare Find and Find + Flync with a *greedy* approach, where nodes become active as soon as their capacitor voltage reaches the turn-on threshold, and a *uniform* approach, where nodes randomly delay their wake-ups by a uniformly distributed time. Overall, the measurement campaign took more than 4 days, in which we performed between 48 and 128 runs for each of the four approaches.

Results. Fig. 13 shows the measured discovery latency for each approach, including the median, the 25th and 75th percentiles, and the 1.5× of the interquartile range. Clearly, the greedy approach performs worst. This is mainly because of interleaved activity phases of the nodes, as visible from the trace in Fig. 14. If we zoom in on the first three and the last three wake-ups in the trace, we notice that nodes repeatedly wake up with the same pattern that prevents discovery despite different charging times and MPPT intervals. In Fig. 15, instead, we see that when nodes use Find to randomly delay each wake-up, they are more likely to be active at the same time. For instance, at about 4.5 s, the nodes wake up with an offset of less than 848 µs and are therefore able to successfully exchange beacons as shown in the detailed plot on the right side of Fig. 15. This explains the significant reduction in median discovery latency from 604 s with greedy to 390 s with Find, as visible in Fig. 13. We also see that Find's optimized delay distribution performs slightly better than the uniform approach (median of 431 s), which matches the magnitude of improvement predicted by our model (see Fig. 5).

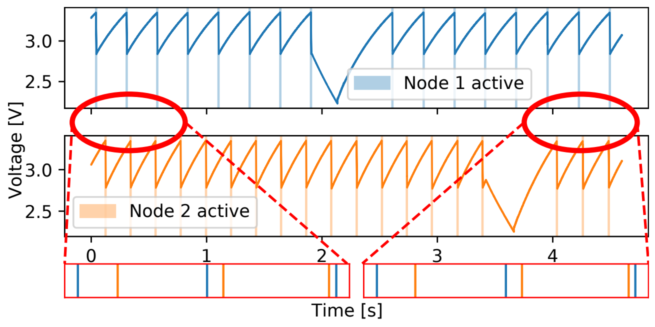


Figure 14: Interleaved activity phases of two nodes when using the *greedy* approach. The zoomed in plots on the bottom show that, despite the disturbances caused by MPPT, the two nodes repeatedly wake up with the same pattern, preventing successful discovery.

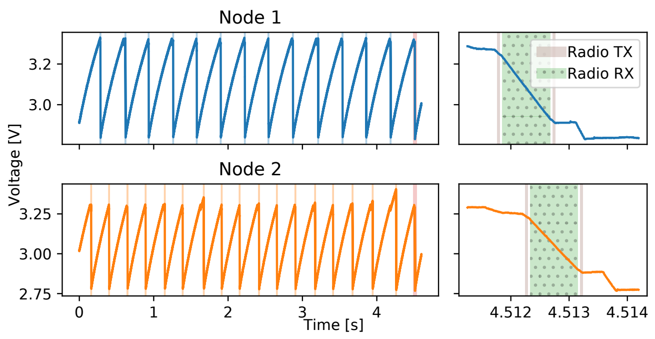


Figure 15: Using Find, nodes prevent *interleaving* by delaying each wake-up by a small random time, enabling quick discovery.

Find + Flync achieves the lowest median discovery latency of 142 s, which corresponds to an overall improvement of 4.3× (median) and 34.4× (99th percentile) compared with greedy.

5.2 Flync Sensitivity

To extract a clock signal, the Flync circuit requires a minimum magnitude of the powerline frequency component in the solar panel current. We empirically determine the corresponding minimum flicker index for our hardware prototype.

Method. The magnitude of the powerline frequency component is proportional to the DC component and decreases with smaller panel size and increasing distance from the light source. We define the worst-case minimum flicker index as the flicker index sufficient to extract a clock signal even at the lowest possible harvesting current. The latter is defined by the minimum power requirements of our prototype when running Find, the panel voltage, and the corresponding efficiency of the DC-DC converter. Our solar panels have a typical panel voltage of 1 V at the maximum power point. At this voltage, our DC-DC converter has an efficiency of 80 %. Thus, the minimum harvesting current to cover the power requirements of our prototype of about 37.5 µW is 50 µA.

We use a Keithley 2600B sourcemeter to generate a current signal with a DC offset of 50 µA while sweeping the ampli-

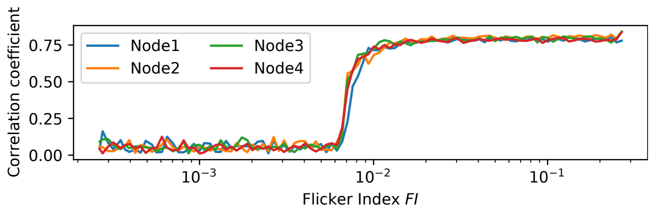


Figure 16: For a flicker index ≥0.008 Flync provides a stable clock.

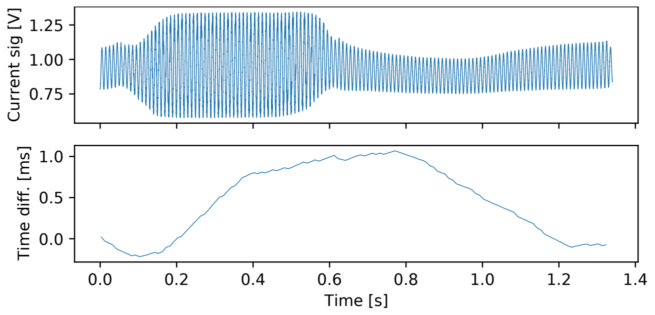


Figure 17: Current signal and time difference between two nodes while one node changes its distance and angle to the light source.

tude of the 100 Hz AC component. The current is fed to the input of our prototype that is usually connected to the solar panel. By limiting the voltage at the output of the sourcemeter to 1.25 V, the MPPT circuit regulates the input to around 1 V. For every setting of the AC amplitude, we record 5 s of clock signal with a mixed-signal oscilloscope. To quantify the quality of the clock signal, we compute the correlation coefficient between the signal and a phase-aligned 100 Hz reference. We repeat these measurements for four of our prototype nodes.

Results. The results in Fig. 16 show that there is a distinct threshold at around $FI = 0.008$ beyond which all nodes begin to output a clean clock signal. Comparing this with Fig. 9, we conclude that, with the exception of 5 LED lamps, our prototype works with the vast majority of the lamps we tested.

5.3 Flync Robustness

We now assess the robustness of Flync when a node changes its position and orientation relative to the light source, when the solar panels of a node are temporarily covered, and when electrical loads are temporarily connected to the same power strip. To this end, we experiment with two nodes powered by a desk lamp and connect them to an oscilloscope. We quantify robustness by measuring the time difference between clock edges on the two nodes. As a benchmark, we note that our implementation can tolerate a time difference of up to 848 µs.

Mobility. We keep one node static and attach the other one to the wrist of a person. The person waves, changing distance and angle between the node's solar panels and the light source.

Fig. 17 shows a period where the node moves closer and farther away from the lamp. The changes in the amplitude of the current signal affect the time difference between the nodes.

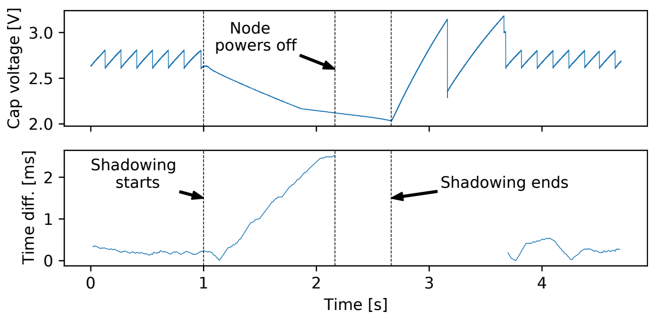


Figure 18: Capacitor voltage and time difference between two prototype nodes while temporarily covering the solar panel of one node.

The comparator that thresholds the sine wave uses a low-pass filter that reacts slowly to changes in the average amplitude. As a result, the clock signal deteriorates temporarily, causing an increased time difference of up to 1 ms. However, after a short while, the time difference recovers to previous levels.

Shadowing. To investigate the impact of shadowing, we put both nodes on a table and temporarily cover one of them by slowly moving a hand between the lamp and the node.

Fig. 18 shows that the time difference increases after covering the panel as the PLL loses its reference signal. However, without significant energy input, the node does not reach the turn-on threshold, which renders communication infeasible anyhow. As soon as the panel is uncovered, the node quickly charges up again and, after less than a second, the clock returns with a small time difference.

Electrical loads. We repeatedly switch on and off a drilling machine and a vacuum cleaner connected to the same power strip as the lamp. We do not observe any noticeable effect of the loads on the time difference between the two nodes.

5.4 Flync Jitter

In a final set of experiments, we look at the time difference between the clock signals of different nodes when these are: (*i*) powered by a single light source, (*ii*) placed in different rooms, and (*iii*) powered by different types of light sources.

Testbed. For these experiments, we built a distributed testbed of observer nodes. The observer nodes are accurately time-synchronized to within 479 ns, and record the clock signals of the attached prototype nodes with a resolution of 62.5 ns.

Single light source. We place six of our prototype nodes in the same room with a single halogen lamp. The experiments are conducted during the day, and the nodes receive a mixture of natural sunlight and artificial light from the lamp. Using our testbed, we record the clock edges of all six nodes for 1 h.

Fig. 19 shows the pairwise time difference between nodes. Because the phase offset resulting from propagation delays of light is negligible, the jitter must be introduced on each node. For example, a slight difference in the offset voltage of the comparator can lead to a significant mean difference of the

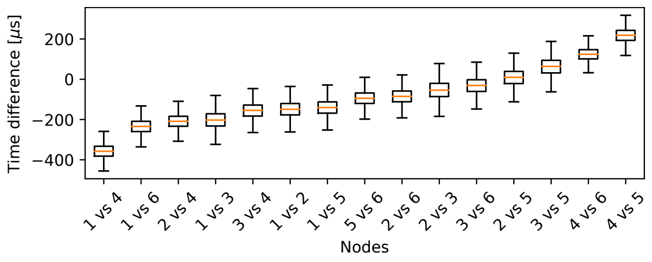


Figure 19: Pairwise time difference between clock edges on different prototype nodes when these are powered by a single light source.

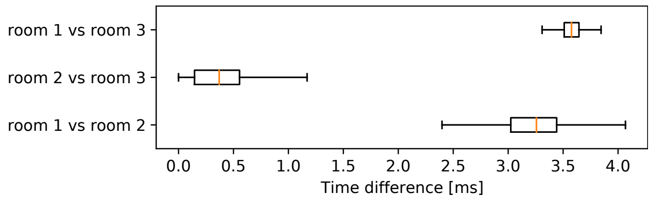


Figure 20: Pairwise time difference between clock edges on prototype nodes placed in different rooms with the same type of lamp.

resulting clock signal. Nevertheless, with 95 % of the more than five million recorded pairs below 244 µs, the jitter is well below the 848 µs tolerated by our Find implementation.

Different rooms. We conduct experiments in three rooms of an office building equipped with fluorescent tubes. The rooms are located on a long hallway with a distance of around 15 m between the middle room and the other two. We place two nodes in each room, and record with our testbed for 4 h while the nodes receive light from the tubes as well as sunlight.

Fig. 20 shows that there is a small offset between rooms 2 and 3 with 95 % of the recorded values being smaller than 700 µs. The offsets between rooms 1 and 2 and rooms 1 and 3 are centered around 3.3 ms. While residential homes are often connected to a single phase, larger apartment blocks or commercial buildings are typically fed by three-phase power. Apparently, the lights in room 1 are connected to a different phase than the lights in rooms 2 and 3, leading to a 60° phase and 3.3 ms time shift between the light intensity variations. Thus, when nodes need to discover neighbors across rooms with lights potentially connected to different power phases, they must be able to become active not only at the edge of their own Flync clock signal, but also with a 60° phase shift.

Different types of light sources. We plug an LED, a fluorescent, and two halogen lamps into the same power strip. We place one node under each lamp so that it only receives light from this lamp, and record for 30 min with our testbed.

Fig. 21 reveals large offsets between the clocks of nodes powered by different types of lamps. These offsets are due to varying phase shifts between the powerline voltage and the brightness variations of the lamp. For example, although the current through an incandescent lamp is in phase with the supply voltage, the filament may take some time to heat

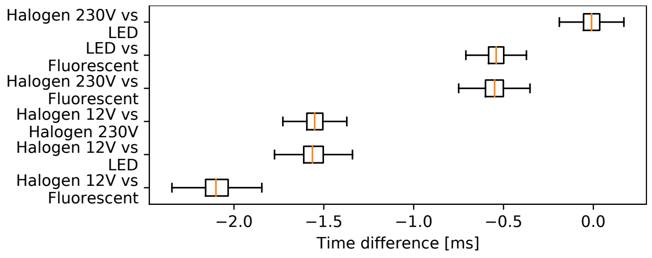


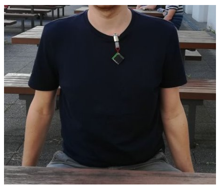
Figure 21: Pairwise time difference between clock edges on different prototype nodes when these are powered by different types of lamps.

up and cool down, leading to the observed phase shift. Other types of lamps contain inductors or capacitive elements, a switching power supply, or an electronic ballast that cause different phase shifts. This shows that Flync does not work out of the box when different nodes are powered by different types of lamps. The static phase shifts would need to be measured during deployment or learned at runtime. On the other hand, Flync may not work reliably when individual nodes receive a mixture of light from different types of lamps. The results from the previous experiments (see Figs. 19 and 20) show that Flync works well when nodes receive a mixture of natural sunlight and artificial light from the same type of lamp.

6 Case Study: Contact Tracing

Automatic contact tracing is important to contain the spread of infectious diseases (e.g., SARS-CoV2) in a scalable manner. It allows to quickly identify contacts of an infected person and to quarantine potentially infected individuals before they become contagious. To assess the potential of our proposed designs for real-world battery-free applications, we conduct a contact tracing case study with our prototype nodes.

Setup. We attach six nodes to the shirts of human participants, as shown in Fig. 22a. The nodes run the Find protocol, logging the timestamp and ID of each discovered node to non-volatile memory. As we are only interested in relatively close contacts that would allow a virus to transmit from one person to another, we set the transmission power of the beacons to −16 dBm. We run experiments indoors and outdoors, as detailed below. After each run, we dump the content of the non-volatile memory of each node to a computer for analysis.



(a) Node on shirt.

(b) Setup of experiment in an open-air pub.

Figure 22: Battery-free contact tracing.

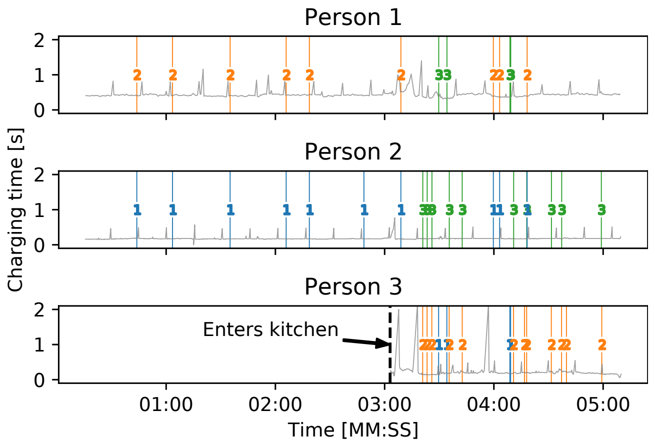


Figure 23: Charging times and rendezvous in coffee kitchen experiment. Vertical markers show rendezvous with the respective person.

Indoor experiment: coffee kitchen. Two persons sit at a table in a small coffee kitchen, roughly 1.5 m apart from each other. After 3 min a third person enters the kitchen and prepares a coffee for 2 min. The kitchen is equipped with fluorescent lamps, and we use Flync together with Find.

Fig. 23 plots the charging times and recorded rendezvous of the three nodes over time. We see a total of 49 received beacons. All contacts are logged successfully with low latency, despite the relatively long charging times of hundreds of milliseconds. Specifically, the first contact between persons 1 and 2 is detected after 43.9 s. When person 3 enters the kitchen, it takes 26.6 s and 17.9 s until the contacts with persons 1 and 2 are detected, respectively. Overall, the median time between rendezvous of the same two nodes is 7.5 s.

Outdoor experiment: open-air pub. Three pairs of persons sit at opposite sides of three tables (see Fig. 22b). Two tables are next to each other; the third table is at a distance of around 4.5 m. We perform the experiments in the morning of a slightly overcast day at an open-air pub without direct sunlight. Receiving only natural sunlight, the nodes do not make use of Flync. We conduct three consecutive 15 min runs.

We measure a total of 4426 received beacons. All contacts between persons on the same table are successfully recorded. More importantly, contacts between persons on different tables in close vicinity are also reliably detected. Due to the low transmit power, we do not see any rendezvous between the first two tables and the third remote table, which is expected and in fact desirable because we only want to trace contacts that are associated with an actual risk of virus transmission. Fig. 24 shows the histogram of the time between consecutive rendezvous between the same two nodes. As expected, the time between rendezvous is approximately exponentially distributed, and the mean is estimated between 2.61 s and 2.78 s with 95 % confidence. This means, under the given conditions, we are able to detect contacts with a resolution of around 2.67 s, allowing for fine-grained contact tracing.

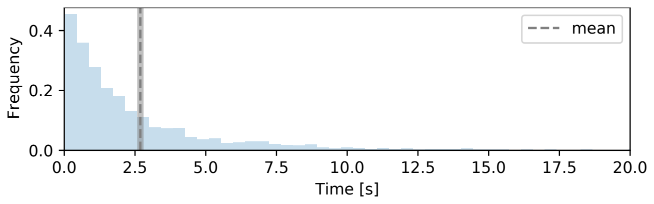


Figure 24: Histogram of the time difference between rendezvous of the same two nodes in the open-air pub experiment.

Summary. The results from our contact tracing case study show that Find and Flync are also effective under uncontrolled real-world conditions. Outdoors, energy availability is high and therefore Find alone enables fast rendezvous and fine-grained contact tracing. Indoors, Flync can compensate for the significantly lower energy density of interior light, providing decent performance even under these challenging conditions.

7 Discussion

We have presented two novel techniques that enable for the first time efficient device-to-device communication in the face of intermittency. By introducing random delays, Find breaks interleaved activity patterns of battery-free devices to discover each other faster and more efficiently. By tapping into the powerline-induced flicker of state-of-the-art lamps, Flync phase-synchronizes devices that harvest energy from indoor light. While we have exploited Flync to further speed up discovery in battery-free networks, Flync is useful for other purposes and also applicable to battery-supported devices.

Recent work tackles the intermittency problem on individual battery-free devices in terms of, for example, computing and time keeping [8,12,18,34]. We instead focus on communication between battery-free devices that operate intermittently. Like prior work, our techniques are relevant if intermittency makes traditional approaches inefficient or unreliable. To understand the scope of our work, we discuss intermittency and relevant impact factors below. Afterward, we discuss the influence of built-in randomness on our proposed techniques.

7.1 When Does Intermittency Occur?

A battery-free device goes through periods with low power requirements (e.g., system-off and sleep modes) and high power requirements (e.g., sensing, processing, and communication). Since the instantaneous power available from a harvester is often insufficient to support a battery-free device during periods with high power requirements, some form of energy storage is needed that buffers energy when the device is inactive to support a high-power workload for a short period of time.

The minimum size of the energy storage is determined by the demands of the largest atomic operation that must not be interrupted. For example, to transmit or receive a packet, the buffered energy needs to be sufficient to power the radio for at least the airtime of one complete packet; other examples of atomic operations include reading out a sensor or executing a checkpoint [21]. In our proposed Find protocol, the largest atomic operation is the frame sequence depicted in Fig. 7.

If a device with an active power draw higher than the harvesting power is equipped with an energy storage that does not support executing multiple iterations of the largest atomic operation from a single full charge, it is forced to go through periods of inactivity—the device is said to operate intermittently. Intermittency is in stark contrast to duty cycling, which is intentionally used on devices with primary or rechargeable batteries, yet the devices can become active at any point in time subject only to an upper bound on the average duty cycle. By contrast, intermittency prevents a device from becoming active at any point in time, and when a device enters and exits the inactivity phases is only partially controllable, at best.

7.2 What Factors Impact Intermittency?

Three key dimensions influence the extent of the intermittency problem: energy input, energy storage, and workload.

Energy input. An ambient energy source may exhibit intermittent behavior, including periods where it emits no energy. Clearly, a battery-free device can only harvest energy when the ambient source emits energy. In this case, provisioning a device with a harvester that provides the power required to continuously operate the device in high-power mode prevents the intermittency problem. This, however, would come with major drawbacks in terms of size, weight, and costs. For example, a battery-free device may draw only 10 µW on average but 10 mW when active, thus requiring to over-provision the harvester by a factor of 1000. While such over-provisioning is in theory always possible, it is severely limited in practice by the constraints imposed by the application requirements.

Energy storage. If permitted by the application requirements, an energy storage larger than the minimum required to execute the largest atomic operation may be used. For example, using a high-capacity rechargeable battery can prevent intermittency. Such batteries have a high energy density, but their minimum physical dimensions are typically orders of magnitude larger than those of capacitors. Batteries are also more expensive and subject to aging, losing capacity over time and eventually malfunctioning with excessive heat and leakage of potentially toxic chemicals. By contrast, capacitors have low energy density, but are extremely cheap, readily available in sizes well below 0.1 mm^3, have negligible aging effects, and do not contain problematic materials (e.g., toxic chemicals). Thus, despite advances in battery technology, alternative systems to store energy are being explored [3] and capacitors are widely regarded as a more sustainable option [11, 45].

When a device is inactive, it accumulates charge until the capacitor voltage reaches a turn-on threshold. The amount of energy that can be stored depends on the turn-on threshold, which is limited by the breakdown voltage of the capacitor and the device's maximum operating voltage. When a device is active, it discharges the capacitor until the voltage reaches a turn-off threshold, which is dictated by the device's minimum operating voltage. Thus, for the same capacitor, a device with a lower minimum operating voltage or a higher maximum operating voltage can increase the effective amount of buffered energy that can be used. This allows to either use a smaller capacitor or execute longer from a single full charge, potentially alleviating the intermittency problem.

Workload. While lower-power hardware can reduce the average power draw in sleep mode and thus the charging time, it does not generally avoid intermittency. This would require pushing also the active power below the harvesting power.

Reducing the transmission power of the radio can extend the time a device can operate from a single full charge. While this may alleviate the intermittency problem, it also reduces the communication range, which may render device-to-device communication infeasible or require multi-hop networking.

Similarly, using backscatter communication instead of active radio communication may bring the active power draw of a device below the harvesting power and thereby enable continuous operation. However, backscatter requires the presence of an external carrier and may pose limitations in terms of communication range and data rate. In particular, existing practical implementations of tag-to-tag backscatter receivers do not yet reach the point where the end-to-end power draw is negligible (i.e., below sleep power of around 1 µW) [35, 39], thus leaving a significant region in the design space of battery-free backscatter devices where intermittency occurs.

7.3 Impact of Built-in Spatial Randomness

Find tackles interleaving by letting nodes randomly and independently delay their wake-ups. This approach is particularly effective in scenarios with little built-in spatial randomness, that is, when the harvested energy exhibits limited variability *between* nodes, regardless of a potentially high temporal variability in harvested energy. We believe this holds for a broad class of battery-free application scenarios, because nodes in a confined space often harvest energy from the same ambient source(s). On the other hand, a high built-in spatial randomness may alleviate the interleaving problem. Although our case study experiments exhibit built-in spatial and temporal randomness, it remains an open question how built-in spatial randomness may influence the choice of Find's delay distribution and scale parameter as well as its overall effectiveness.

8 Related Work

Battery-free device-to-device communication. Prior work on battery-free wireless device-to-device communication is mainly theoretical [25, 51], studying the capacity limits for different energy scheduling, transmission, and decoding policies. Understanding energy issues on the receiver side [2] and the impact of intermittency have been open problems. On the other hand, practical work on tag-to-tag backscatter communication has primarily focused on physical-layer issues and considers intermittency an orthogonal problem [20, 32, 35].

Work	Type	Sensing Signal	Power
Syntonistor [42]	frequency	EM radiation	300 µW
Flight [31]	frequency	light sensor	5394 µW
Flync	freq.+phase	solar current	5 µW

Table 2: Compared with prior work using powerline frequency for synchronization, Flync provides frequency and phase synchronization from the solar panel current at significantly lower power draw.

Rendezvous and neighbor discovery protocols. Blind rendezvous is the process of establishing a communication link between nodes in a distributed system without any prior information [16]. Neighbor discovery protocols for wireless networks target a sub-class of the blind rendezvous problem with the goal of optimizing the trade-off between discovery latency and energy consumption. Deterministic protocols let nodes wake up according to a schedule based on (co-)prime numbers [13, 23], a quorum [19, 27, 28], or by systematically traversing slots [5, 50]. This way, they can provide guaranteed bounds on discovery latency [24]. Probabilistic protocols are stateless, robust to varying conditions, and offer low average discovery latency [10]. For example, the influential birthday protocol [37] and follow-up work [48, 49] analyze optimal transmit probabilities to maximize the fraction of links discovered in a given time. However, none of the existing neighbor discovery protocols are applicable to battery-free networks because they require nodes to be able to wake up at arbitrary points in time, not taking into account intermittency.

Powerline-based clock synchronization. We are not the first to exploit the powerline frequency signal for synchronization. The Syntonistor extracts a stable clock signal from electromagnetic (EM) powerline radiation using a large coil [42]. It draws 300 µW of power, 60× more than Flync. Flight samples a light sensor to synchronize a node's oscillator to the powerline-induced brightness variations of fluorescent lamps [31]. Using Flight, synchronization takes 100 ms at a power draw of 5394 µW, 1000× more than Flync. As summarized in Table 2, both approaches only synchronize the frequency of local clocks, eliminating the need to periodically compensate for clock drift, but do not exploit phase information. They also use dedicated high-power sensors, whereas Flync uses a low-power circuit to extract the signal from the current of the solar panel.

Energy harvesters as sensors. Previous work has explored the use of the harvesting current or voltage as a sensing signal for indoor positioning [41], gait recognition [33], gesture recognition [47], activity classification [44], and transport-mode detection [43]. To the best of our knowledge, we are the first to exploit context information from harvested energy for synchronization. Furthermore, Flync is the first design that extracts the sensing signal from current variations of a solar panel that is simultaneously used to power the system.

Visible light communication. Flync exploits the powerline-induced brightness variations as an intrinsic property of ubiquitous lamps. When modifying existing lighting infrastructure, it is possible to encode arbitrary data into the brightness variations. This opportunity has been used for downlink communication [40], indoor positioning [26], and battery-free duplex visible light communication [29]. By modulating light with a well-defined synchronization signal, the efficiency and applicability of Flync could be further improved. Also, our approach to harvest energy while simultaneously demodulating encoded signals from the same panel may reduce the size and power of existing visible light communication receivers.

9 Conclusions

Leaving batteries behind allows for building cheap, tiny, and maintenance-free devices that can be embedded into smart textiles, intelligent surfaces, or even the human body. In this paper, we have addressed the problem of enabling efficient battery-free device-to-device communication. Experiments with a prototype platform and implementation show that our proposed techniques empower battery-free devices to quickly and efficiently discover each other despite their unpredictable intermittent operation. By bootstrapping battery-free wireless networks, we believe that our work provides a stepping stone for future research toward full system and communication stacks for this emerging kind of networked system.

Availability

Artifacts are available to the public under a permissive MIT license at `https://find.nes-lab.org/`. These include a Python implementation of the Find model from Sec. 2, which can be used to reproduce the analytical results in Figs. 3 to 6, as well as the hardware design files and the firmware of our prototype implementation from Sec. 4, which we used for the experiments and case study described in Secs. 5 and 6.

Acknowledgments

Thanks to Brano Kusy, Carlos Pérez-Penichet, and Carsten Herrmann for their feedback that helped improve this paper, to Friedrich Schmidt for his help with the hardware design, to James Broadhead for the preliminary experiments, and to all participants of the contact tracing case study. Thanks also to the anonymous reviewers, and to our shepherd, Shyam Gollakota. This work was supported by the Emmy Noether project NextIoT (DFG grant ZI 1635/2-1) and cfaed.

References

[1] T52x/t530 polymer electrolytic capacitors. Technical report, KEMET Electronics Corporation, 2018.

[2] Kofi Sarpong Adu-Manu, Nadir Adam, Cristiano Tapparello, Hoda Ayatollahi, and Wendi Heinzelman. Energy-harvesting wireless sensor networks (EH-WSNs): A review. *ACM Transactions on Sensor Networks*, 14(2), 2018.

[3] Science Communication Unit at University of the West of England. Towards the battery of the future. *Science for Environment Policy*, 2018.

[4] Abu Bakar and Josiah Hester. Making sense of intermittent energy harvesting. In *Proceedings of the 6th ACM International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems (ENSSys)*, 2018.

[5] Mehedi Bakht, Matt Trower, and Robin Hilary Kravets. Searchlight: Won't you be my neighbor? In *Proceedings of the 18th ACM Annual International Conference on Mobile Computing and Networking (MobiCom)*, 2012.

[6] James Bates, Marc Beaulieu, Michael Miller, and Joseph Paulus. Reaching the highest reliability for tantalum capacitors. Technical report, AVX Corporation, 2013.

[7] Naveed Anwar Bhatti, Muhammad Hamad Alizai, Affan A. Syed, and Luca Mottola. Energy harvesting and wireless transfer in sensor network applications: Concepts and experiences. *ACM Transactions on Sensor Networks*, 12(3), 2016.

[8] Adriano Branco, Luca Mottola, Muhammad Hamad Alizai, and Junaid Haroon Siddiqui. Intermittent asynchronous peripheral operations. In *Proceedings of the 17th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2019.

[9] R. P. Brent. An algorithm with guaranteed convergence for finding a zero of a function. *The Computer Journal*, 14(4), 1971.

[10] Lin Chen and Kaigui Bian. Neighbor discovery in mobile sensing applications: A comprehensive survey. *Ad Hoc Networks*, 48, 2016.

[11] Albert Cohen, Xipeng Shen, Josep Torrellas, James Tuck, Yuanyuan Zhou, Sarita Adve, Ismail Akturk, Saurabh Bagchi, Rajeev Balasubramonian, Rajkishore Barik, Micah Beck, Ras Bodik, Ali Butt, Luis Ceze, Haibo Chen, Yiran Chen, Trishul Chilimbi, Mihai Christodorescu, John Criswell, Chen Ding, Yufei Ding, Sandhya Dwarkadas, Erik Elmroth, Phil Gibbons, Xiaochen Guo, Rajesh Gupta, Gernot Heiser, Hank Hoffman, Jian Huang, Hillery Hunter, John Kim, Sam King, James Larus, Chen Liu, Shan Lu, Brandon Lucia, Saeed Maleki, Somnath Mazumdar, Iulian Neamtiu, Keshav Pingali, Paolo Rech, Michael Scott, Yan Solihin, Dawn Song, Jakub Szefer, Dan Tsafrir, Bhuvan Urgaonkar, Marilyn Wolf, Yuan Xie, Jishen Zhao, Lin Zhong, and Yuhao Zhu. Inter-disciplinary research challenges in computer systems for the 2020s. Technical report, 2018.

[12] Jasper de Winkel, Carlo Delle Donne, Kasim Sinan Yildirim, Przemysław Pawełczak, and Josiah Hester. Reliable timekeeping for intermittent computing. In *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[13] Prabal Dutta and David Culler. Practical asynchronous neighbor discovery and rendezvous for mobile sensing applications. In *Proceedings of the 6th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2008.

[14] Francesco Fraternali, Bharathan Balaji, Yuvraj Agarwal, Luca Benini, and Rajesh Gupta. Pible: Battery-Free Mote for Perpetual Indoor BLE Applications. In *Proceedings of the 5th ACM Conference on Systems for Built Environments (BuildSys)*, 2018.

[15] Kai Geissdoerfer, Mikołaj Chwalisz, and Marco Zimmerling. Shepherd: A portable testbed for the batteryless IoT. In *Proceedings of the 17th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2019.

[16] Zhaoquan Gu, Yuexuan Wang, Qiang-Sheng Hua, and FC Lau. *Rendezvous in Distributed Systems*. Springer, 2017.

[17] Josiah Hester and Jacob Sorber. Flicker: Rapid prototyping for the batteryless Internet-of-Things. In *Proceedings of the 15th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2017.

[18] Josiah Hester, Nicole Tobias, Amir Rahmati, Lanny Sitanayah, Daniel Holcomb, Kevin Fu, Wayne P. Burleson, and Jacob Sorber. Persistent clocks for batteryless sensing devices. *ACM Transactions on Embedded Computing Systems*, 15(4), 2016.

[19] Tingpei Huang, Haiming Chen, Li Cui, and Yuqing Zhang. EasiND: Neighbor discovery in duty-cycled asynchronous multichannel mobile WSNs. *International Journal of Distributed Sensor Networks*, 9(7), 2013.

[20] Vikram Iyer, Vamsi Talla, Bryce Kellogg, Shyamnath Gollakota, and Joshua Smith. Inter-technology backscatter: Towards internet connectivity for implanted devices. In *Proceedings of the ACM SIGCOMM Conference*, 2016.

[21] Neal Jackson, Joshua Adkins, and Prabal Dutta. Capacity over capacitance for reliable energy harvesting sensors. In *Proceedings of the 18th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, 2019.

[22] Kang Eun Jeon, James She, Jason Xue, Sang-Ha Kim, and Soochang Park. luXbeacon—A batteryless beacon for green IoT: Design, modeling, and field tests. *IEEE Internet of Things Journal*, 6(3), 2019.

[23] Arvind Kandhalu, Karthik Lakshmanan, and Ragunathan (Raj) Rajkumar. U-connect: A low-latency energy-efficient asynchronous neighbor discovery protocol. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, 2010.

[24] Philipp H. Kindt and Samarjit Chakraborty. On optimal neighbor discovery. In *Proceedings of the ACM SIGCOMM Conference*, 2019.

[25] Meng-Lin Ku, Wei Li, Yan Chen, and K. J. Ray Liu. Advances in energy harvesting communications: Past, present, and future challenges. *IEEE Communications Surveys & Tutorials*, 18(2), 2016.

[26] Ye-Sheng Kuo, Pat Pannuto, Ko-Jen Hsiao, and Prabal Dutta. Luxapose: Indoor positioning with mobile phones and visible light. In *Proceedings of the 20th ACM Annual International Conference on Mobile Computing and Networking (MobiCom)*, 2014.

[27] Shouwen Lai, Binoy Ravindran, and Hyeonjoong Cho. Heterogenous Quorum-Based Wake-Up Scheduling in Wireless Sensor Networks. *IEEE Transactions on Computers*, 59(11), 2010.

[28] Shouwen Lai, Bo Zhang, Binoy Ravindran, and Hyeonjoong Cho. CQS-Pair: Cyclic quorum system pair for wakeup scheduling in wireless sensor networks. In *Proceedings of the International Conference on Principles of Distributed Systems (OPODIS)*. Springer, 2008.

[29] Jiangtao Li, Angli Liu, Guobin Shen, Liqun Li, Chao Sun, and Feng Zhao. Retro-VLC: Enabling battery-free duplex visible light communication for mobile and IoT applications. In *Proceedings of the 16th ACM International Workshop on Mobile Computing Systems and Applications (HotMobile)*, 2015.

[30] Tianxing Li and Xia Zhou. Battery-free eye tracker on glasses. In *Proceedings of the 24th ACM Annual International Conference on Mobile Computing and Networking (MobiCom)*, 2018.

[31] Zhenjiang Li, Wenwei Chen, Cheng Li, Mo Li, Xiang-Yang Li, and Yunhao Liu. FLIGHT: Clock calibration using fluorescent lighting. In *Proceedings of the 18th ACM Annual International Conference on Mobile Computing and Networking (MobiCom)*, 2012.

[32] Vincent Liu, Aaron Parks, Vamsi Talla, Shyamnath Gollakota, David Wetherall, and Joshua R. Smith. Ambient backscatter: Wireless communication out of thin air. In *Proceedings of the ACM SIGCOMM Conference*, 2013.

[33] Dong Ma, Guohao Lan, Weitao Xu, Mahbub Hassan, and Wen Hu. SEHS: Simultaneous energy harvesting and sensing using piezoelectric energy harvester. In *Proceedings of the 3rd IEEE/ACM International Conference on Internet-of-Things Design and Implementation (IoTDI)*, 2018.

[34] Kiwan Maeng and Brandon Lucia. Adaptive dynamic checkpointing for safe efficient intermittent computing. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[35] Amjad Yousef Majid, Michel Jansen, Guillermo Ortas Delgado, Kasim Sinan Yildirim, and Przemysław Pawełczak. Multi-hop backscatter tag-to-tag networks. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, 2019.

[36] Amjad Yousef Majid, Patrick Schilder, and Koen Langendoen. Continuous sensing on intermittent power. In *Proceedings of the 19th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, 2020.

[37] Michael J. McGlynn and Steven A. Borbash. Birthday protocols for low energy deployment and flexible neighbor discovery in ad hoc wireless networks. In *Proceedings of the 2nd ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc)*, 2001.

[38] Sung Y. Park and Anil K. Bera. Maximum entropy autoregressive conditional heteroskedasticity model. *Journal of Econometrics*, 150(2), 2009.

[39] Carlos Pérez-Penichet, Claro Noda, Ambuj Varshney, and Thiemo Voigt. Battery-free 802.15.4 receiver. In *Proceedings of the 17th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, 2018.

[40] Niranjini Rajagopal, Patrick Lazik, and Anthony Rowe. Visual light landmarks for mobile devices. In *Proceedings of the 13th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, 2014.

[41] Julian Randall, Oliver Amft, Jürgen Bohn, and Martin Burri. LuxTrace: Indoor positioning using building illumination. *Personal and Ubiquitous Computing*, 11(6), 2007.

[42] Anthony Rowe, Vikram Gupta, and Ragunathan (Raj) Rajkumar. Low-power clock synchronization using electromagnetic energy radiating from AC power lines. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2009.

[43] Muhammad Moid Sandhu, Kai Geissdoerfer, Sara Khalifa, Raja Jurdak, Marius Portmann, and Brano Kusy. Towards energy positive sensing using kinetic energy harvesters. In *Proceedings of the IEEE International Conference on Pervasive Computing and Communications (PerCom)*, 2020.

[44] Muhammad Moid Sandhu, Sara Khalifa, Kai Geissdoerfer, Raja Jurdak, and Marius Portmann. SolAR: Energy positive human activity recognition using solar cells. In *Proceedings of the IEEE International Conference on Pervasive Computing and Communications (PerCom)*, 2021.

[45] Mahadev Satyanarayanan, Wei Gao, and Brandon Lucia. The computing landscape of the 21st century. In *Proceedings of the 20th ACM International Workshop on Mobile Computing Systems and Applications (HotMobile)*, 2019.

[46] Lukas Sigrist, Rehan Ahmed, Andres Gomez, and Lothar Thiele. Harvesting-aware pptimal communication scheme for infrastructure-less sensing. *ACM Transactions on Internet of Things*, 1(4), 2020.

[47] Ambuj Varshney, Andreas Soleiman, Luca Mottola, and Thiemo Voigt. Battery-free visible light sensing. In *Proceedings of the 4th ACM Workshop on Visible Light Communication Systems (VLCS)*, 2017.

[48] Sudarshan Vasudevan, Micah Adler, Dennis Goeckel, and Don Towsley. Efficient algorithms for neighbor discovery in wireless networks. *IEEE/ACM Transactions on Networking*, 21(1), 2013.

[49] Sudarshan Vasudevan, Donald Towsley, Dennis Goeckel, and Ramin Khalili. Neighbor discovery in wireless networks and the coupon collector's problem. In *Proceedings of the 15th ACM Annual International Conference on Mobile Computing and Networking (MobiCom)*, 2009.

[50] Keyu Wang, Xufei Mao, and Yunhao Liu. BlindDate: A neighbor discovery protocol. In *Proceedings of the 42nd International Conference on Parallel Processing (ICPP)*, 2013.

[51] Tongxin Zhu, Jianzhong Li, Hong Gao, and Yingshu Li. Broadcast scheduling in battery-free wireless sensor networks. *ACM Transactions on Sensor Networks*, 15(4), 2019.

AIRCODE: Hidden Screen-Camera Communication on an Invisible and Inaudible Dual Channel

Kun Qian[*#], Yumeng Lu[*], Zheng Yang[*1], Kai Zhang[*]
Kehong Huang[*], Xinjun Cai[*], Chenshu Wu[†], Yunhao Liu[*‡]
[*]*Tsinghua University,* [#]*University of California San Diego*
[†]*University of Maryland College Park,* [‡]*Michigan State University*
{qiank10, lym.mongo, hmilyyz, nezharen1995, huangkh13, caixj08, wucs32, yunhaoliu}@gmail.com

Abstract

Hidden screen-camera communication emerges as a key enabler for the next generation videos that allow side information, such as TV commercials, augmented contents, and even the video itself, to be delivered to machines during normal watching. To guarantee imperceptibility to human eyes, existing solutions have to sacrifice data rate and reliability enormously. This paper presents AIRCODE, a hidden screen-camera communication system built upon invisible visual and inaudible audio dual channel. While ensuring great unobtrusiveness, AIRCODE achieves robust communication at a remarkably high rate of >1Mbps, for the first time, enabling imperceptible transmission of not only texts but also videos. AIRCODE makes two key technical contributions. First, AIRCODE takes the complementary advantages of video and audio channels by exploiting the reliable yet low-rate inaudible audio link as the control channel while the unreliable but high-rate visual link as the data channel. Second, AIRCODE incorporates visual odometry to accurately identify and track the captured screen, regardless of dynamic video contents and surrounding interference. Experiments on commercial monitors and smartphones demonstrate that AIRCODE significantly outperforms the state-of-the-art system, yielding a remarkable data rate of 1069 Kbps while with BER of 5%.

1 Introduction

Over the past few decades, video has risen into popularity across the globe. Billions of video are produced, captured, shared, and viewed every day and everywhere. Rather than merely watching them, audience often desires to acquire extra information related to the video content, especially with their carry-on devices, e.g., smart phone and smart glasses. For example, an advertisement can deliver a second video introducing detailed usage and function of the advertised product to its potential users. Instead of letting the viewers record a blurry video with unintended surrounding contents and color distortion, a video can provide a low quality version of itself for direct sharing. An AR video can compute and send the augmented contents to a viewer's device for direct rendering without draining its limited computing and power resources. To make these applications a reality, convenient and friendly communication approach, better with high data rates and reliability, is needed.

One intuitive solution is to integrate screens with Wi-Fi or Bluetooth, like nowadays smart TV, and convey side information through the wireless channel. However, such combination has several drawbacks. First, wireless connection requires explicit setup. A viewer wearing smart glasses may see multiple screens during his daily life and be bothered to search their Wi-Fi from many available networks. Second, wireless device and screen are not well synchronized. For example, a screen playing an AR video should deliver the augmented contents synchronized at the frame level, e.g., 16.7 ms with a frame rate of 60 Hz, which cannot be always achieved by Wi-Fi due to occasionally large latency and jitter [29]. Last but not least, the broadcasting behavior of wireless devices causes information spamming or leakage to unwanted people. For example, in TV commercial, only the viewers of the advertisement need the side information about the product. In video sharing, an eavesdropper must not access the low quality video due to copyright protection.

Alternatively, as cameras are indispensable to smart devices, it is more convenient, impromptu and secure to embed extra information into video and deliver them through the screen-camera channel without impacting viewers' watching experience. Thanks to the high refresh rates ($\geq$ 120 Hz) of modern screens, pioneer works [24, 36, 41] hide data in high frame rate videos to cheat human eyes that have much lower perception rates (40-50 Hz) [31]. However, they emphasize on human perception but sacrifice data rates (e.g., 551 Kbps with a bit error rate (BER) of 8% [41]), hardly reliable for real data communication of potential applications. Improving reliable data rates on hidden screen-camera channel, however, is non-trivial. Unlike conventional communication channels [8, 17, 22, 27], enhancing the signal does not help for the screen-camera channel because it immediately sets up

[1] Zheng Yang is the corresponding author.

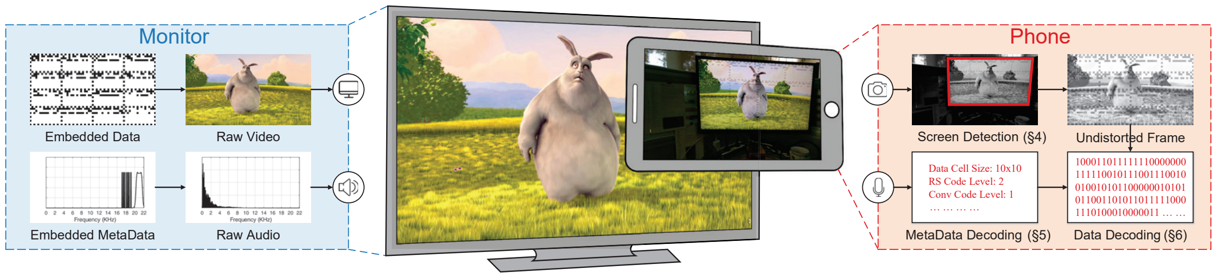


Figure 1: System overview of AIRCODE. The monitor displays a video with embedded data in raw video frames, and metadata (control information) in raw audio signals. The phone records the video, then detects screen in camera images to equalize distorted video frames, decodes metadata from audio signals, and finally decodes data embedded in video.

a conflict with unobtrusiveness to human eyes. We refer the reader to Section 2 for the more detailed analysis of the unique characteristics of the screen-camera channel. Essentially, we are facing the dilemma of foregoing three contradicting goals.

In this study, we first investigate the state-of-the-art works from distinct perspectives and discuss the root causes of unreliability. On this basis, we present AIRCODE, an imperceptible screen-camera communication system that supports high-throughput and reliable data transmission on top of regular video viewing, as demonstrated in Figure 1. The working scenario of AIRCODE is non-sophisticated: When a watcher intends to acquire information on the side channel, she simply shoots a video of the screen playing the encoded video using her smartphone, which automatically receives and decodes the embedded bits, at a rate of as high as 1069 Kbps, almost 2$\times$ higher than the state-of-the-art ChromaCode [41], underpinning various applications like video-in-video sharing. AIRCODE boosts data throughput while reducing BER by an invisible visual and inaudible audio dual-channel in three distinct ways:

(1) Precise screen tracking: Precise screen detection and tracking is critical to equalization of video frames, the key process for successful decoding of visual codes. According to our measurements, errors of a few pixels may significantly affect packet reception (§2). Precise screen detection, however, is non-trivial due to various factors, such as video contents, surrounding background, hand motions, etc. In AIRCODE, we exploit the idea of visual odometry [21] for this purpose. Specifically, AIRCODE constructs a 3-D map of a screen of interest and tracks the screen by estimating the phone pose with projections of map points and then projecting the screen with the pose estimated.

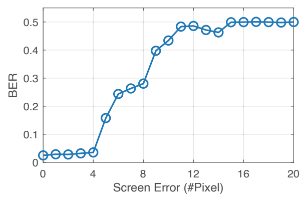
(2) Reliable audio channel: While the video channel shares the same frequency band with the ambient illuminance [41], the near-ultrasound audio channel is resistant to ambient low-frequency noises [38] and more reliable. Thus, AIRCODE allocates the inaudible speaker-microphone link as the control channel for the critical metadata of visual codes (e.g., code layout, coding scheme, etc.). To seamlessly comply with the video channel, the audio channel should fulfill two conditions, i.e., short packet duration that matches the high frame rate of the video channel, and low packet error rate (PER) that is required by the overall communication. To achieve this, AIRCODE carefully designs acoustic packets to overcome problems of reverberation and frequency spectral leakage caused by short packet duration and achieves high reliability with nearly zero PER.

(3) High-rate visual channel: AIRCODE leverages the screen-camera link as the high-rate data channel. Data bits are transmitted via imagery codes imperceptibly embedded in the primary carrier video. To minimize flickers, we adaptively choose the required lightness changes according to the spatial texture of the primary video content and perform lightness alteration as introduced in [36, 41]. The high data rate is then achieved by 1) embedding full-frame visual codes with a carefully designed frame structure; 2) effective error correction with an adaptive concatenated coding scheme.

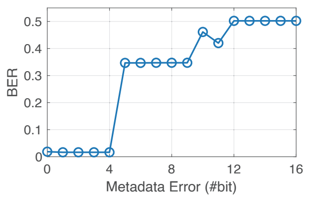
We implement AIRCODE using commodity computer monitors and smartphones. We conduct real-world experiments and extensive evaluations on key metrics including BER, throughput, goodput and screen-tracking accuracy. Besides, various system parameters, such as distance, angle, signal strength, background interference, and frame size, are tested. Videos with various texture, luminance, quality and audios with different types of sound are used for evaluation. The imperceptibility is tested with a user study. Experimental results demonstrate that AIRCODE achieves a remarkable data rate of 1069 Kbps with an average BER of 5%, which significantly outperforms existing approaches.

The core contributions of this paper are:

- We present AIRCODE, a hidden screen-camera communication system that achieves >1Mbps throughput for the first time, allowing not only text but also image and video transmission.
- To the best of our knowledge, we are the first to exploit an invisible visual and inaudible audio channel, which jointly enables fully imperceptible, high-rate, and reliable communication by their complementary advantages.
- We propose an algorithm using visual odometry that precisely tracks screen locations even with dynamic video content, complex ambient contexts, and camera motions caused by unconsciously hand motion and shake.

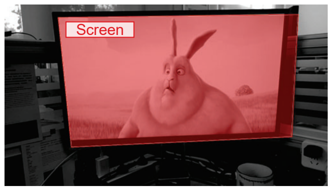


(a)

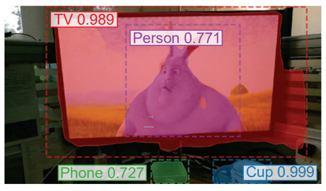


(b)

Figure 2: Key factors for screen-camera link quality. (a) Screen localization error, and (b) Metadata error.



(a)



(b)

Figure 3: Examples of screen detection failure. (a) rule-based approach [41], and (b) learning-based approach [10].

2 Motivation

Hidden screen-camera communication is possible with the natural difference between the human vision system and camera system: Human eyes cannot resolve flickers, or luminance fluctuations, at frequencies higher than 100Hz, but instead only perceive the average lightness, due to the low-pass flicker-fusion property of human vision system [31]. Thus, if we inversely modify the lightness of a pair of subsequent video frames, which are termed as complementary frames, and play them at high frame rate, e.g., 120 frames per second (fps) that is supported by modern commercial monitors and TVs, human eyes will not perceive the change of lightness. Instead, they will only observe the original video content, which is the average of complementary frames. In contrast, commodity cameras with a high capturing rate can acquire the differences and decode the data if certain information is modulated on top of the change of lightness. Pioneer work has exploited this phenomenon for hidden screen-camera communication by embedding data bits unobtrusively in primary carrier videos. The visual channel of AIRCODE is also built upon this idea. The hidden screen-camera channel is reported error-prone due to some unique intrinsic errors [24, 36, 41], such as projection distortions, blurring, Moiré patterns, rolling shutter effects, and frequent changes of camera pose caused by hand motion and shake, etc.

2.1 Challenges and Measurements

Previous proposals mainly aim at minimizing flickers for good unobtrusiveness, yet sacrifice data rates and robustness. The best result to date reported is from ChromaCode, with the data rate of up to 551 Kbps and BER of 8%, which is far from sufficiency for many applications such as the above-mentioned video-in-video sharing. What's even worse, such performance is obtained with limitations including visible border markers [36] and stable cameras [24, 41].

To reveal the root causes of these limitations, we consider the state-of-the-art works, ChromaCode, and evaluates how its reliability suffers when (1) the detected screen deviates from the ground-truth, and (2) the received metadata (e.g. visual code layout, coding scheme, etc.) has bit errors. Experiments are conducted in a cubicle space and with different types of videos, as listed in Table 1. During experiments, the phone is placed static with several fixed poses, so that the ground-truth of the screen can be manually marked. Besides, the metadata is known prior as well. To evaluate the impact of screen detection, we gradually shift the input of the screen location by a small number of pixels and calculate BER accordingly. To evaluate the impact of metadata decoding, we manually add random error bits in the channel code of metadata.

(1) Impact of screen detection. Figure 2a shows that the BER of the screen-camera link remains around 2% when the location error of the screen is less than 4 pixels (in a 720p video), which is about the half-length of the smallest data cell in ChromaCode, but explodes when the location error exceeds 4 pixels. It fails when the location error reaches 15 pixels. It means that highly accurate screen detection is necessary for the screen-camera link. Recent phones incorporates image stabilization [20, 32] to reduce blurring caused by camera motion. However, these techniques can only remove small handshakes and thus require the user to uncomfortably hold the phone at a fixed position during the recording. Figure 3 further illustrates failure cases of screen detection with two representative approaches, the rule-based approach used in ChromaCode, and Mask R-CNN [10], a learning-based approach. The rule-based approach tries to find a quadrilateral whose outer bound consists of edges in the image and has consistent similar lightness, which may be misled by video contents and the surrounding environment. While Mask R-CNN can accurately recognize objects in images after sufficient training, it fails to meet the stringent requirement on accurate segmentation with errors of only several pixels for screen-camera communication. Besides, both approaches lack pertinent mechanisms to deal with continuous tracking of the screen in a video.

(2) Impact of metadata decoding. Figure 2b shows that the screen-camera link becomes lost when the number of error bits in metadata exceeds 4, which is consistent with that ChromaCode encodes 5-bit metadata with 15-bit code and corrects at most 3-bit errors. Through all experiments, 24.6% frames have BERs over 8%, among which 29.3% are due to failure in decoding metadata. The drawback of existing sole screen-camera communication schemes is that while metadata is more important than data and requires robust communication with lower BER, it is conveyed in the same erroneous visual channel as data. To achieve reliable communication with the screen-camera link, a more robust channel is needed for transmission of metadata.

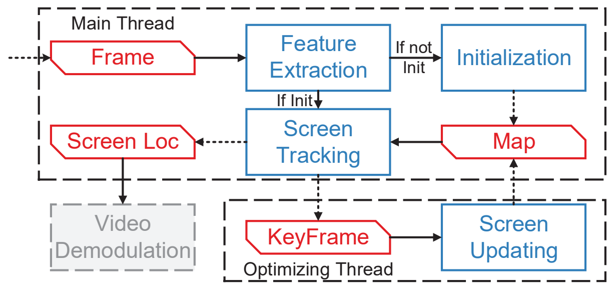


Figure 4: Logic flow of screen detection.

3 System Overview

Motivated by the measurements above, we design AIRCODE to address the two significant drawbacks, namely erroneous screen tracking and vulnerable transmission of metadata and deliver an imperceptible, high-rate, and reliable screen-camera communication system.

Figure 1 shows the system overview of AIRCODE. At the sender side, AIRCODE embeds data (visual codes) into raw video frames and embeds metadata of visual codes into raw audio signals. During playing the video, users run AIRCODE on their phones and shoot the monitor. At the receiver side, upon receiving camera images and acoustic signals, AIRCODE simultaneously tracks the screen in images with visual odometry to equalize distorted frames (§ 4) and decodes metadata embedded in acoustic signals (§ 5). Afterward, both undistorted frames and metadata are used to decode data conveyed in video frames (§ 6).

4 Screen Detection

Detecting screen is the key process for successful decoding of visual code, due to unknown perspective distortion of the screen in images. However, various video content and background environments may interfere in screen detection, leading to failure of decoding, as discussed in § 2. Existing works embed specific position codes at the edges and corners of video for screen detection, which wastes communication resources and affects the watching experience of audiences. In contrast to existing solutions that try hard to reduce the interference caused by complex background environment, AIRCODE exploits it as visual clues for odometry [21] to track the screen in frames.

Visual odometry alternately tracks the poses of image frames and maintains a global map of 3-D points seen by multiple image frames. Generally, visual odometry consists of three main steps. (1) Initialization. Visual odometry matches feature pixels of two image frames and estimates their relaive pose. The global map is then initialized with the 3-D map points of the matched feature pixels. (2) Tracking. Upon receiving a new frame, visual odometry matches its feature pixels with 3-D points in the global map, estimates the pose of the frame, and updates 3-D map points of the matched feature pixels in the global map. (3) Optimization. Visual odometry periodically buffers some frames, termed as keyframes, to refine the global map and avoid drifting error. When a new keyframe arrives, visual odometry jointly optimizes poses of all buffered keyframes that share common feature pixels with the new keyframe and all 3-D points seen by these keyframes in the global map.

To enable screen detection, AIRCODE modifies the main steps of visual odometry to further keep 4 screen points in the global map. Each screen point corresponds to one corner point of the rectangular screen. Thus, the screen can be uniquely identified and represented by its four corner points. Figure 4 shows the logic flow of screen detection of AIRCODE. In addition to visual odometry, during initialization, AIRCODE estimates the initial 3-D locations of the screen points. During the tracking process, AIRCODE projects the screen points in the frame with the frame pose for decoding. In the optimization process, AIRCODE updates the screen points with the refined poses of keyframes and the projections of the screen points on these keyframes to avoid drifting error.

4.1 Feature Extraction

Feature pixels are representative pixels invariant to translation, scaling and rotation of an image and can be used for robustly tracking successive image frames and mapping 3-D points corresponding to these feature pixels. AIRCODE uses ORB [28] which is fast to compute and match, and remains invariant across different viewpoints, yielding sufficient efficiency and accuracy for visual odometry.

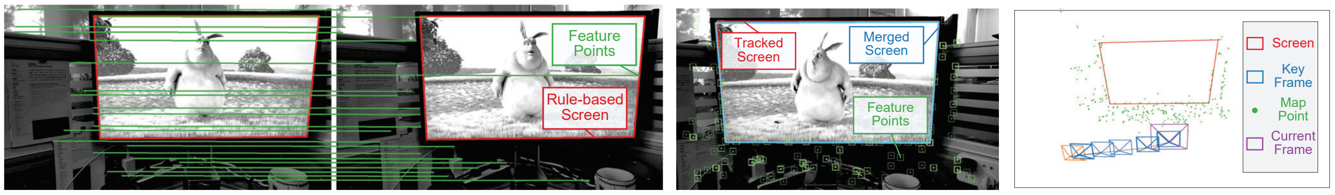
In practice, the video content may change with time, and feature pixels within the screen are not consistent across successive frames. To avoid the negative impact of mismatching these feature pixels, AIRCODE first obtains a coarse estimation of the screen frame, and filter out all feature pixels within it. The initial screen frame is calculated by the rule-based algorithm during initialization (as in § 2) or inherited from the last frame during the tracking process. Figure 5a shows examples of feature extraction, where the screen frame is highlighted and feature pixels within it are removed.

4.2 Initialization

The initialization process computes the relative pose between two frames and triangulates an initial set of 3-D map points for tracking and 4 screen corner points for communication. Lacking the knowledge of screen points, AIRCODE uses the rule-based algorithm to detect the screen and decode frames during initialization. As a successful decoding indicates accurate screen detection, AIRCODE selects two decoded frames with their detected projections of the screen for initialization.

Figure 5a illustrates an example of initialization with two matched frames. Denote the two selected frames as F_0 and F_1, any matched feature pixels as $\mathbf{p}_0$ and $\mathbf{p}_1$, and any matched projection pixels of screen corner points as $\mathbf{s}_0$ and $\mathbf{s}_1$, AIRCODE uses epipolar geometry and computes the fundamental matrix $\mathbf{F}_{10}$ that connects any pair matched pixels [9]:

$$\mathbf{p}_0^T \mathbf{F}_{10} \mathbf{p}_1 = 0, \quad \mathbf{s}_0^T \mathbf{F}_{10} \mathbf{s}_1 = 0. \tag{1}$$



(a) (b) (c)

Figure 5: Key steps of screen detection: (a) Out-of-screen feature extraction and matching for initialization, (b) frame tracking with matched feature pixels and screen tracking with frame pose, and (c) global map and key frames for screen updating.

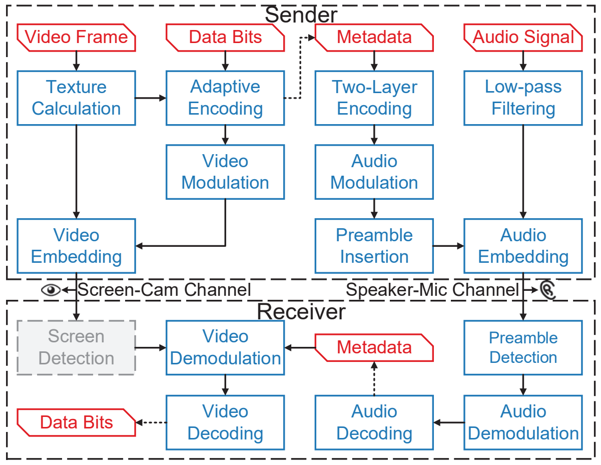


Figure 6: Logic flow of dual-channel communication.

F_{10} can be solved by integrating eight pairs of matched pixels [18]. Since the successful decoding strongly indicates that screen points are accurately located in the two frames, AIRCODE selects the projection pixels of four screen points (i.e., $\mathbf{s}$) as four pairs of the matched pixels, and four pairs of matched feature pixels (i.e., $\mathbf{p}$) as the rest.

To recover the camera pose, the fundamental matrix $\mathbf{F}_{10}$ is converted to the essential matrix $\mathbf{E}_{10}$ using the intrinsic matrix $\mathbf{K}$ of the camera:

$$\mathbf{E}_{10} \equiv \mathbf{t}_{10}^{\wedge}\mathbf{R}_{10} = \mathbf{K}^T\mathbf{F}_{10}\mathbf{K}, \tag{2}$$

where $\mathbf{t}_{10}$ and $\mathbf{R}_{10}$ the are relative translation vector and rotation matrix respectively, and the operator $(\cdot)^{\wedge}$ is to calculate the skew-symmetric matrix from a vector. The four possible poses (i.e., $\mathbf{t}_{10}$ and $\mathbf{R}_{10}$) are derived from the essential matrix $\mathbf{E}_{10}$ via singular value decomposition [9]. With each ambiguous candidate, AIRCODE triangulates 3-D points corresponding to all matched feature pixels and projection pixels of four screen points. A valid 3-D point with high confidence should be in front of the camera and have a significant parallax between the two frames. Thus, AIRCODE selects the candidate with most such points as the true pose. Finally, AIRCODE initializes the 3-D global map and the screen with all valid 3-D points of the true pose. The two frames used in initialization are set as initial keyframes.

4.3 Screen Tracking

After initialization, AIRCODE continuously tracks poses of new frames and projects the screen accordingly for decoding. As visual odometry, AIRCODE optimizes the pose of the current frame by minimizing the location error between the projections of map points and their matched feature pixels in the frame. Specifically, suppose that the pose matrix of the i-th frame is $\mathbf{T}_i = [\mathbf{R}_i \mid \mathbf{t}_i]$, where $\mathbf{R}_i$ and $\mathbf{t}_i$ are the corresponding rotation matrix and translation vector, and N matches $\langle \mathbf{P}_j, \mathbf{p}_{i,j} \rangle$ are detected, where $\mathbf{P}_j$ is the j-th map point and $\mathbf{p}_{i,j}$ is the matched feature pixel in the i-th frame, the pose $\mathbf{T}_i$ can be optimized via bundle adjustment [35]:

$$\mathbf{T}_{i,opt} = \arg\min_{\mathbf{T}_i} \sum_{j=1}^{N} \|\mathbf{p}_{i,j} - \pi(\mathbf{T}_i, \mathbf{P}_j)\|^2, \tag{3}$$

where $\pi(\cdot)$ projects the 3-D map points onto the image frame given its pose [9]:

$$\pi(\mathbf{T}_i, \mathbf{P}_j) = [\frac{(\mathbf{K}(\mathbf{R}_i\mathbf{P}_j + \mathbf{t}_i))_0}{(\mathbf{K}(\mathbf{R}_i\mathbf{P}_j + \mathbf{t}_i))_2}, \frac{(\mathbf{K}(\mathbf{R}_i\mathbf{P}_j + \mathbf{t}_i))_1}{(\mathbf{K}(\mathbf{R}_i\mathbf{P}_j + \mathbf{t}_i))_2}]^T. \tag{4}$$

With the estimation of the pose, AIRCODE projects screen points onto the current frame:

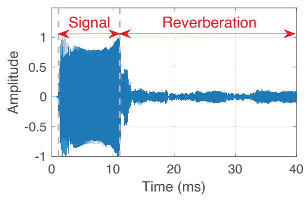
$$\mathbf{s}_{i,j} = \pi(\mathbf{T_i}, \mathbf{S}_j), \tag{5}$$

where $\mathbf{S}_j$ is the j-th screen point ($j = 1,2,3,4$). To further minimize projection error, AIRCODE searches outstanding Shi-Tomasi corners [34] within neighborhoods of these projections as the final estimation.

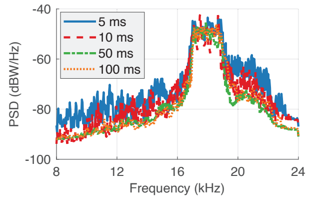
AIRCODE may fail to decode due to small but intolerant deviations of screen tracking. In contrast, despite frequent failures, the rule-based algorithm can accurately detect edges and corners in the frame and yield more accurate estimation if the screen is successfully detected. Thus, when decoding fails, AIRCODE further executes the rule-based algorithm to obtain a second estimation of the screen, denoted as $\mathbf{s}'_{i,j}$. Then, it merges the corner points of the two screens as:

$$\mathbf{s}''_{i,j} = \begin{cases} \mathbf{s}'_{i,j} & \|\mathbf{s}_{i,j} - \mathbf{s}'_{i,j}\| \geq \delta_h \\ \mathbf{s}_{i,j} & \|\mathbf{s}_{i,j} - \mathbf{s}'_{i,j}\| \leq \delta_l \end{cases}, \tag{6}$$

As screen tracking yields consistently small errors, an upper threshold, δ_h, is used to reject totally wrong estimation from the rule-based algorithm. Meanwhile, a lower threshold, δ_l, is used to accept more accurate estimation from the rule-based algorithm. When $\delta_l < \|\mathbf{s}_{i,j} - \mathbf{s}'_{i,j}\| < \delta_h$, AIRCODE has no extra information to determine which estimation is better. Thus, it forks two screen candidates, whose j-th points are assigned as $\mathbf{s}_{i,j}$ and $\mathbf{s}'_{i,j}$ respectively. Finally, AIRCODE tries all screen candidates until the frame is successfully decoded. Note that in the worst case where the screen tracking fails, e.g., due to lack of visual features around the screen, AIRCODE



(a) Reverberation



(b) Spectral Leakage

Figure 7: Main problems of short packet duration. (a) enlarged impact of reverberation, and (b) aggravated spectral leakage.

degenerates to the rule-based algorithm and still maintains a moderate decoding rate. Figure 5b illustrates the process of screen tracking, where all matched feature pixels in the frame and both the tracked screen and the merged screen are highlighted.

To avoid drifting error during tracking, AIRCODE periodically selects frames and passes them to the optimizing thread. Specifically, it creates new keyframes only when one of the following conditions are satisfied:

1. More than 0.5 fps frames have passed after the creation of the last keyframe, and the current frame tracks less than 90% points than the last keyframe.
2. The current frame fails to be decoded with the tracking screen but can be decoded with the merged screen.

Condition (1) avoids computing cost with redundant frames, while condition (2) ensures timely updating of the screen.

4.4 Screen Updating

The optimizing thread periodically refines map points and screen points to avoid drifting error. Figure 5c shows the structure of global map and keyframes. Upon receiving a new keyframe, AIRCODE first deletes obsolete map points that are seen by fewer than 3 keyframes and add new points from matches between the current keyframe and previous keyframes, as that in visual odometry. Next, AIRCODE collects all map points seen by the current keyframe and all keyframes that see any of these map points for optimization. Specifically, suppose M key frames together with the current keyframe are selected, the pose of the i-th keyframe is $\mathbf{T}_i$, and N_i matches $\langle \mathbf{P}_{i_j}, \mathbf{p}_{i,j} \rangle$ are detected in the i-th keyframe. The pose $\mathbf{T}_i$ and map points $\mathbf{P}_k$ are optimized via bundle adjustment:

$$\{\mathbf{T}_{i,opt}\}_{i=0}^{M}, \{\mathbf{P}_{\mathbf{k},\mathbf{opt}}\}_{k=1}^{K} = \underset{\{\mathbf{T}_i\}_{i=0}^{M}, \{\mathbf{P}_k\}_{k=1}^{K}}{\arg\min} \sum_{i=1}^{M} \sum_{j=1}^{N_i} \|\mathbf{p}_{i,j} - \pi(\mathbf{T}_i, \mathbf{P}_{i_j})\|^2. \tag{7}$$

Then, AIRCODE optimizes screen points via multi-view triangulation with the refined poses of keyframes. Specifically, denote the projection of the j-th screen point $\mathbf{S}_j$ on the i-th key frame as $\mathbf{s}_{i,j}$, the 3-D location of the j-th screen point is calculated by minimizing the following projection error:

$$\mathbf{S}_{j,opt} = \underset{\mathbf{S}_j}{\arg\min} \sum_{i=1}^{M} \|\mathbf{T}_i \bar{\mathbf{S}}_j - (\hat{\mathbf{s}}_{i,j}^T \mathbf{T}_i \bar{\mathbf{S}}_j)\hat{\mathbf{s}}_{i,j}\|^2, \tag{8}$$

where $\hat{\mathbf{s}}_{i,j} = \frac{\mathbf{K}^{-1}\bar{\mathbf{s}}_{i,j}}{\|\mathbf{K}^{-1}\bar{\mathbf{s}}_{i,j}\|}$ is the direction vector of the projection $\mathbf{s}_{i,j}$, and $\bar{\mathbf{S}}_j$ and $\bar{\mathbf{s}}_{i,j}$ are the homogeneous representation of $\mathbf{S}_j$ and $\mathbf{s}_{i,j}$, respectively. The term $(\hat{\mathbf{s}}_{i,j}^T \mathbf{T}_i \bar{\mathbf{S}}_j)\hat{\mathbf{s}}_{i,j}$ calculates the projection coordinate of $\mathbf{S}_j$ along the direction $\hat{\mathbf{s}}_{i,j}$. Intuitively, when $\mathbf{S}_j$ is accurately localized, $\mathbf{T}_i\bar{\mathbf{S}}_j$ has the same direction as $\hat{\mathbf{s}}_{i,j}$ and the error term becomes 0. The optimal solution $\mathbf{S}_{j,opt}$ is the eigenvector corresponding to the minimal eigenvalue of the matrix $\sum_{i=1}^{M}(\mathbf{T}_i - \hat{\mathbf{s}}_{i,j}\hat{\mathbf{s}}_{i,j}^T\mathbf{T}_i)^T(\mathbf{T}_i - \hat{\mathbf{s}}_{i,j}\hat{\mathbf{s}}_{i,j}^T\mathbf{T}_i)$, and can be calculated directly in closed form.

Finally, after optimization, AIRCODE discards redundant keyframes whose map points have been seen in at least other three keyframes, as in [33], to maintain a reasonable number of keyframes for fast optimization, and avoid large estimation uncertainty caused by co-located key frames [19].

5 Audio Control Channel

Despite orders of magnitude smaller throughput, the speaker-microphone link is more reliable than the screen-camera link due to little interference in the near ultrasound band. Thus, AIRCODE exploits the speaker-microphone link to send critical metadata of visual codes, e.g., coding layout, coding scheme, etc. By doing so, not only is metadata robustly delivered, but also more video coding area is saved to convey more data. Figure 6 shows the logic flow of the audio communication part. At the sender side, metadata is encoded with a two-layer encoding scheme, which includes Golay coding and Manchester coding, and then modulated on frequency subcarriers. A chirp signal is prepended as a preamble for timing alignment. The raw audio signal is low-pass filtered and then embedded with the control signal. At the receiver side, the signal is first high-pass filtered to remove raw audio signal and background noises, then aligned with a chirp template, and finally demodulated and decoded to yield metadata.

5.1 Design Challenges

Frame-level Packet Duration. To convey metadata for the video channel, whose data frame (a pair of complementary video frames) rate is 60 fps, the audio packets should be sent within tens of milliseconds. However, most existing solutions send packets at the second level, which is sufficiently reliable but cannot be agilely adjusted according to the video channel. For example, Dolphin [38] adopts long OFDM packets, which is 3.56 s long. Chirp packets, whose duration is 1.463 s, are used in [12]. To match the need for the high packet rate, a shorter packet should be designed.

However, reducing packet duration will cause two problems, reverberation in the time domain and spectral leakage in the frequency domain. First, the impact of reverberation is enlarged. Due to the multi-path effect, the microphone not only hears the direct signal but also delayed reverberating signals reflected by surrounding environments. Figure 7a shows an example of a 10 ms chirp signal received and its reverberations.

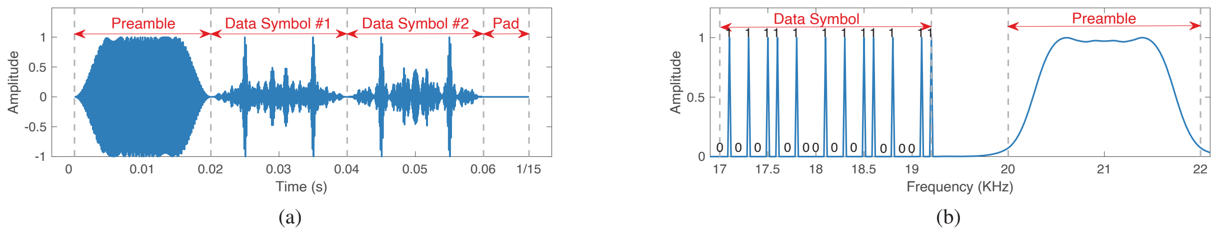


Figure 8: Audio packet design. (a) The 66.7 ms packet consists of one 20 ms preamble, two 20 ms data symbols and one silence pad. (b) The preamble and data symbol are separately located within 2 ranges of 20 – 22 kHz and 17 – 19.2 kHz, the data symbol consists of 23 subcarriers spaced at 100 Hz apart.

The dominant echo, spans the first 3 ms of the reverberation period, making it impossible to modulating bits at the sub-millisecond level in the time domain. Second, the spectral leakage aggravates as the length of bit duration decreases. Figure 7b depicts the spectrum of chirp signals modulated with on-off keying with different bit duration. Though with a duration of 5 ms, the leakage significantly increases, leading to the perception of the audience.

Low Packet Error Rate. AIRCODE assigns the audio link as the control channel. Thus, it must be reliable with a low packet error rate (PER). As audio channel suffers from multipath effect and frequency selectivity of speakers and microphones, pilot symbols are usually used to account for different channel responses of subcarriers and decide demodulation thresholds accordingly [38]. However, the threshold-based method requests that all bits '1' across all symbols are modulated with the same amplitude, leading to waste of energy and low SNR, when some symbols contain only a few bits '1'.

5.2 Audio Packet Design

Figure 8a shows the audio packet format in both time and frequency domain. To cope with the video channel, we set the packet duration as $\frac{1}{15}$ s, corresponding to 4 data frames, during which video configuration is unlikely to change.

To solve the challenge of short packet duration, AIRCODE modulates bits in the frequency domain and sets data symbol length as 20 ms, which is about 4 times longer than the major echo and has low spectral leakage. To further avoid the impact of reverberation, AIRCODE separates the preamble and the data into different frequency bands. Specifically, as shown in Figure 8b, the chirp preamble is within 20 – 22 kHz, and the data symbol is within 17 – 19.2 kHz. Both of which are inaudible to most human ears [42]. To further reduce spectral leakage, AIRCODE applies a tapered-cosine window on the preamble and each data symbol.

To solve the challenge of low packet error rate, AIRCODE adopts the two-layer coding scheme. Specifically, each packet contains 12 control bits. The 12 control bits are first encoded into 23 bits codeword via Golay code [23], which can correct any 3-bit error. Then, to fully exploit signal power allocated to the control signal, AIRCODE further adopts Manchester coding, which encodes bits '1' as '10' and bits '0' as '01'. During modulation, AIRCODE assigns 22 Manchester bits of the first 11 bits and the first Manchester bit of the 23rd bit of the Golay codeword to the 23 subcarriers of the first data symbol, and the rest 23 Manchester bits to the second data symbol. Figure 8b shows an example of the spectrum of one data symbol, where the subcarrier spacing is 100 Hz. Since each pair of Manchester bits are assigned to either adjacent subcarriers or data symbols, they experience similar channel responses caused by multipath effect and frequency selectivity of audio devices. Thus, the receiver can demodulate the data symbol by directly comparing amplitudes of pairs of Manchester bits without thresholds, and the sender can distribute all allocated signal power evenly to bits '1' in each data symbol, to increase average SNR and reduce overall PER.

6 Video Data Channel

The screen-camera link supports high throughput communication with imagery codes invisible to human eyes. The basic principle is to exploit the low-pass flicker fusion property of human eyes. Specifically, screen-camera communication generates high-rate complementary frames by inversely modifying the lightness of a pair of adjacent raw video frames. When videos are played at high frame rate, e.g., 120 FPS, human eyes can only observe raw video frames, which are the average of complementary frames, and cannot perceive the embedded data frames. In contrast, cameras with a high capturing rate can still acquire and decode data frames. AIRCODE takes the idea of hidden screen-camera communication, but focus on lower BER as well as higher data rate, by incorporating robust speaker-microphone link as the control channel, and accurate screen detection based on visual odometry.

Figure 6 shows the video communication part of AIRCODE. At the sender side, data bits are firstly encoded with a concatenated coding scheme, including Reed-Solomon (RS) coding as source code and Convolutional coding as channel code. The coding scheme is adaptively selected according to the texture complexity of the raw video, to achieve high throughput in plain frames, as well as low BER in textured frames. The encoded bits are modulated as data frames and embedded into pairs of complementary raw video frames. At the receiver side, AIRCODE inputs camera images into the screen detection process to obtain successive locations of the screen in frames. Meanwhile, the associated audio signal is pro-

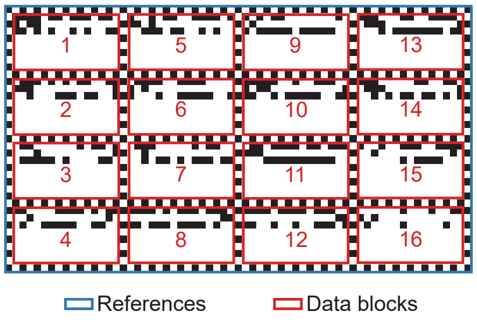


Figure 9: Data frame design. The frame consists of 16 data blocks, surrounded by black-and-white lines as references.

cessed to get corresponding metadata. Then, camera images are equalized with screen location information, and data is finally demodulated and decoded with metadata.

6.1 Data Frame Design

Figure 9 shows AIRCODE's data frame. AIRCODE follows the basic design of ChromaCode, but removes the code preamble blocks in ChromaCode that redundantly encode metadata of data frames, thanks to the use of the audio channel. AIRCODE divides the *whole* image frame into 16 data blocks with equal size. Similar to ChromaCode, each data block contains several data cells and is surrounded with alternate black and white cells as references. The encoded bits are interleaved and filled into data blocks. Specifically, the i-th bit is assigned to the $\lfloor \frac{i}{16} \rfloor$-th cell in the $(i \mod 16)$-th block.

The metadata of a data frame, including the levels of RS coding and Convolutional coding for encoding, and the data cell size for modulation, are transmitted as control packets over the reliable acoustic channel. Each control packet of metadata contains 12 data bits, with 2 bits for the RS coding level, 1 bit for the Convolutional coding level, 6 bits for the data cell size, and the rest 3 bits for parity bits for these three parameters. To save audio channel capacity, four successive data frames (embedded in eight video frames) share the same set of parameters in one control packet, as the video content is unlikely to drastically change during such a short period.

To avoid floating time offsets between the audio control channel and the video data channel, AIRCODE periodically (e.g., 5 s) sends video frames with the highest coding level and audio packets. These video frames and audio packets contain the synchronization information. At the receiver, each control audio packet will be aligned with 4 video data frames according to the latest synchronizing frame.

6.2 Adaptive Error Correction

In practice, textures of carrier video can considerably impact data transmission. In particular, the BER of the screen-camera link is lower in plain frames consisting of pixels in similar colors but significantly higher in textured frames. Therefore, to ensure robust data transmission, an error correction scheme with high correction ability should be used for textured frames. In contrast, a scheme with high efficiency yet potentially low correction ability will be preferred in plain frames to achieve high throughput.

This observation leads to the adaptive error correction scheme of AIRCODE, which adjusts the screen-camera channel parameters in metadata, i.e., RS coding level, Convolutional coding level, and data cell size, according to the dynamic texture complexity of the carrier video. In detail, AIRCODE supports 40 levels of data cell size, 2 levels of Convolutional coding and 4 levels of RS coding, which result in 320 levels of error correction. Given a video, a subset of error correction levels are selected and linearized by setting the data cell size as the most significant factor and the RS coding as the least significant factor. The video is segmented where a segment can be as short as 8 frames (i.e., $\frac{1}{15}$ s), corresponding to 1 audio packet. For each video segment, AIRCODE quantizes the texture complexity by calculating the average contrast of pixels against their neighboring 9×9 pixels, and maps the texture complexity to the error correction level.

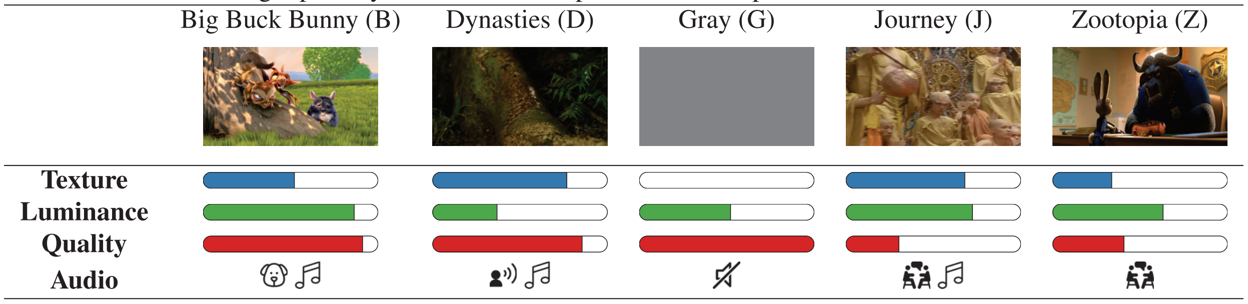
7 Evaluation

7.1 Experimental Methodology

Experiment Setting and implementation. We use an AOC AGON AG271QX monitor as the sender of encoded videos and audios. It supports 120Hz refresh rate along with 1920×1080 resolution and has two audio speakers on its back. One Nexus 6P smartphone without hardware modification is used as the receiver. videos and audios are played by a DELL XPS 8900-R17N8 desktop, which equips a GeForce RTX 2070 GPU, and MPV player with VDPAU acceleration. AIRCODE uses FFmpeg to extract and compress video frames and audio tracks. OpenCV is used for the implementation of the rule-based screen detection algorithm and the screen tracking algorithm. AIRCODE exploits MediaCodeC interface for efficient hardware video coding and Boost library for implement of audio coding.

Video & Audio Selection. Table 1 shows the selected videos and their audios. For the video with pure gray images, an empty audio track is used as its origin audio. The bars indicate the levels of texture complexity, luminance and quality of the selected video contents, which cover a variety of combination. The more portion of a bar is filled with colors, the higher level of the metric is achieved by the corresponding video. E.g., the video Dynasties (D) has complex texture, low luminance and high quality. The corresponding audio contents include different types, e.g., human speaking, animal sound, an object moving and dropping, and background music, indicated by the icons. In practice, the quality of a streaming video frequently switches. However, frames between two adjacent switches still have the same quality. As AIRCODE encodes data at the frame level, frequent switching of streaming videos has a minor impact on AIRCODE.
Evaluation Metrics. For screen-camera communication, we compare AIRCODE with ChromaCode and InFrame++ [36]

Table 1: Origin primary video & audio clips used in the experiments and their characteristics


Big Buck Bunny (B)
Dynasties (D)
Gray (G)
Journey (J)
Zootopia (Z)
Texture
Luminance
Quality
Audio

in terms of throughput, goodput and data BER. Throughput is the amount of all received bits. Goodput is the effective throughput of correctly decoded data bits. And BER is the number of error bits divided by total received data amount. ChromaCode uses the rule-based screen detection algorithm, and InFrame++ uses visible markers and alignment patterns at screen corners and borders. Since the speaker-microphone link serves as control channel and does not convey much data, we only evaluate its reliability in terms of BER and PER.

User Study. We invite 12 participants, including students and staffs on campus, to score the watching experience of encoded videos and audios. The ages of these participants range from 20 to 30. Nine participants are male, while the rest 3 are female. In total 40 videos together with their audios are prepared and tested. Videos and audios with or without data are randomly shuffled and played. Participants, not knowing their orders, are asked to give scores on the quality of them. Denote the scores for encoded videos and audios as *S_enc* and that for raw videos and audios as *S_raw*, the final scores for encoded videos and audios are normalized as $\frac{S\_enc}{S\_raw}$. In some cases, the normalized score is larger than 1, meaning that participants cannot statistically differentiate encoded videos and audios from their original counterparts. Moreover, to check whether young children can perceive the embedded code, we intentionally invite one 5-year-old child to watch the testing videos and audios and give feedback. Considering that children at such a young age may not have enough comprehension ability as adults to understand the meaning of the experiment, we neutrally ask him whether he can see or hear anything peculiar in videos or audios. All experiments are approved by our IRB, and do not raise any ethical issues.

7.2 Overall Performance

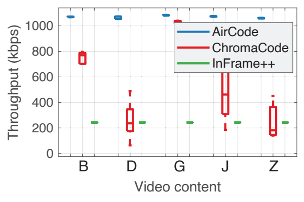
Performance of the communication processes. We first report the overall performance of AIRCODE in terms of throughput, goodput, and BER of the video channel, and BER and PER of the audio channel.

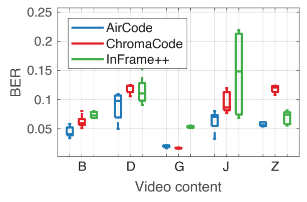
Figure 10a-10c show the performance of screen-camera communication in AIRCODE with different video contents. Two state-of-the-art approaches, ChromaCode and InFrame++ are evaluated for comparison. As shown in Figure 10a, AIRCODE achieves 1 Mbps throughput for all 5 videos, attributed to robust transmission of metadata through speaker-microphone channel. In contrast, though encoded with the whole screen, ChromaCode has lower throughput, especially with the videos 'D' and 'Z'. InFrame++ has the lowest throughput, due to its inefficiently CDMA-like coding scheme. Figure 10b shows that except the video 'D', the average BER achieved by AIRCODE is about 5%, which is statistically lower than its counterparts, owing to the accurate screen detection process. Figure 10c further compares the goodput of AIRCODE and ChromaCode. Due to the additive effect of accurate screen detection and robust metadata transmission, AIRCODE outperforms ChromaCode throughout all testing videos. Another key observation from Figure 10b and 10c is that the system performance is highly impacted by video content, where the video with more plain frames (i.e., 'G') tends to have lower BER and higher goodput. AIRCODE takes adaptive error correction, which further increases the reliability of AIRCODE with different video contents.

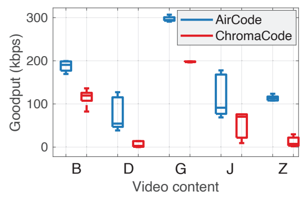
Figure 10d shows the BER and PER of the speaker-microphone link, whose reliable communication is of great importance for the overall system. Across all types of audio contents, the BER of the audio control channel consistently remains below 1%. With the help of Golay code, over 99.9% audio control packets can be successfully decoded, providing robust metadata for the screen-camera link.

Performance of the screen tracking process. The screen detection rate and tracking error of AIRCODE are evaluated and compared with the rule-based algorithm of ChromaCode. Since labeling ground-truth of all video frames is labor-intensive, we indirectly calculate screen detection rate as the ratio $\frac{r_{method}}{r_{base}}$, where r_{method} is the frame decoding rate of the method, and r_{base} is the frame decoding rate when the camera is fixed at some sampling locations. To calculate screen tracking error, we randomly sample frames with 0.5s intervals from all videos and mark the ground-truth screen in them for comparison.

On average, AIRCODE takes 0.79 s to initialize. Figure 11a shows that AIRCODE consistently detects the screen in over 97% frames across different types of videos. In contrast, the rule-based algorithm suffers from confusing video contents, and ChromaCode can detect the screen in over 90% frames in videos 'B', 'G' and 'J', but only about 70% frames in videos 'D' and 'Z'. Figure 11b further shows the tracking error of the two methods in terms of image pixels. AIRCODE achieves


(a) Video throughput


(b) Video BER


(c) Video goodput

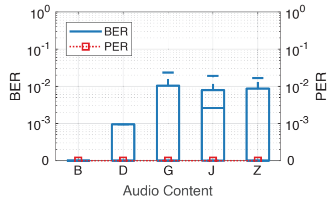
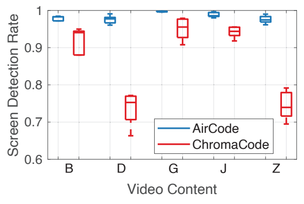

(d) Audio BER & PER

Figure 10: Performance of the screen-camera communication and the speaker-microphone communication.


(a) Screen detection rate

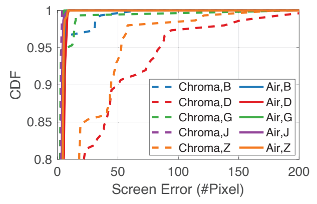
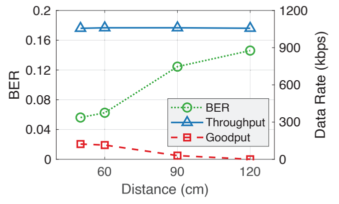

(b) Screen tracking error

Figure 11: Performance of screen tracking.


(a) Video

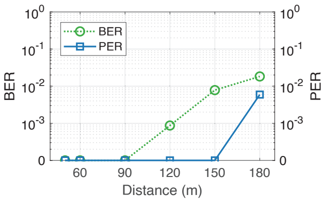

(b) Audio

Figure 12: Impact of monitor-phone distance.

Table 2: Benefits of individual module.

		Throughput (kbps)	BER (%)	Goodput (kbps)
Screen Detection	**Tracking**	1086.4	6%	139.5
	Rule	1073.8	8.3%	77.3
Control Channel	**Audio**	1084.3	4.4%	149.1
	Video	893.7	4.6%	144.3
Error Correction	**Adaptive**	1086.3	5%	159.4
	Fixed	1086.4	4.4%	149.4

consistently small tracking error for all videos, while the performance of ChromaCode is highly related to video content. Specifically, all tracking errors of AIRCODE are within 9 pixels. In contrast, 12.8% tracking errors of ChromaCode exceed 9 pixels, and the maximal error is even beyond 60 pixels. This validates the effectiveness of screen tracking.

Benefits of individual module. We further evaluate the individual performance improvement from the screen detection, control channel and error correction of AIRCODE. When one module is evaluated, the other modules are disabled (error correction) or assumed to be perfect (screen detection and control channel). Table 2 shows the average system performance. First, the screen tracking method of AIRCODE has lower BER and significantly higher goodput than the rule-based method of ChromaCode, thanks to its consistently accurate screen tracking. Second, comparing with sending metadata via video, using the audio channel improves throughput and goodput. On one hand, it saves the area where metadata has to be duplicated for reliable detection in video frames. On the other hand, the audio channel is more robust and overcomes the occasion when even the duplicated metadata in video frames cannot be correctly decoded. Third, by considering the texture complexity of videos, although the adaptive error correction of AIRCODE has a slightly higher BER than error correction with fixed coding level, it achieves overall 10 kbps more goodput, as more data bits are adaptively encoded.

7.3 Parameter Study

Distance. Figure 12 shows the impact of distance between the monitor and the phone. For the video part, the data cell size is fixed to 10×10. As shown in Figure 12a, the goodput of AIRCODE abruptly decreases when the distance becomes longer than 90 cm. One main reason is that small data cells become hard to recognize when the distance increases. It suggests that larger data cells should be used at longer distances, however, with the sacrifice of the throughput. The other reason for performance deterioration is that the focal length of the receiving camera will automatically change more frequently with larger distances, leading to more blurry video records.

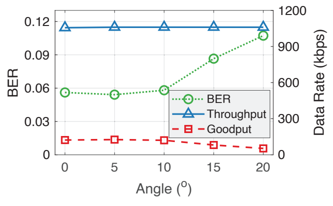
For audio part, as shown in Figure 12b, while the BER gradually increases with the distance, due to decay of receiving signal strength, all errors can be corrected and the PER remains 0 when the phone is within 150 cm from the monitor, which fulfills the requirement of the video data channel.

Angle. We evaluate the impact of relative angles between the phone and the monitor, as shown in Figure 13. For video, Figure 13a shows that the goodput decreases while the BER increases with a large angle. The main reason is that the projection distortion becomes severer as the camera deviates from the front of the monitor, and in this case, video frames cannot be easily decoded even with accurate screen estimation.

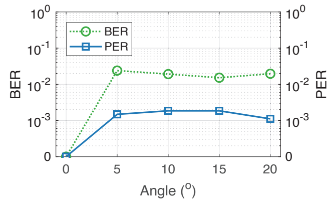
The same trend also appears in the speaker-microphone link, where the BER increases with the angle between the monitor and the phone, as shown in Figure 13b. It is mainly due to the high directivity of the monitor loudspeaker. Even though, AIRCODE still achieves ultra-low PER of about 1‰.

Signal strength. For the video part, the signal strength is represented as the quantity of lightness modification of data frames on raw video frames. As shown in Figure 14a, the BER gradually decreases as the amplitude of lightness modification increases, since data cells can be more easily recognized from the raw video content, and thus correctly decoded.

For the audio part, the signal strength is represented as the

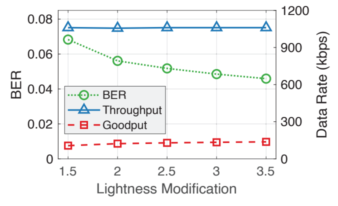


(a) Video

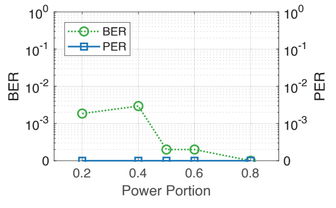


(b) Audio

Figure 13: Impact of monitor-phone angle.

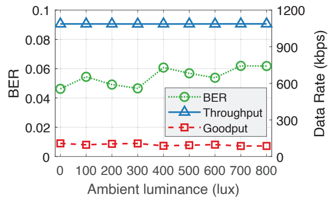


(a) Video

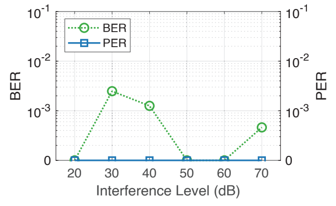


(b) Audio

Figure 14: Impact of signal strength.

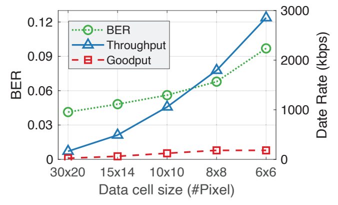


(a) Video

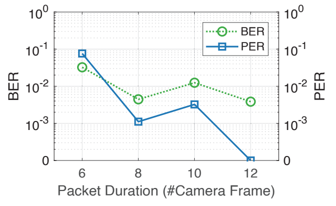


(b) Audio

Figure 15: Impact of surrounding interference.



(a) Video



(b) Audio

Figure 16: Impact of frame size.

portion of power allocated to the embedded signal. As shown in Figure 14b, the BER declines as the signal strength of the embedded signal increases, while the PER remains 0 even when only 20% power is allocated.

Background interference. We further evaluate the robustness under interferences from the environment. For the video part, the major source of interference is ambient luminance. Typical indoor ambient luminance ranging from 0 to 800 lux is evaluated. Specifically, the ambient luminance less than 600 lux is generated by an unshielded lamp suspended over the screen-camera link, while that between 600 and 800 lux is generated by natural sunlight during the daytime. A photometer is placed adjacent to the camera to measure the ambient luminance. As shown in Figure 15a, while strong ambient luminance reduces the contrast between complementary frames, the system performance only slightly degrades. It is concluded that ambient luminance has little impact on data transmission.

For the audio part, the major source of interference is environmental noises. For evaluation, we set AIRCODE to work normally at 1 m away from the monitor, and let a loudspeaker be 3 m away from the phone and play various types of audios as background interferences. The volume of the loudspeaker is adjusted to create interferences varying from 20 dB (0% volume) to 70 dB (100% volume) at the speaker. Figure 15b shows that the variance of BER is small and the PER remains 0 no matter how loud the loudspeaker plays interferences, demonstrating the robustness of the audio channel.

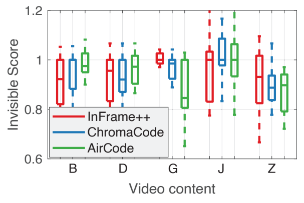
Frame size. For the video part, the frame size refers to data cell size. Figure 16a shows the data rate and BER with different data cell sizes. By gradually reducing data cell size, the throughput boosts from 163 Kbps to 2855 Kbps. AIRCODE achieves a high goodput of 182 Kbps with 8×8 data cell but saturates when data cells are further scaled down to 6×6, due to a higher chance of recognition failure and more bit errors. In practice, given the requirement of the BER and communication distance of an application, AIRCODE can trade-off the throughput by selecting appropriate data cell sizes.

For the audio part, the frame size refers to packet duration. As shown in Figure 16b, both BER and PER decrease with longer packets, as longer data symbols are less affected by reverberation and have finer frequency resolution for demodulation. By observing that the decline of BER is not obvious when the packet duration exceeds the length of 8 video frames (i.e., $\frac{1}{15}$ s), AIRCODE adopts $\frac{1}{15}$ s as the default packet duration for the audio control channel.

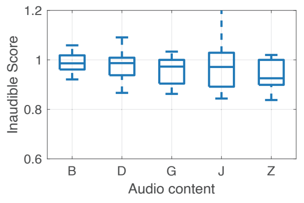
7.4 Perception Test

Video & audio content. Figure 17 shows the variation of code insensibility with different video and audio content. It is observed that all normalized scores for videos are higher than 0.6 and those for audios are higher than 0.8 for AIRCODE, indicating relatively high insensibility of the system. Among the results, some normalized scores are equal or even higher than 1, meaning that the quality of the encoded videos and audios is sufficiently high that participants cannot distinguish encoded videos and audios from original ones. For different video contents, higher scores are given to videos 'B', 'D' and 'J', which have more textured frames. It means that larger lightness modification can be applied to textured frames to ensure low BER without much impacting raw videos.

Biases of audiences. Scores given by 12 participants demonstrate different biases of their perception, as shown in Figure 18. For example, the participant 2 gives the statistically highest scores for encoded videos. From the interview after the experiment, he admits that it is difficult for him to recognize embedded data hidden in videos, meaning that the encoded videos have almost the same quality as the raw videos for him. In contrast, the participant 10 gives the lowest scores for encoded videos of all three approaches, meaning that he is relatively sensitive to unnatural alternation in videos. Among all 12 participants, only 2 (i.e., 6 and 7) of them give the highest average score to InFrame++. Yet, considering the scoring fluctuations of participants, there are no significant

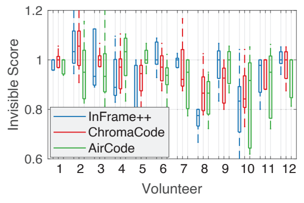


(a) Video

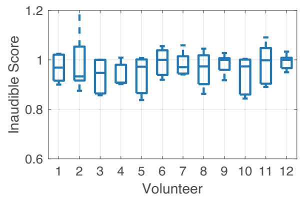


(b) Audio

Figure 17: Content diversity.



(a) Video



(b) Audio

Figure 18: Audience Diversity.

quality differences among the three approaches.

Besides 12 adults, one 5-year-old child also participates in the test. We first show him training videos with perceptible visual codes and acoustic noises. Then, the testing videos and audios are played intermittently and the child answers whether there are such codes or noises. To avoid biased hints to the child, we only give the instruction at the beginning and remain silence during playing of all testing videos. The result is that both encoded videos and audios do not discomfort the child, and he cannot observe or hear any interferences that do not belong to the origin videos and audios, demonstrating high insensibility of AIRCODE even for children.

8 Related Work

Hidden screen-camera communication. Hidden communication with the screen-camera link conveys information without interfering watching experience of audience [11, 15, 24, 30, 36, 37]. Along this direction, pioneer works, InFrame [37] and InFrame++ [36], propose and implement the basic idea, which is to switch barcodes with complementary lightness and leverage the flicker fusion property of human eyes. However, InFrame++ uses visible markers at corners of video frames for block localization, which explicitly interferes audience and consumes precious screen spaces. Extensive efforts are further made to improve the audience's experience. TextureCode [24] adaptively embeds data only in textured regions, which however is limited in throughput. PixNet [26] designs 2D OFDM symbols that can correct perspective distortions. However, they are not guaranteed to be invisible after embedded into video frames and the frequency property of OFDM symbols is destructed by changing video contents. ChromaCode [41] improves code invisibility by modifying lightness in uniform color space and achieves full imperception. However, pursuing code invisibility increases the difficulty of detecting code frames. In comparison, ensuring code invisibility, AIRCODE adopts visual odometry for accurate screen detection and achieves lower BER and higher goodput.

Speaker-microphone communication. Communication with off-the-shelf speaker-microphone links has been studied in [12, 14, 16, 22, 25, 38, 40]. Dhwani [22] adopts OFDM with PSK modulation and realizes acoustic near field communication. However, it works in the audible 6 – 7 kHz band, which interferes hearing experience of the audience and cannot be embedded in normal audio. Chirp signal is widely used in hidden acoustic communication in [12, 14, 16], for its fine correlation property. However, data rates achieved with chirp-based communication are less than 100 bps. Dolphin [38] proposes dual-mode unobtrusive communication, which still only achieves average data rates of up to 500 bps, even with the usage of a much wider frequency band from 8 kHz to 20 kHz. Noting the limitation of throughput and the advantage of the reliability of speaker-microphone communication with off-the-shelf devices, AIRCODE designs reliable and agile acoustic communication as the control channel of screen-camera link with high frame rate, boosting the overall throughput of the whole communication system. The acoustic channel will congest when multiple screens supporting AIRCODE coexist. In such a case, the screens should sense the audio channel before accessing it. Besides, audio packets should be modified to identify their belongings to the screens. We leave the support of multiple screens as future work.

Visual Odometry. Monocular visual odometry has the goal of estimating camera trajectory and reconstructing the environment with monocular commercial cameras [1–7, 13, 21, 39]. Pioneer works, e.g., MonoSLAM [2], applies filtering-based approach. Later, optimization-based methods [13] gradually substitutes the filtering-based ones for their tracking accuracy and efficiency. ORB-SLAM [21] is the representative work that uses ORB features in tracking and mapping. AIRCODE exploits the idea of visual odometry to accurately detect screens in images, enabling robust and practical screen-camera communication.

9 Conclusion

In this paper, we present the design and implementation of AIRCODE, the first hidden screen-camera communication system that achieves considerably high data rates of >1Mbps. AIRCODE is also the first system of its kind that exploits an invisible video and inaudible audio dual channel. With the high data rates being supported, AIRCODE opens up new opportunities and underpins various applications yet to be imagined for hidden screen-camera communication.

Acknowledgement

We thank the anonymous reviewers of NSDI, and our shepherd Rajalakshmi Nandakumar. This work is supported in part by the NSFC under grant 61832010, 61632008, 61672319, and Microsoft Research Asia.

References

[1] Andrew J Davison. Real-time simultaneous localisation and mapping with a single camera. In *Proceedings of IEEE ICCV*, 2003.

[2] Andrew J Davison, Ian D Reid, Nicholas D Molton, and Olivier Stasse. Monoslam: Real-time single camera slam. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, (6):1052–1067, 2007.

[3] Erqun Dong, Jingao Xu, Chenshu Wu, Yunhao Liu, and Zheng Yang. Pair-navi: Peer-to-peer indoor navigation with mobile visual slam. In *Proceedings of IEEE INFOCOM*, 2019.

[4] Ethan Eade and Tom Drummond. Scalable monocular slam. In *Proceedings of IEEE CVPR*, 2006.

[5] Jakob Engel, Vladlen Koltun, and Daniel Cremers. Direct sparse odometry. *IEEE transactions on pattern analysis and machine intelligence*, 40(3):611–625, 2018.

[6] Jakob Engel, Thomas Schöps, and Daniel Cremers. Lsd-slam: Large-scale direct monocular slam. In *Proceedings of Springer ECCV*, 2014.

[7] Christian Forster, Matia Pizzoli, and Davide Scaramuzza. Svo: Fast semi-direct monocular visual odometry. In *Proeedings of IEEE ICRA*, 2014.

[8] Ezzeldin Hamed, Hariharan Rahul, and Bahar Partov. Chorus: truly distributed distributed-mimo. In *Proceedings of ACM SIGCOMM*, 2018.

[9] Richard Hartley and Andrew Zisserman. *Multiple view geometry in computer vision*. Cambridge university press, 2003.

[10] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In *Proceedings of IEEE ICCV*, 2017.

[11] M. Izz, Z. Li, H. Liu, Y. Chen, and F. Li. Uber-in-light: Unobtrusive visible light communication leveraging complementary color channel. In *Proceedings of IEEE INFOCOM*, 2016.

[12] Soonwon Ka, Tae Hyun Kim, Jae Yeol Ha, Sun Hong Lim, Su Cheol Shin, Jun Won Choi, Chulyoung Kwak, and Sunghyun Choi. Near-ultrasound communication for tv's 2nd screen services. In *Proceedings of ACM MobiCom*, 2016.

[13] Georg Klein and David Murray. Parallel tracking and mapping for small ar workspaces. In *Proceedings of IEEE ISMAR*, 2007.

[14] Hyewon Lee, Tae Hyun Kim, Jun Won Choi, and Sunghyun Choi. Chirp signal-based aerial acoustic communication for smart devices. In *Proceedings of IEEE INFOCOM*, 2015.

[15] Tianxing Li, Chuankai An, Xinran Xiao, Andrew T. Campbell, and Xia Zhou. Real-time screen-camera communication behind any scene. In *Proceedings of ACM MobiSys*, 2015.

[16] Qiongzheng Lin, Lei Yang, and Yunhao Liu. Tagscreen: Synchronizing social televisions through hidden sound markers. In *Proceedings of IEEE INFOCOM*, 2017.

[17] Vincent Liu, Aaron Parks, Vamsi Talla, Shyamnath Gollakota, David Wetherall, and Joshua R Smith. Ambient backscatter: wireless communication out of thin air. In *Proceedings of ACM SIGCOMM*, 2013.

[18] H Christopher Longuet-Higgins. A computer algorithm for reconstructing a scene from two projections. *Nature*, 293(5828):133, 1981.

[19] JM Martínez Montiel, Javier Civera, and Andrew J Davison. Unified inverse depth parametrization for monocular slam. Robotics: Science and Systems, 2006.

[20] Carlos Morimoto and Ramalingam Chellappa. Fast electronic digital image stabilization. In *Proceedings of IEEE ICPR*, 1996.

[21] Raul Mur-Artal, Jose Maria Martinez Montiel, and Juan D Tardos. Orb-slam: a versatile and accurate monocular slam system. *IEEE Transactions on Robotics*, 31(5):1147–1163, 2015.

[22] Rajalakshmi Nandakumar, Krishna Kant Chintalapudi, Venkat Padmanabhan, and Ramarathnam Venkatesan. Dhwani: secure peer-to-peer acoustic nfc. In *Proceedings of ACM SIGCOMM*, 2013.

[23] Andre Neubauer, Jurgen Freudenberger, and Volker Kuhn. *Coding theory: algorithms, architectures and applications*. John Wiley & Sons, 2007.

[24] Viet Nguyen, Yaqin Tang, Ashwin Ashok, Marco Gruteser, Kristin Dana, Wenjun Hu, Eric Wengrowski, and Narayan Mandayam. High-rate flicker-free screen-camera communication with spatially adaptive embedding. In *Proceedings of IEEE INFOCOM*, 2016.

[25] Aditya Shekhar Nittala, Xing-Dong Yang, Scott Bateman, Ehud Sharlin, and Saul Greenberg. Phoneear: interactions for mobile devices that hear high-frequency sound-encoded data. In *Proceedings of ACM EICS*, 2015.

[26] Samuel David Perli, Nabeel Ahmed, and Dina Katabi. Pixnet: Interference-free wireless links using lcd-camera pairs. In *Proceedings of ACM MobiCom*, 2010.

[27] Nirupam Roy and Romit Roy Choudhury. Ripple ii: Faster communication through physical vibration. In *Proceedings of USENIX NSDI*, 2016.

[28] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. Orb: An efficient alternative to sift or surf. In *Proceedings of IEEE ICCV*, 2011.

[29] Daniël Schoonwinkel. *Practical measurements of Wi-Fi Direct in content sharing, social gaming android applications*. PhD thesis, Stellenbosch: Stellenbosch University, 2016.

[30] Shuyu Shi, Lin Chen, Wenjun Hu, and Marco Gruteser. Reading between lines: High-rate, non-intrusive visual codes within regular videos via implicitcode. In *Proceedings of ACM UbiComp*, 2015.

[31] Ernst Simonson and Josef Brozek. Flicker fusion frequency: background and applications. *Physiological reviews*, 32(3):349–378, 1952.

[32] Myeong-Gyu Song, Hyun-Woo Baek, No-Cheol Park, Kyoung-Su Park, Taeyong Yoon, Young-Pil Park, and Soo-Cheol Lim. Development of small sized actuator with compliant mechanism for optical image stabilization. *IEEE Transactions on Magnetics*, 46(6):2369–2372, 2010.

[33] Wei Tan, Haomin Liu, Zilong Dong, Guofeng Zhang, and Hujun Bao. Robust monocular slam in dynamic environments. In *Proceedings of IEEE ISMAR*, 2013.

[34] Tiziano Tommasini, Andrea Fusiello, Emanuele Trucco, and Vito Roberto. Making good features track better. In *Proceedings of IEEE CVPR*, 1998.

[35] Bill Triggs, Philip F McLauchlan, Richard I Hartley, and Andrew W Fitzgibbon. Bundle adjustment—a modern synthesis. In *International workshop on vision algorithms*. Springer, 1999.

[36] Anran Wang, Zhuoran Li, Chunyi Peng, Guobin Shen, Gan Fang, and Bing Zeng. Inframe++: Achieve simultaneous screen-human viewing and hidden screen-camera communication. In *Proceedings of ACM MobiSys*, 2015.

[37] Anran Wang, Chunyi Peng, Ouyang Zhang, Guobin Shen, and Bing Zeng. Inframe: Multiflexing full-frame visible communication channel for humans and devices. In *Proceedings of ACM HotNet*, 2014.

[38] Qian Wang, Kui Ren, Man Zhou, Tao Lei, Dimitrios Koutsonikolas, and Lu Su. Messages behind the sound: real-time hidden acoustic signal capture with smartphones. In *Proceedings of ACM MobiCom*, 2016.

[39] Jingao Xu, Hengjie Chen, Kun Qian, Erqun Dong, Min Sun, Chenshu Wu, Li Zhang, and Zheng Yang. ivr: Integrated vision and radio localization with zero human effort. *Proceedings of the ACM IMWUT*, 3(3):1–22, 2019.

[40] Hwan Sik Yun, Kiho Cho, and Nam Soo Kim. Acoustic data transmission based on modulated complex lapped transform. *IEEE Signal Processing Letters*, 17(1):67–70, 2010.

[41] Kai Zhang, Chenshu Wu, Chaofan Yang, Yi Zhao, Kehong Huang, Chunyi Peng, Yunhao Liu, and Zheng Yang. Chromacode: A fully imperceptible screen-camera communication system. In *Proceedings of ACM MobiCom*, 2018.

[42] Eberhard Zwicker and Hugo Fastl. *Psychoacoustics: Facts and models*, volume 22. Springer Science & Business Media, 2013.

Device-Based LTE Latency Reduction at the Application Layer

Zhaowei Tan[1], Jinghao Zhao[1], Yuanjie Li[2], Yifei Xu[3], Songwu Lu[1]
[1]*UCLA*, [2]*Tsinghua University*, [3]*Peking University*

Abstract

We design and implement LRP, a device-based, standard-compliant solution to latency reduction in mobile networks. LRP takes a data-driven approach. It works with a variety of latency-sensitive mobile applications without requiring root privilege, and ensures the latency is no worse than the legacy LTE design. Using traces from operational networks, we identify all elements in LTE uplink latency and quantify them. LRP designates small dummy messages, which precede uplink data transmissions, thus eliminating latency elements due to power-saving, scheduling, etc. It imposes proper timing control among dummy messages and data packets to handle various conflicts. The evaluation shows that, LRP reduces the median LTE uplink latency by a factor up to 7.4× (from 42ms to 5ms) for four tested apps over five mobile carriers.

1 Introduction

Low latency is critical to the proper functioning of various delay-sensitive mobile applications, such as mobile VR/AR, mobile gaming, mobile sensing, mobile machine learning, and emerging robot/drone-based image/speech recognition [22, 29, 36, 40]. These applications typically run on 4G LTE and 5G mobile networks, which offer ubiquitous access and seamless service. In this work, we study how to reduce network latency over LTE networks for such applications in the connected state. This complements the work that reduces the connection setup latency [33].

Many emergent latency-sensitive mobile apps differentiate themselves for their heavier *uplink* data transfer (e.g., user motion control, sensory data, and live camera streaming) from the device to the infrastructure. Our experiments further reveal that, uplink latency contributes to a large portion of overall latency in tested apps over operational LTE (§3.1). Reducing the uplink latency is thus as important as reducing the downlink latency. While the downlink transfer has been extensively optimized, the uplink data transfer is less studied.

Reducing the uplink LTE latency turns out to be more challenging than the downlink latency. The uplink data transfer in LTE is more complex, since it involves the interaction between the device and the network. It adopts the feedback-based device power-saving, base station-controlled scheduling for data transfer, on-demand radio resource allocation, retransmissions, etc. This results in more network latency sources with complex interactions. Traditional infrastructure-based solutions fall short to optimize them due to the lack of knowledge on device-side application usage patterns.

We design and implement LRP, a device-based, software-only LTE latency reduction solution that is readily usable for *every* commodity smartphone device. A salient feature of LRP is that it does not require root/system privilege, firmware modification or hardware change. It is thus applicable to every off-the-shelf commodity device, including Android and iOS. LRP explores the application-driven network latency reduction at the device, which complements those existing infrastructure-centric solutions that are good at downlink latency reduction. LRP focuses on reducing LTE uplink network latency, which is the bottleneck based on our empirical analysis.

The overall design of LRP takes the data-driven approach. Through analysis of operational LTE traces, we identify all elements in LTE uplink latency, and quantify them via two popular applications (§3.2). Based on the gained insights, LRP designates small dummy messages, which precede those uplink data packet transfers. It thus eliminates the latency elements due to power-saving and scheduling (§5). To make this conceptually simple idea work, LRP infers critical LTE parameters at the application layer, and performs proper timing control among dummy messages and data packet streams. To reduce the overall latency, LRP further resolves the conflict that arises among dummy messages, and avoids the conflict between data packets and dummy messages. All these solution components work at the application layer without root privilege, thus available for every off-the-shelf commodity mobile device. While LRP is mainly designed in 4G LTE, it can be readily generalized to benefit the emergent 5G (§5.5).

We implement LRP on commodity Android phones (§6). Our experiments confirm LRP's effectiveness (§7). LRP reduces the median LTE uplink latency by up to 7.4× (from

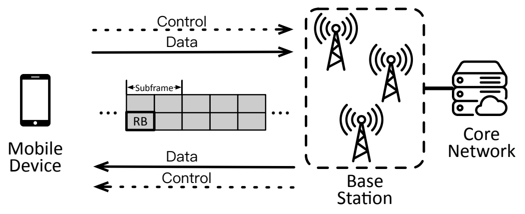


Figure 1: Network data transmission over LTE.

42ms to 5ms) for four tested applications over five mobile carriers. In any case, LRP ensures that the network latency for data transfer is no worse than the legacy LTE design. The incurred energy and data volume overhead is negligible.

2 Latency-Sensitive Mobile Apps over LTE

The mobile networks, such as 4G LTE and 5G, offer the only large-scale infrastructure that ensures universal coverage and "anytime, anywhere" access. Its infrastructure consists of radio base stations (BSes) and the core network (see Figure 1). A mobile device transfers its data with a local BS ("cell"), which covers a geographic area. The BS further relays the data to the Internet via the LTE core. A mobile device has uplink (device→BS) and downlink (BS→device) transmissions. In 4G LTE, data transfer uses scheduling-based mechanisms, where a BS schedules radio resources for each device in the cell for its uplink and downlink data transfer.

We next exemplify some representative latency-sensitive applications over the mobile networks.

Mobile VR A mobile virtual reality (VR) app typically involves 3D scenes and associated graphical engines [10, 15, 29, 31]. Standalone VR headsets such as Google Daydream [17] render 3D scenes locally. However, due to limited computation resources and high power consumption on mobile devices, high-quality VR applications typically need the edge/cloud servers to offload the rendering task [44]. In this client-server scheme, the mobile headsets or pads provide sensory/control data, while the server renders the 3D scene in the form of graphical frames. The server coordinates multiple devices, renders the VR graphical frames based on the device's input, and constructs the appropriate 3D scene for each given device.

• *Showcase VR prototype:* Following the above paradigm, we have built an example VR game with Unity 3D engine [42] on Android phones to study LTE latency. It has three modules: the controller at the device, the camera controller at the server, and the streaming component. The Android controller app acquires the device rotation data from the gyroscope sensor to control the in-game camera rotation. The GPS location is fed into the VR game so that the virtual character moves with the player's location updates. Upon receiving the player's sensory data, the camera controller at the server processes them and makes corresponding position and rotation movements for the virtual camera. We implement the streaming module with open-sourced libraries Unity Render Streaming [46] and WebRTC for Unity [18]. With the streaming module, the camera view is rendered and streamed back in 60FPS to the player with WebRTC. Players open the camera stream with the Web browser on the phone to get the real-time camera view.

Mobile sensing Smartphones today are equipped with multiple sensors: accelerometer, gyroscope, camera, to name a few. Many mobile sensing apps collect sensory data and upload them at runtime to the cloud for processing. For example, a localization app sends the GPS data to the cloud for realtime navigation. All such sensing apps are latency sensitive.

Mobile gaming In multi-player mobile games, the device acts as a controller that collects user motion, while the remote server processes the game logic. The server further provides proper synchronization and coordination among players. Moreover, pure cloud-based gaming (with rendering being processed in the cloud) is also trendy [29]. It is a new gaming paradigm being pushed by companies [40].

Cloud/edge-assisted machine learning Mobile apps with machine learning features (e.g., image/object recognition or speech understanding [14, 22]) also pose latency requirements. Network latency becomes a bottleneck for smart assistants, such as Alexa [7] and Siri [45]. Users may tolerate at most 200ms response time, while deep learning based local transcriptions take only 10ms [13].

Networking usage patterns by these mobile apps All the above representative mobile apps involve *frequent and regular uplink* data transfer. The mobile VR, sensing, and gaming [49, 50] applications collect data from device sensors and upload them to the server for subsequent actions. These sensors typically produce small data *periodically*. The user can only configure the sampling periodicity through the API provided by the mobile OS [21]. The machine learning based apps also have predictable traffic. They typically perform local computations with predictable latency before an uplink data transfer. For example, face recognition apps process a video frame locally using a fixed-sized neural network (NN). A user can gauge the delay based on the NN size. Emerging robotic or drone-based applications perform local tasks for a certain duration (e.g., scanning the surrounding environment for a few seconds [6]) before uploading the result. Such apps also exhibit uplink traffic that can be accurately predicted.

3 Demystifying LTE Latency in Mobile Apps

In this section, we empirically analyze where the application-perceived LTE latency stems from. We address two issues:

• *How large can LTE uplink latency be over operational 4G networks?* We use measurements to quantify it in §3.1.

• *Why is the uplink latency prohibitively high over LTE?* We break down this latency into multiple elements. We quantify their impact, identify root causes, and share insights in §3.2.

App	Latency	AT&T	T-Mobile	Verizon	Sprint
PUBG	UL Net	10.7	9.9	10.0	17.7
	DL Net	5.0	5.0	5.0	5.0
	UL/Total	68.2%	66.4%	66.7%	78.0%
VR	UL Net	N/A [1]	18.4	23.8	N/A
	DL Net	N/A	8.5	10.6	N/A
	UL/Total	N/A	68.4%	69.2%	N/A

Table 1: LTE latency (ms) for two mobile apps.

3.1 Measuring LTE Latency

We quantify the uplink latency over operational LTE networks via measurements and trace analysis.

Methodology We analyze the traces from our showcase VR game and another popular mobile application PUBG Mobile [40]. Our VR application uploads user motion packets (∼60Bytes) and receives 60FPS, 5Mbps downlink video stream. The downlink data packets are sent from the deployed server to the device over LTE. PUBG is a mobile game with frequent uplink data (∼40ms interval) and downlink responses. Both uplink and downlink packets are small (<100Bytes). The latency due to server processing is less than 1ms. The mobile devices (a Pixel 2 and a Pixel XL) run the apps. We collect both app logs and LTE signaling traces via MobileInsight [32]. We carry out our experiments over four US mobile carriers from 12/2019 to 09/2020. The tests cover static, low-mobility (∼1m/s), and high-mobility (∼30mph) cases, with varying signal strength (-120∼-80dBm).

Results We first measure the LTE uplink latency. We monitor the device buffer and compute the latency for each data packet. This information is available in the MobileInsight message "LTE MAC UL Buffer Status Internal". Despite small packet size, the uplink latency turns out to be non-negligible, as shown in Table 1. For all four carriers, the uplink latency (UL NET) ranges from 9.9-17.7ms for PUBG and 18.4-23.8ms for VR. These latency values might not meet the requirements of a number of latency-sensitive apps [5].

Who is the latency bottleneck? We further discover that, instead of downlink, the uplink latency poses as a major component in overall latency. We compute the downlink latency from logs of "MAC DL Transport Block" in MobileInsight. The results (DL NET) are in Table 1. We see that, uplink latency accounts for 66.4-78.0% in PUBG and 68.4-69.2% in VR. Surprisingly, even for the downlink-heavy VR app, uplink latency still contributes to a large portion of the overall latency. Recent techniques (e.g., MIMO and carrier aggregation) and 5G further reduce the DL latency with faster PHY designs. In contrast, as we will see later, the scheduling design employed for the uplink will likely be retained in 5G. As a result, we will focus on the uplink latency in this paper.

[1]VR cannot run on AT&T and Sprint, since their firewalls block the traffic.

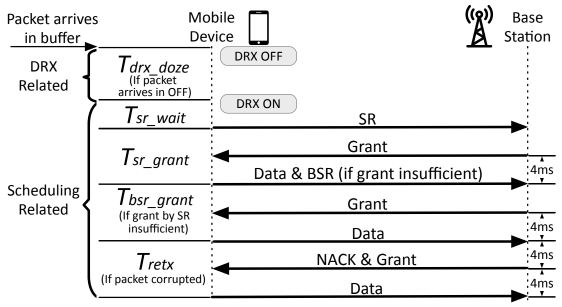


Figure 2: LTE uplink procedure & latency elements.

Latency (ms)	AT&T	T-Mobile	Verizon	Sprint
T_{drx_doze}	29.7	31.9	28.3	29.2
T_{sr_wait}	4.4	4.4	4.6	9.0
T_{sr_grant}	8.2	8.5	8.0	10.1
T_{bsr_grant}	0.03	0.00	0.03	0.16
T_{retx}	0.17	0.14	0.32	0.72

Table 2: Measured latency elements for VR application. T_{drx_doze} is the average value when present.

3.2 Why Long Latency: Breakdown Analysis

We next analyze the root causes for long network latency in 4G LTE. We identify various latency elements for the LTE uplink latency by analyzing the 3GPP standards [1, 2].

We breakdown the uplink latency as shown in Figure 2. The average number of each latency element is shown in Table 2. We can observe that, the major uplink latency bottlenecks are T_{drx_doze}, T_{sr_grant}, and T_{sr_wait}, while T_{bsr_grant} and T_{retx} are one magnitude smaller compared to other elements. We will see how each latency element acts and why it poses or does not pose as the latency bottleneck to our applications.

3.2.1 DRX Doze Latency.

Power-Saving Mode through DRX The power-saving mechanism DRX (Discontinuous Reception) in LTE may also affect latency. In a nutshell, DRX is a technique for a device to save power over LTE (see Figure 3). Instead of continuously waking up for potential downlink delivery from the BS, the device might sleep in the absence of data transfer, thus reducing its energy consumption. In DRX, a device has three states: Long DRX Cycle, Short DRX Cycle, and Continuous Reception (CRX) [2]. In CRX, the device wakes up during the ON period to monitor downlink channels. In long/short DRX, the device only wakes up for a short period of time (set by the onDuration Timer) at the start of each DRX cycle. It dozes off during the OFF period for the remaining time.

The DRX state transition is shown in Figure 3. In the Long/Short cycle state, if any downlink data is received during the ON period, the device enters the CRX state and starts the drx-InactivityTimer. Upon sending an uplink data, the device initiates an SR request. It then switches to the CRX state as well. If the device receives downlink data or initiates another SR request, the timer restarts. The short DRX state

Parameters		AT&T	T-Mobile	Verizon	Sprint
T_{sr_grant}	8ms	96.6%	96.5%	98.8%	0
	10ms	0	0.2%	0.1%	98.1%
	others	3.4%	3.3%	1.2%	1.9%
$T_{sr\_periodicity}$	10ms	94.0%	98.1%	92.3%	11.9%
	20ms	6.0%	1.9%	0	48.9%
	40ms	0	0	7.7%	39.%
$T_{inactivity\_timer}$	200ms	100.0%	99.5%	99.6%	84.5%
	others	0	0.5%	0.4%	15.5%

Table 3: Critical LTE parameters for uplink latency.

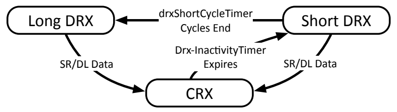


Figure 3: State transition for LTE DRX power-saving.

is entered once the drx-InactivityTimer expires. In this state, the device enters long DRX after the number of drxShortCycleTimer short cycles. All such involved timer parameters are negotiated between the device and the BS during connection setup through RRC.

How downlink DRX incurs long uplink latency DRX is designated for power saving over *downlink* transmissions. It should not block any uplink transfer. In fact, the 3GPP specification [2] stipulates that, upon the uplink sending an SR, downlink DRX should enter the CRX state as if receiving a downlink data packet. However, we found that this is not the case in practice. A new data packet refuses to invoke an SR if the device is in the doze mode. Instead, it continues to doze for a while (the time is denoted as T_{drx_doze}). It then waits for an SR slot to initiate the SR, while migrating the device to the CRX state. Table 2 shows that, T_{drx_doze} is 28.3-31.9ms on average in the four carriers. The maximum latency is 59ms with the 90th percentile being 42ms.

Note that the DRX doze latency is different from the known downlink packet delay due to waiting for DRX ON state. 3GPP [1, 2] does not mandate to prepare for SR at the DRX state. Although this latency element is not standardized, it is common for vendors as they use DRX doze to save energy. The DRX-induced doze timer is hinted in Qualcomm patents [52], where the device defers its SR during DRX OFF for energy savings. We also indirectly validate this behavior in a ZTE Z820 with Mediatek Chipset. For packets with an interval of 1 second, the measured average RTT is 35ms longer than that of packets with a small interval.

Insight: A packet keeps the device at the CRX state for $T_{inactivity\_timer}$. The idea is to reduce the DRX doze latency by sending a dummy message in advance. This way, the device is kept in the ON period before data arrival. The data packet can thus be sent without deferring until the doze period ends.

3.2.2 Scheduling Latency.

Uplink/downlink scheduling in LTE: In LTE, the uplink and downlink data transfers take different approaches:

• *Uplink data transfer over LTE* As shown in Figure 1, the uplink data transmission is through PUSCH (Physical Uplink Shared Channel). All data transmissions are regulated by the BS, which allocates resource blocks (RBs) for the actual transfer. An RB is the smallest unit allocated for a device.

In the scheduling-based LTE design, uplink data *cannot* be immediately sent out before the device is granted resource. This is done via the *request and grant* mechanism. Specifically, the device sends an SR (Scheduling Request) through PUCCH (Physical Uplink Control Channel). SR is a signaling message notifying new data arrival at the mobile device. Moreover, an SR signal cannot be sent at any time instantly. It can only be sent during certain subframes (called SR occasions). The periodicity of SR occasions is notified by the BS during connection setup. Upon receiving an SR, the BS returns an uplink Grant (i.e., *grant*) to the device. A grant specifies what RBs and modulation the device could use *4ms later*. The number of RBs in response to an SR depends on the BS configuration, since SR is just a message stating "device has data to send" without specifying the amount.

• *Downlink data transfer over LTE* LTE still uses the scheduling-based operations for its downlink. However, BS directly allocates RBs for each device upon data arrival, since BS knows what data to transmit to which device.

How scheduling incurs long latency The device also suffers from its uplink scheduling latency. It must wait for an SR occasion before receiving a grant from the BS to upload its data packet. The latency element, denoted as T_{sr_wait}, is thus affected by the periodicity of an SR occasion $T_{sr\_periodicity}$. The device then waits for a grant, which the device could use 4ms later. The latency from sending the SR to sending the data packet is denoted as T_{sr_grant}. The two elements of scheduling are shown in Figure 2. We measure them in Table 2. The SR waiting latency T_{sr_wait} is 4.4ms for AT&T, 4.4ms for T-Mobile, 4.6ms for Verizon, and 9.0ms for Sprint. Sprint has the largest T_{sr_wait} because it has the longest SR cycle. T_{sr_grant} is 8.2ms, 8.5ms, 8.0ms, and 10.1ms for the four carriers. The accumulative latency is denoted as $T_{scheduling} = T_{sr\_wait} + T_{sr\_grant}$.

Insight: This scheduling latency can be reduced. If a grant is pending at the device, a new arriving data packet can use it for transfer. Therefore, we may use a dummy message to request for a grant in advance, so that the data packet can use this grant for actual transfer without delay.

3.2.3 Other Latency Elements.

Buffer status report (BSR) SR is an indicator that informs the BS of new pending data, without specifying *how much*. When the packet that triggers SR is large, the initial grant might be insufficient. The device then sends a BSR (Buffer Status Report) together with the data packets in the scheduled RBs. Unlike SR, a BSR includes the info on how much data still remains in the device buffer. Upon receiving

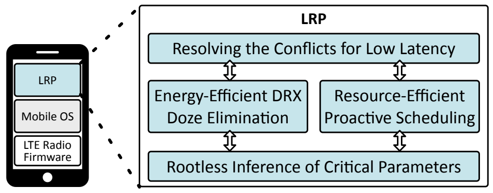

LRP
Mobile OS
LTE Radio Firmware
LRP
Resolving the Conflicts for Low Latency
Energy-Efficient DRX Doze Elimination
Resource-Efficient Proactive Scheduling
Rootless Inference of Critical Parameters

Figure 4: An overview of LRP.

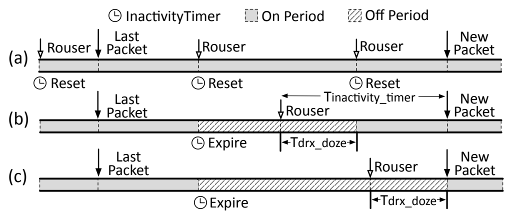


Figure 5: Component solution to DRX doze latency.

the BSR, the BS will process it and respond with sufficient grants for the buffered uplink data.

We note BSR's impact on uplink latency is negligible for most applications in §2. The latency between a BSR and the time to use the grant (denoted as T_{bsr_grant}) is illustrated in Figure 2. Conceptually, it is the request processing time + 4ms, similarly to T_{sr_grant} ($\approx 10ms$). However, it equals to 0 when the initial grant is sufficient. The measurement results are in Table 2. The BSR latency is less than 1ms on average for four US carriers. This is because a base station usually provides a large grant (>100B) sufficient for our apps in response to SR,

Retransmission in LTE An uplink data packet might be corrupted during transfer. Upon receiving a corrupted packet, the BS notifies the device by sending a NACK and a grant. The device uses the grant to retransmit the corrupted data. Similar to BSR latency, the retransmission has limited impact on the uplink latency for apps in §2. The ReTx latency for uplink data packet is fixed at 8ms if needed [1] and 0 otherwise. We denote this latency as T_{retx} and the procedure is shown in Figure 2. Among all data packets, 2.1% in AT&T, 1.7% in T-Mobile, 4.0% in Verizon, and 9.1% in Sprint perceive ReTx latency. Less than 1ms latency is incurred on average, shown in Table 2. Unlike downlink with up to 10% retransmissions [48], uplink packets are small and less prone to corruption.

4 LRP Overview

We devise LRP, an in-device software solution to latency reduction for mobile apps. Figure 4 shows LRP's components. It runs as a user-space daemon at the device, **without** requiring system/root privilege, firmware modification, or hardware support. It is applicable to both Android and iOS. LRP masks the LTE latency elements in §3.2 for applications by *proactively* requesting the needed radio resources and high-speed transfer mode, while still retaining low energy and data consumption overhead. As an application-layer solution, LRP cannot directly control the low-level LTE mechanisms (that require root privilege or firmware access). Instead, it indirectly regulates the LTE uplink transfer with well-crafted *dummy packets*. LRP complements solutions designed to reduce other non-network latency elements [19, 51, 54]. While conceptually simple, LRP must address three challenges:

• **Accurate timing control for each latency element (§5.1–5.2):** Initializing the dummy packets at the right time is crucial to both reducing latency and minimizing energy consumption, signaling overhead, and radio resource usage (thus billing). The proper timing depends on the traffic pattern *and* the unique characteristics of each latency element. To this end, LRP customizes the timing control for critical latency elements, including the DRX doze and scheduling (§3.2).

• **Conflict handling for overall latency reduction (§5.3):** Simply reducing each latency element does not suffice to reduce the overall latency. Due to the complex interactions between LTE latency elements, reducing one latency element may increase other latency elements. Moreover, the dummy packets may compete radio resources with the legitimate data, incurring additional data latency. To this end, LRP devises resolution and avoidance schemes for both types of conflicts.

• **Rootless inference of critical LTE parameters (§5.4):** To be readily usable by *every* device, we design LRP as a user-space software daemon *without* requiring system/root privilege. The challenge is that, LRP's latency reduction requires the fine-grained knowledge of low-level LTE parameters inside the hardware modem chipset. Existing solutions to directly access them (e.g., MobileInsight [32] and QXDM [41]) require root privilege or external hardware support. According to [28], only 7.5% of global mobile devices are rooted. We propose a novel approach to infer these parameters without any system privilege or firmware/hardware modification.

5 The LRP Design

We next elaborate on LRP's design. We first propose component designs to reduce each latency element (§5.1–§5.2), and resolve potential conflicts among them (§5.3). To realize the components without root privilege, we propose a novel inference method at the application layer (§5.4). We analyze LRP and extend the discussion to irregular traffic and 5G (§5.5).

5.1 Energy-Efficient DRX Doze Elimination

To reduce the DRX doze latency in §3.2.1, LRP should ensure the device is in ON period when a data packet arrives at the device buffer. As an application layer solution, LRP cannot directly switch the device to the CRX state (that needs firmware modification). Instead, it sends a dummy packet (*rouser*) *before* the data packet's arrival.

Despite being straightforward at the first glance, a *rouser* is only effective if being sent at the right time. An imprudent *rouser* can either incur unacceptable energy waste or cannot help reduce latency. Therefore, timing control is crucial to balancing latency and energy cost. We first discuss some naive solutions with limitations, and then present our design.

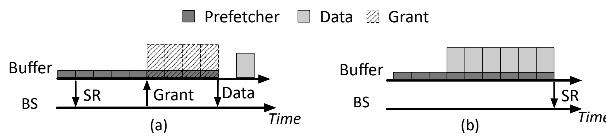


Figure 6: Impact of *prefetcher* Timing.

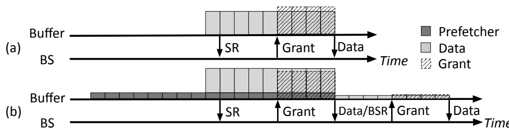


Figure 7: Corner case: a *prefetcher* increases latency.

Naive timing control One naive solution is to keep DRX at CRX state at all times by frequently sending *rouser*s. As shown in Figure 5(a), this can be achieved by sending a *rouser* every $T_{inactivity\_timer}$. Unfortunately, this results in unacceptable energy waste, as the device never enters the doze mode.

A better choice is to send a *rouser* with the time in advance, denoted as t_r, being set to $t_r = T_{inactivity\_timer}$ (Figure 5(b)). On one hand, as the packet keeps the ON period for $T_{inactivity\_timer}$ after dozing, $t_r = T_{inactivity\_timer}$ ensures that the data packet enters the buffer during the ON period. On the other hand, this saves power compared to the first naive choice, since the extra ON period is capped at $T_{inactivity\_timer}$ for each packet at most. However, extra energy consumption is still incurred. Since $T_{inactivity\_timer}$ (∼200ms) is typically much larger than T_{drx_doze} (∼30ms) in reality, the ON period between wakeup from the doze mode and the data packet is unnecessary.

`LRP`'s approach `LRP` prioritizes latency over marginal energy waste with proper timing control. Instead of frequent *rouser*s in naive solutions, `LRP` only sends a *rouser* for the time T_{drx_doze} in advance. We thus keep updating the maximum T_{drx_doze}, denoted as $T_{drx\_doze\_max}$. The timing to send the *rouser* is $t_r = T_{drx\_doze\_max}$. If the device enters the ON period during doze, i.e. $t_r > T_{drx\_doze}$, the *rouser* finishes dozing before the data packet arrives, thus eliminating the doze latency for the data packet. It is also likely that T_{drx_doze} for a *rouser* exceeds t_r. In this case, the packet enters the buffer and endures the dozing latency together with the *rouser*. Although the doze latency is not eliminated, the *rouser* reduces it by t_r.

5.2 Resource-Efficient Proactive Scheduling

`LRP` next seeks to mask the round trips of the scheduling in §3.2.2 for the mobile app. The idea is to send a scheduling request (SR) *before* the arrival of the data, so that the data does not need to wait for the radio grants. As an application-layer solution, `LRP` cannot directly trigger the SR early (which requires modifying the firmware). Instead, it requests a grant from the BS in advance by sending a dummy message, named *prefetcher*. This is feasible since the grant is not tied with the packet that requests it. Moreover, since the BS responds to each SR regardless of the pending data size, a small dummy message can receive a grant that allows for much-larger-size transmission than itself, thus sufficing to accommodate the followup data packet transfer in a single transmission.

Similar to the DRX doze elimination in §5.1, an effective *prefetcher* also needs accurate timing control. As shown in Figure 6, imprudent timing can offset the latency reduction, and/or waste radio resources. We next discuss both naive solutions in Figure 6, and then show `LRP`'s approach.

Naive timing control A too early *prefetcher* might result in both resource waste and prolonged latency as shown in Figure 6(a). The *prefetcher* is sent too early so that the timing to use the returned grant is already passed when the data packet arrives. The resource is thus wasted, while the data packet misses the opportunity to reduce its scheduling latency.

Similarly, a late *prefetcher* could also miss the opportunity to reduce the scheduling latency for the data packet, as shown in Figure 6(b). If the *prefetcher* is sent too late after a potential SR that could reduce latency, the data packet might have to wait for scheduling latency as if no *prefetcher* is issued. In the worst case for both early and late *prefetcher*, it may result in missed latency savings up to $T_{sr\_periodicity} + T_{sr\_grant}$.

`LRP`'s approach `LRP` aims at reducing the scheduling latency at marginal radio resource cost. Let a *prefetcher* be sent t_p before the data packet. The parameter t_p must meet two requirements. First, we should ensure $t_p \geq T_{sr\_grant}$. Note that, an SR can only request a grant to be used at T_{sr_grant} after the SR. Therefore, $t_p \geq T_{sr\_grant}$ guarantees that the SR is sent only if it helps to reduce the scheduling latency for the data packet. Second, we must ensure $t_p \leq T_{sr\_grant}$. This is to let the requested grant be used to transmit the data packet. No resource waste or premature SR is incurred.

Consequently, our timing design is to set the time advance as $t_p = T_{sr\_grant}$, which meets both requirements. Note that T_{sr_grant} is typically constant for a BS, being the accumulative latency of SR processing latency + 4ms, where 4ms is a standardized parameter in [1]. In our experiment, more than 96.5% of T_{sr_grant} is identical under a BS regardless of the carrier. If T_{sr_grant} changes after handover to a new BS, we update T_{sr_grant} immediately. Even if T_{sr_grant} may vary, our solution is no worse than the current practice.

Impact of the data packet size A *prefetcher* helps reduce scheduling latency if the data packet size ≤ grant - *prefetcher* size, which is common in reality as >99% of initial grants in our experiments exceed 100B in all operators, while the uplink sensory data is smaller than half of that. Therefore, a *prefetcher* initiates an SR, and gets a returned grant that suffices for the data packet to be sent with the *prefetcher*.

However, a corner case arises when the grant in response to SR is enough for the data packet, but not for a *prefetcher* + the data packet. As shown in Figure 7(b), the device could only send the *prefetcher* and a portion of the data packet. A BSR further requests a grant for the remaining data. The data packet thus suffers extra BSR latency compared to the case without *prefetcher* (Figure 7(a)). In the worst-case scenario, this latency increases by T_{sr_grant} (∼8ms). We discuss the

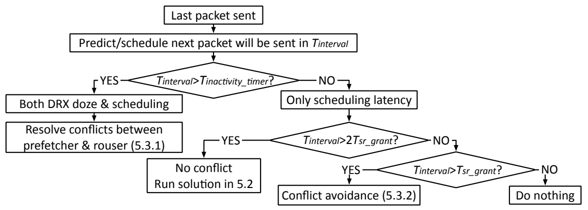

Last packet sent
Predict/schedule next packet will be sent in $T_{interval}$
YES
$T_{interval}>T_{inactivity\_timer}$?
NO
Both DRX doze & scheduling
Only scheduling latency
Resolve conflicts between prefetcher & rouser (5.3.1)
YES
$T_{interval}>2T_{sr\_grant}$?
NO
No conflict Run solution in 5.2
YES
$T_{interval}>T_{sr\_grant}$?
NO
Conflict avoidance (5.3.2)
Do nothing

Figure 8: The workflow of conflict handling in LRP.

probability of this case in Appendix B. However, even in this corner case, the worst case happens only when the data and the *prefetcher* arrive in the same SR period, with probability T_{sr_grant}/T_{sr_wait}. For other conditions in the corner case, the latency is the same as vanilla LTE.

5.3 Handling the Conflicts for Low Latency

LRP further resolves several conflicts for overall latency reduction. Figure 8 illustrates the workflow of LRP. Let $T_{interval}$ be the time interval between the last and the next expected packet. LRP thus reduces various latency elements. It handles improper interplay between latency elements, and between dummy and data packets.

5.3.1 Conflict Resolution Between Latency Elements

LRP issues two types of dummy packets for latency reduction: *rouser*s for DRX-induced doze latency, and *prefetcher*s for scheduling latency. Figure 9(a) illustrates their conflicts. A *rouser* itself is a dummy message that needs to be sent before a *prefetcher*. Once turning the device to DRX ON, it asks for the grant, which could carry both *rouser* and *prefetcher*. Therefore, the *prefetcher* is sent by the grant requested by the *rouser*. The grant-induced scheduling latency is not reduced at all. The latency penalty can be as large as $T_{sr\_periodicity} + T_{sr\_grant}$ compared to no-conflict case in Figure 9(b).

To resolve this conflict, we refine the timing control to ensure both dummy packets' effectiveness. Specifically, we should make sure a *rouser* is sent when a *prefetcher* hits the device buffer, so that the *prefetcher* can take effect and reduce the scheduling latency. A *rouser* takes at most $T_{sr\_periodicity} + T_{sr\_grant}$ to be sent out as a dummy message and a *prefetcher* needs to be sent T_{sr_grant} before the data packet. Therefore, we adapt the timer from $t_p = T_{drx\_doze\_max}$ to $T_{drx\_doze\_max} + T_{sr\_periodicity} + 2T_{sr\_grant}$ to ensure a *rouser* is sent before a *prefetcher*. The *rouser* thus endures $T_{drx\_doze\_max}$ that guarantees the doze is completed and then sent out.

5.3.2 Conflict Avoidance Between Dummy and Data

The next conflict arises between LRP's dummy packets and the last legitimate data packet. If a *rouser* conflicts with the last packet, this does not pose an issue: the *rouser* can still help the device to remain in the ON period for $T_{inactivity\_timer}$. We thus only discuss where a *prefetcher* intervenes with the last packet. We show how LRP adapts this for latency reduction.

There are two instances when a *prefetcher* arrives in the buffer *before* the last data packet being completely sent out, shown in Figure 10. In case (a), the *prefetcher* does not provide any latency reduction. The grant for the last packet has enough room to carry the *prefetcher*, which will be sent together. There is no *prefetcher*-requesting grant for the next data packet. In case (b), a *prefetcher* may increase the latency. The grant for the last data packet cannot accommodate the piggy-backed transmission of the *prefetcher*. A BSR request is thus triggered by the device to request for more grants. Since BSR specifies the size for the dummy message *prefetcher*, the returned grant does not suffice to transmit the data packet. This subsequently invokes another round of BSR-grant operations. The data packet might suffer from extra BSR latency.

To avoid the conflicts, LRP adjusts the timing of a *prefetcher*. It leaves enough time for the last packet to complete its transmission before the *prefetcher*. Recall that the theoretical maximum uplink latency that the last packet would experience after optimization is T_{sr_grant}. The dummy *prefetcher* is then sent at least T_{sr_grant} after the application sent its last packet. Specifically, if the time gap (between the last packet arrival and the next packet arrival) is larger than $2T_{sr\_grant}$, we send a *prefetcher* T_{sr_grant} before the next packet. This is the timing we designed in §5.2; it will not break the above condition. Otherwise, we send a *prefetcher* T_{sr_grant} after the last packet. This choice will reduce less latency compared to the timing in §5.2 without conflicts. However, we avoid the cases of Figure 10, where a conflict negatively affects the latency.

5.4 Rootless Inference of Critical Parameters

As shown in §5.1–5.3, LRP relies on knowing certain LTE parameters for latency reduction. Obtaining such parameters through the root privilege can definitely work. However, such an approach limits the applicability of LRP. To let LRP work with *every* commodity device, we seek to infer these parameters at the application layer. Note that existing tools typically require system privilege (e.g., MobileInsight [32]) or additional hardware (e.g., QXDM [41]).

To infer these critical LTE timers, LRP exploits packet pairs for probing. Figure 11 shows the general procedure. LRP sends two adjacent probing requests and records their interval t_1. Upon receiving the responses to both packets, LRP compares the responses' intervals t_2 with t_1, and estimates the corresponding timers. This approach is based on the premise that, the difference between t_1 and t_2 mainly arises from the different uplink LTE latency experienced by two packets. This premise largely holds in practice, because latency fluctuations from the base station are much larger than those in the core network or servers[2]. Compared with the conventional packet-pair technique, LRP customizes probing packets with the LTE

[2]We have validated this premise in operational LTE. We send a pair of DNS requests at $t_1 = 0$. A UL grant suffices to send both requests; they arrive at the BS simultaneously. t_2 is solely affected by the core network and DL. The results show that $t_2 < 1ms$ for >99% responses.

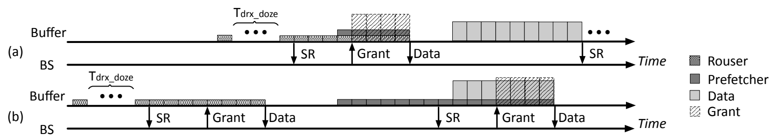


Figure 9: Improper timing control causes conflicts between components.

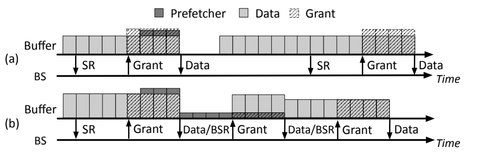


Figure 10: Conflicts: dummy msgs & data.

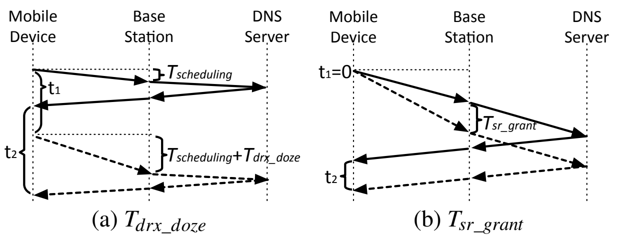


Figure 11: Inferring parameters at application layer.

domain knowledge for accurate inference.

Inferring DRX-related parameters To reduce DRX doze latency, LRP should know $T_{drx\_doze\_max}$ (§5.1). Recall that the DRX doze latency is only present when the packet interval is large. The idea is to let t_1 be large enough so that the first response packet cannot keep the second request in DRX ON. The second packet in the pair experiences UL DRX doze latency, while the first does not as we immediately start next pair after one is done. We can thus use the interval difference $t_2 - t_1$ to infer DRX doze latency. Consider that two requests can also be different in terms of scheduling latency, we repeat the pair for 10 times and take the interval difference average. Figure 11(a) illustrates this procedure.

One caveat is that we need to know how large t_1 is so that the second request suffers from DRX doze latency. We increase the interval t_1 gradually until a certain spike appears in measured RTT for the second request, caused by DRX doze latency ($\approx$30ms as shown in §3). The time interval between the first response and the second request that triggers such spike infers $T_{inactivity\_timer}$. We can thus infer $T_{drx\_doze} = t_2 - t_1$. We take the max in multiple rounds as $T_{drx\_doze\_max}$.

Inferring scheduling-related parameters LRP needs T_{sr_grant} to reduce the scheduling latency (§5.2). Figure 11(b) shows how LRP infers it. We let t_1 as 0 by sending both requests together. As we just showed, the grant is sufficient for a single request packet. We increase the size of the second request so that the grant will not be sufficient for both request packets. According to the scheme, the first request experiences only scheduling latency while the second experiences the same scheduling latency plus T_{bsr_grant}, which equals to T_{sr_grant} under a same BS. We thus can derive LRP optimization parameter T_{sr_grant} from the measured t_2 as $T_{sr\_grant} = t_2$.

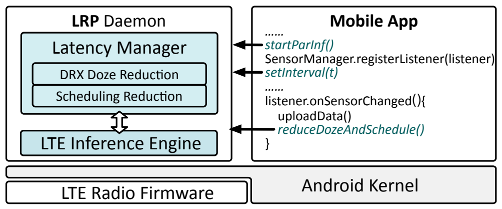
5.5 Miscellaneous Issues

Energy Analysis LRP incurs extra energy overhead from four sources. First, transmitting a *rouser* incurs a longer ON period. The time to send a *rouser* can be as long as $T_{sr\_periodicity} + T_{sr\_grant}$. It incurs $T_{sr\_periodicity}/2 + T_{sr\_grant}$ on average. Second, T_{drx_doze} is not predictable so we select $T_{drx\_doze\_max}$ to prioritize latency over energy. The extra ON period is $\delta = T_{drx\_doze\_max} - T_{drx\_doze}$ for each *rouser*. If the packet arrives during DRX OFF, $T_{drx\_doze\_max}$ equals to T_{drx_doze} and $\delta = 0$. Otherwise, $T_{drx\_doze} = 0$ and $\delta = T_{drx\_doze\_max}$. The expectation of δ is thus $p_{on} \cdot T_{drx\_doze\_max}$, where p_{on} is the probability of a packet arriving during DRX ON period. If we ignore background traffic and assume the packet arrives in the buffer at a random time, p_{on} = onDurationTimer / DRX cycle. Third, an early *rouser* (due to inaccurate estimation) also causes a longer ON period. Denote ε as the estimation error. When the *rouser* arrives during DRX OFF, the extra ON period is ε. Otherwise, ε incurs no extra ON period. Finally, sending extra small messages incurs extra energy waste.

Impact on the spectrum efficiency For every data packet, we define its spectrum efficiency $SE = \frac{\text{sizeof (data packet)}}{\text{sizeof (Total UL resource granted)}}$. When a data packet suffers from doze latency and LRP sends a *rouser*, it reduces *SE* by half: the *rouser* and the *prefetcher* initiate two grants, while the legacy LTE only requests for one. The extra grant occupies $\approx$2 RB in commercial networks. LRP trades-off *SE* for low latency. When LRP sends a *prefetcher* only, two scenarios arise. In the normal case, the grant from SR can carry both the data packet and a *prefetcher*. Therefore, LRP requests no extra grant and *SE* is the same as the legacy LTE. In the corner case discussed in §5.2, the BS allocates at most sizeof (*prefetcher*) extra grant. One extra RB is thus wasted, since a single RB is sufficient to carry a *prefetcher*. *SE* is reduced by $\frac{\text{sizeof}(\textit{prefetcher})}{\text{sizeof}(\textit{prefetcher} + \text{grant from SR})}$. This value multiplying the probability of the corner case (see Appendix B) yields the expectation of *SE* reduction.

Impact of background traffic LRP still reduces latency in the presence of background traffic. No matter whether the background packet is sent before a *rouser* or between a *rouser* and a data packet, the *rouser* will keep the data packet at the DRX ON state, thus eliminating the DRX doze latency. On scheduling latency, if the background traffic is sent after the data packet, it does not affect the *prefetcher*. If the background traffic is in between, the *prefetcher* reduces its latency, which indirectly reduces latency for the real data packet. It still does not increase the latency compared to legacy LTE without LRP. When the background traffic is sent before a *prefetcher*, it will be sent out through BSR before the data packet in the worst case, equivalent to no optimization.

What if the uplink traffic is not strictly regular? While mobile sensors produce regular data packets, the actual uplink



```
......
startParInf()
SensorManager.registerListener(listener)
setInterval(t)
......
listener.onSensorChanged(){
    uploadData()
    reduceDozeAndSchedule()
}
```

Figure 12: Implementation of **LRP** in Android.

data packets might not be strictly periodic. This can be caused by mobile OS overhead, prediction inaccuracy, or sensor periodicity variance. LRP still guarantees no worse latency than legacy LTE, and saves LTE latency in most scenarios. We show the following Theorem 5.1 and prove it in Appendix C.

Theorem 5.1. For the data packet that should have arrived at $T_{interval}$ but actually arrives at T, LRP does not incur extra latency compared with the legacy 4G LTE.

LRP for non-regular traffic For those ML/AI apps of §2 with irregular but predictable uplink traffic, LRP works equally well. For others with irregular yet unpredictable uplink traffic, we do not recommend LRP for such apps. If users intend to use our APIs, latency reduction cannot be ensured.

Applicability to 5G In principle, LRP is applicable to 5G, which has three usage cases. Enhanced Mobile Broadband (eMBB) extends the current 4G technology. Massive Machine Type Communication (mMTC) is for cellular IoT devices. Its design is based on LTE-M and NB-IoT [8]. The scheduling mechanisms of both modes largely remain unchanged [3, 4]. LRP is still applicable. Ultra Reliable Low Latency Communications (URLLC) targets low-latency communication. The potential grant-free scheduling might partially achieve LRP's latency reduction, but LRP's DRX doze latency reduction will still help the URLLC applications.

Network impact LRP incurs little overhead on the network side. The overhead stems from processing extra signaling, which is marginal compared with normal operations. This is because the BS monitors the control and data channels continuously, regardless of whether it receives an SR.

Impact on other users If the devices under a BS all use LRP, they will still benefit from LRP. The core idea of LRP is to schedule a device's allocated resources in advance if its data arrival can be predicted. The procedure does not sacrifice other users' access in general. Moreover, if certain device does not adopt LRP, its latency may be slightly prolonged. This arises when the BS assigns the last available resource to an LRP user who advances its scheduling, while this resource could have been available to the non-LRP user. However, the impact is minimal, as the BS will serve the user the next subframe (in 1ms) and the throughput is not affected.

6 Implementation

We implement LRP as a standalone user-space daemon with Android NDK. A similar implementation is also feasible for iOS. Figure 12 shows its key components, including a latency manager for latency reduction with conflict resolution in §5.1–5.3, an inference engine that offers key parameters for LRP based on the solutions in §5.4, and a set of APIs for latency-sensitive applications. To use LRP, a latency-sensitive mobile app requests LRP service using its APIs as detailed below. At runtime, LRP first detects if the device connects to a new base station by checking the change of serving cell ID. Upon cell changes, LRP starts to infer the key LTE parameters for this new cell. Once the key parameters are obtained, LRP initiates its latency manager to reduce DRX doze latency in §5.1 and scheduling latency in §5.2, and resolves the conflicts in §5.3.

APIs LRP provides easy-to-use application-layer APIs for mobile application developers. Figure 12 showcases these APIs with a mobile VR application. The app first calls startParInf() so that LRP daemon starts and infers the LTE parameters relevant to latency reduction components. The daemon detects possible parameter changes (say, upon handover) and re-runs the inference procedure whenever necessary. As our VR application uploads periodic sensory data packets, it calls setInterval(t) to inform LRP such periodicity. Whenever a data packet is sent, the application calls reduceDozeAndSchedule() for LRP to reduce latency for the next packet.

Latency manager It realizes the latency reduction in §5.1–5.2 and conflict resolution in §5.3. A practical issue to realize them is to optimize the dummy packet's construction and delivery for low cost. Both *prefetcher* and *rouser* messages in LRP should be as small as possible so that extra data overhead is minimized. In addition, a smaller *prefetcher* will decrease the likelihood of the corner case discussed in §5.2. The smallest packet we could generate in the Android device without root is an ICMP ping packet with IP header only via system command. Our implementation issues only one small ICMP packet to the local gateway in LTE that serves the users.

LTE inference engine It infers the key LTE parameters for LRP's latency reduction based on the approaches in §5.4. We use DNS requests/responses as probing packets, which have low deployment cost (by using LTE's readily-available DNS servers) and higher accuracy (compared to other probing packets delivered with low priority such as ICMP). For DNS servers, LTE assigns its own in-network DNS server when the device attaches to it, which provides fast and stable service. We use such DNS servers for our experiment.

Moreover, we note that simply running the inference in §5.4 may be inaccurate in practice, since it is sensitive to the noises from background traffic, vendor-specific base station behaviors, and server load. To this end, we optimize our implementation to mitigate these noises and improve the inference accuracy. Specifically, we add a few filters to get rid of the noises. For instance, when measuring the scheduling-related parameters, we know that T_{rtt} should be greater than 4ms in reality, therefore, if the packet response pair is received within 4ms, we ignore this round of experiment.

App		AT&T			Verizon			T-Mobile			Sprint			China Mobile		
		Leg	LRP	η	Leg.	LRP	η	Leg.	LRP	η	Leg.	LRP	η	Leg.	LRP	η
Mobile VR	Med.	N/A	N/A	N/A	12.0	8.0	0.5×	11.0	6.0	0.8×	N/A	N/A	N/A	N/A	N/A	N/A
	95%	N/A	N/A	N/A	28.0	15.0	0.9×	40.0	14.0	1.9×	N/A	N/A	N/A	N/A	N/A	N/A
Gaming	Med.	10.0	6.0	0.7×	9.0	7.0	0.3×	9.0	7.0	0.3×	17.0	11.0	0.5×	4.0	3.0	0.3×
	95%	17.0	15.0	0.1×	15.0	15.0	0×	15.0	15.0	0×	27.0	21.0	0.3×	10.0	5.0	1.0×
Localization	Med.	38.0	5.0	6.6×	50.0	14.0	2.6×	42.0	5.0	7.4×	30.5	14.0	1.2×	11.0	3.5	2.1×
	95%	46.0	14.0	2.3×	59.0	23.0	1.6×	48.0	10.0	3.8×	61.7	25.8	1.4×	22.0	6.0	2.7×
Object Detection	Med.	23.0	7.0	2.3×	38.0	9.0	3.2×	33.0	5.0	5.6×	30.0	15.0	1.0×	14.0	6.0	1.3×
	95%	47.8	16.0	2.0×	51.0	15.3	2.3×	45.0	10.0	3.5×	59.0	27.5	1.1×	22.0	17.0	0.3×

NOTE: Mobile VR is evaluated under Verizon and T-Mobile only. Other operators' firewalls block the VR traffic. Leg: Legacy LTE. Med: Median.

Table 4: Uplink network latency (ms) reduction by `LRP` in evaluations with four apps. η=(Legacy-`LRP`)/`LRP`.

7 Evaluation

We assess how LRP improves the overall latencies and QoEs for emergent mobile applications, evaluate the effectiveness of solution components in LRP, and quantify LRP's overhead.

Experimental setup We run LRP on Google Pixel, Pixel 2, Pixel XL, and Pixel 5. We quantify the latency reduction in both US and China over AT&T, Verizon, T-Mobile, Sprint, and China Mobile. The evaluation covers 375 unique cells. We repeat the tests in static, walking (∼1m/s), and driving (∼30mph) scenarios. We do experiments mostly in metropolitan areas while driving tests cover rural areas as well. The radio signal strength varies from -120 to -80dBm, covering good (>-90dBm), fair ([-105, -90dBm]), and bad (<-105dBm) conditions. To quantify LRP's latency reduction, we use MobileInsight [32] to extract the ground truth of fine-grained per-packet latency breakdown from the chipset.

To gauge LRP's impact on the network side, we build a USRP-based testbed. A server with Intel i7-9700k CPU and 32G RAM runs srsLTE [20] for the functions of core network and BS processing. A USRP B210 connects to the server and provides wireless access for the devices. We plug sysmoUSIM [47] into the test phones, and register them.

7.1 Overall Benefits for the Applications

We showcase LRP's latency reduction and QoE improvements with four representative emergent mobile applications:

◦ *Mobile VR.* We use the showcase VR game as described in §2. We measure the latency of the sensor data and control data, and use it to gauge how our design reacts to VR games.

◦ *Localization.* We write an Android app that uploads the periodic GPS location status to the cloud via the Android API [21]. We encode each location update in 22 bytes and send it to the cloud every second.

◦ *Object recognition.* We prototype an object recognition app using MobileNetV2 [43], a phone-based deep learning model. The app processes camera frames and uploads the recognition result to the cloud. The typical inference time is 250ms.

◦ *Gaming.* We evaluate its latency by replaying the traces from PUBG Mobile [40], one of the most popular multi-player online mobile games. Since PUBG traffic is not strictly regular, we use it to demonstrate the effectiveness of LRP as discussed in §5.5. We use the traffic emulator to send data packets based on the trace.

Overall LTE latency reduction Table 4 and Figure 13 show LRP's latency reduction for these apps in static settings with fair-good signal strength; other scenarios have similar results as detailed in §7.2. On average, LRP achieves 4-5ms (0.5-0.8×) latency reduction in mobile VR, 8-37ms (1.2-7.4×) reduction in localization, 8-29ms (1.0-5.6×) reduction in object detection, and 1-6ms (0.3-0.7×) reduction in gaming for all 5 LTE carriers. Our breakdown analysis further shows these apps suffer from different latency bottlenecks. For the localization and object detection, the majority of data packets suffer from both DRX doze and scheduling latency. For the VR and gaming with more frequent packets, the scheduling latency is the major latency bottleneck. LRP can reduce both bottleneck latencies and thus benefit all these applications.

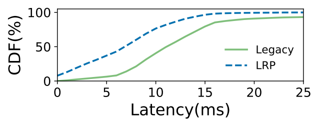
QoE improvement To showcase the impact of LRP on the mobile VR, we conduct a user study with 10 participants to evaluate the subjective experiences of using VR with/without LRP. Figure 14 shows the average Mean Opinion Score (MOS) on three aspects: graphical visual quality, responsiveness, and overall experience. Participants rate 1 (Bad) to 5 (Excellent) on these three aspects of the VR game with constant head position changes. The results show that LRP can improve the visual quality by 8% (3.1→4.0), responsiveness by 63% (2.4→3.9), and overall experience by 46% (2.4→3.5).

5G latency reduction We evaluate how LRP reduces 5G latency under AT&T 5G network. Since we do not have access to its fine-grained data-plane traces, we measure RTT at the application layer. LRP reduces RTT by 4.6ms for Gaming, 20.5ms for Localization, and 19.8ms for Object Detection. The results are similar to the latency reduction in AT&T 4G.

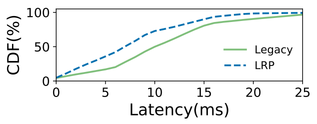
7.2 Micro-Benchmarks

We next assess LRP's solution components under various signal strengths and user mobility patterns.

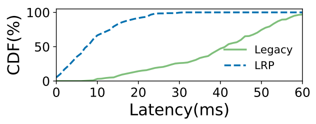
DRX-induced latency reduction (§5.1) As shown in §5.1, LRP helps reduce the DRX doze latency if the inter-packet interval larger than $T_{inactivity\_timer}$ (otherwise the DRX doze latency is always 0 with/without LRP). Figure 15 shows LRP's



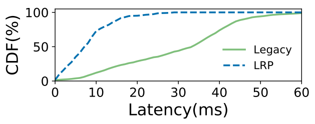
(a) Mobile VR



(b) Gaming



(c) Localization



(d) Object Detection

Figure 13: LTE latency with and without `LRP` in representative apps.

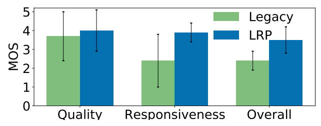


Figure 14: MOS of mobile VR.

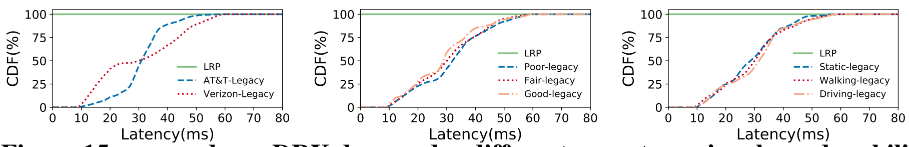


Figure 15: `LRP` reduces DRX doze under different operators, signals, and mobility.

Scenario		AT&T			Verizon			T-Mobile			Sprint			China Mobile		
		Leg.	`LRP`	η	Leg.	`LRP`	η	Leg.	`LRP`	η	Leg.	`LRP`	η	Leg.	`LRP`	η
Static-Poor	Med.	12.0	5.0	1.4×	12.0	9.0	0.3×	11.0	7.0	0.6×	20.0	12.0	0.7×	16.0	5.0	2.2×
	95%	17.0	11.0	0.5×	17.0	17.0	0×	17.0	16.0	0.1×	30.0	26.0	0.2×	26.0	23.0	0.1×
Static-Fair	Med.	13.0	8.0	0.6×	11.0	8.0	0.4×	17.0	13.0	0.3×	18.0	14.0	0.3×	14.0	4.0	2.5×
	95%	17.0	15.0	0.1×	17.0	13.0	0.3×	12.0	6.0	1.0×	32.0	29.0	0.1×	24.0	10.0	1.4×
Static-Good	Med.	13.0	6.0	1.2×	10.0	5.0	1.0×	8.0	6.0	0.3×	13.0	7.0	0.9×	13.0	4.0	2.3×
	95%	17.0	11.0	0.5×	17.0	16.0	0.1×	16.0	11.0	0.5×	27.0	18.0	0.5×	24.0	9.0	1.7×
Walking	Med.	11.0	8.0	0.4×	13.0	6.0	1.2×	12.0	7.0	0.7×	19.0	13.0	0.5×	16.0	9.0	0.8×
	95%	17.0	11.0	0.5×	16.0	16.0	0×	17.0	16.0	0.1×	30.0	26.0	0.2×	30.0	26.0	0.2×
Driving	Med.	14.0	8.0	0.8×	14.0	8.0	0.8×	12.0	8.0	0.5×	17.0	13.0	0.3×	17.0	10.0	0.7×
	95%	18.0	17.0	0.1×	17.0	11.0	0.5×	17.0	16.0	0.1×	29.0	27.0	0.1×	37.0	28.0	0.3×

Table 5: Scheduling latency (ms) in five mobile carriers. η=(Legacy-`LRP`)/`LRP`.

DRX latency reduction under various signal strengths and mobility patterns. We run this test under the most popular setting of $T_{inactivity\_timer}$ = 200ms (Table 3) when the inter-packet interval is $1.5 \cdot T_{inactivity\_timer}$. We also test other intervals and get similar results. In all scenarios, `LRP` reduces the DRX doze latency to 0 for all LTE carriers. This results in 21–41ms mean latency reduction and 40–57ms 95% latency reduction.

Scheduling latency reduction (§5.2) We next quantify the reduction in uplink scheduling latency. The latency reduction ratio, η, is defined as that of the reduced latency and the `LRP` latency. Table 5 shows the results in different carriers, signal strengths, and mobility patterns. In all these scenarios, `LRP` reduces the median scheduling latency by 0.3-2.5×, and reduces the 95th latency by up to 1.7×.

Conflict handling for latency reduction (§5.3) We confirm the effectiveness of `LRP`'s conflict resolution/avoidance. We adapt `LRP`'s APIs to enable/disable the conflict handling in §5.3. Table 6 compares the overall latency with/without `LRP`'s conflict handling. We first illustrate `LRP` can resolve *rouser* and *prefetcher* conflict. We use Localization as its traffic pattern satisfies the condition (long interval) for potential conflict. Compared with no conflict resolution, `LRP` reduces extra 8.82-60.0% latency in all operators. We next evaluate how `LRP` handles data and dummy packets conflicts. The heavy traffic in the Gaming application potentially causes such conflict. We run the Gaming application with `LRP` and the APIs without conflict avoidance. Compared with no conflict avoidance, `LRP` reduces up to 20% extra latency.

Accuracy of critical parameter inference (§5.4) We finally check how accurate our LTE parameter inference is.

Conflicts		AT&T	Ver.	T-M.	Spr.	C. M.
rouser & *prefetcher*	w/o Res.	28.0	30.0	34.0	10.0	13.0
	`LRP`	33.0	36.0	37.0	16.0	16.0
	Extra Red.	17.9%	20.0%	8.82%	60%	23.0%
Data & dummy	w/o Res.	3.5	2.0	2.0	5.0	2.0
	`LRP`	4.0	2.0	2.0	6.0	2.0
	Extra Red.	14.3%	0.0%	0.0%	20%	0.0%

Table 6: Latency (ms) reduction with conflict handle.

	AT&T	Verizon	T-Mobile	Sprint	C. Mobile
Infer T_{rtt}	3.2%	1.5%	2.0%	3.0%	2.0%
Infer $T_{drx\_doze\_max}$	3.0%	1.3%	3.0%	1.3%	2.5%

Table 7: Error rate of `LRP` parameter inference.

For each cell, we first collect ground truth by analyzing the physical/link/RRC-layer signaling messages from MobileInsight. We then use `LRP` component to infer the parameters and compare them with the ground truth. We calculate the average error rate in terms of inference. The results are shown in Table 7. As we can see, the inference error rate is at most 3.2% for both parameters in all 5 operators. `LRP` inference is accurate as argued in §5.4 and §6.

7.3 Overhead

Overhead of dummy messages The dummy messages may incur additional data usage and thus billing. Table 8 shows that `LRP` incurs no more than 0.6KB data per second under all carriers. The data overhead depends on the frequency of calling LRP APIs. For heavy traffic applications (VR, Gaming), the extra overhead is 0.33KB/s while the number for the other two apps is 0.05KB/s. The overhead is acceptable in typical data plans and the extra data is only incurred when `LRP` APIs are called. As explained in §6, `LRP` has minimized the use of dummy for efficient latency reduction.

	AT&T	Verizon	T-Mobile	Sprint	C. Mobile
Extra Data (KB/s)	0.20	0.15	0.41	0.23	0.60
Extra Sig. Msg	3.8%	3.7%	4.3%	3.3%	1.1%
Energy Overhead	1.7%	4.2%	5.8%	2.1%	4.7%

Table 8: Overhead of `LRP`.

Extra signaling message The dummy messages incur extra signaling between the device and the BS. We measure this overhead as shown in Table 8. LRP incurs up to 4.3% messages, which are marginal compared with the total volume of signaling messages. Reducing latency for apps with DRX doze generates more messages. LRP incurs on average 1.6 extra signaling messages per second for Location and Object Detection. While the other two apps with LRP generate 0.8 extra message every second on average.

Energy consumption While LRP exploits the DRX for lower latency, it still respects the LTE's energy saving with accurate timing control and incurs marginal energy cost. We first compare the percentage of the extra ON period with and without LRP. We track the CDRX events with MobileInsight. As shown in Table 8, for all carriers, at most 5.8% of extra ON period is invoked. Furthermore, we fully charge the device and run Object Detection (with DRX doze) and VR (no DRX doze) applications for one hour, and compare the energy consumption with or without LRP. With LRP, two applications incur 2.5% (16.12% to 16.52% of total battery) and 1.0% (37.04% to 37.40% of total battery) extra battery consumption, respectively. This overhead is marginal, as we adjust the timing of *rouser*s to reduce unnecessary energy waste.

Network impact We measure the network impact of LRP in our SDR testbed. Even in the absence of data transfer, the server spends 0.055ms on average to process the collected signal in every subframe (1ms). In contrast, processing LRP's extra signaling costs 0.002ms, about 3.6% extra overhead.

Impact on other users We next examine whether LRP affects those non-LRP users. We test a two-device scenario, with both running the Gaming app. Device A never uses LRP, whereas device B turns on/off LRP in the test. When B does not run LRP, A's average uplink network latency is 15.79ms, and the 95th percentile is 24.0ms. When B activates LRP, A's average latency becomes 15.84ms and the 95th percentile is 24.0ms. Both numbers are not visibly affected. Therefore, the latency of non-LRP device is not affected, regardless of whether the other runs LRP or not.

8 Related Work

Many cross-layer techniques have been designed to improve user experience and application performance in mobile networks (see [19] for a survey). They use lower-layer information to improve video streaming [54], to optimize Web access [12, 26, 34, 35, 51], to name a few. Most such solutions seek to boost the application-perceived throughput. Other recent proposals detect whether LTE is the bottleneck for applications [9], estimate the radio link speed [11], or examine how LTE configurations affect applications [25]. In contrast, we focus on devising LTE latency-oriented reduction solutions.

Early efforts are also made to reduce the LTE network latency. They analyze the latency for Web access over LTE [39, 53], devise application-specific solutions to LTE scheduling latency with modified modem firmware [48], measure the impact of DRX upon LTE from the energy perspective [23], and adjust the RRC parameters to reduce data-plane latency with infrastructure update [38]. Recent work [9, 30] also makes device-based throughput prediction for performance improvement. Our work differs from them since we work on the latency elements that cannot be eliminated with higher throughput. Authors from [16, 33] target reducing one-time connection setup latency, while LRP reduces latency elements for every data packet in the connected state. Other recent efforts seek to refine the 4G/5G network infrastructure [24, 37]. In contrast, we propose an effective solution without root privilege, device firmware change, or infrastructure upgrade.

9 Conclusion

Reducing latency is critical to many delay-sensitive applications, such as mobile AR/VR, mobile gaming, sensing, machine learning, and robot/drone-based image/speech recognition. In mobile networks like 4G LTE, reducing uplink latency is more challenging than its downlink counterpart, since it involves multiple latency elements stemming from power-saving, scheduling, on-demand resource allocation, etc.

We have designed and implemented LRP, a device-based solution to LTE latency reduction without any infrastructure changes. LRP does not require root privilege at the device and works with mobile apps directly. It ensures the network latency is no worse than the legacy 4G LTE, and is applicable to the upcoming 5G. By design, LRP uses small dummy messages with proper timing control and conflict handling, in order to eliminate unnecessary latency components from scheduling and power-saving operations. Our experiments have confirmed its effectiveness with a variety of mobile apps.

In the broader context, reducing latency poses a more challenging problem than improving throughput for the networked system community. In the mobile network domain, various tricks have been invented for boosting throughput (e.g., massive MIMO, more sophisticated modulation, mmWave, etc.). This is not the case for latency. Both its fundamental theory and effective practice are lacking. Moreover, exploring pure device-based solution, which does not require root privilege and has direct access to user application-level information, offers a nice complement to the infrastructure-centric design, which typically takes years to be deployed.

Acknowledgments We greatly appreciate our shepherd Dr. Lin Zhong and the anonymous reviewers for their constructive feedbacks and advices to help improve the quality of this work. We also thank Mr. Kaiyuan Chen and other members at UCLA WiNG for their early effort and feedback. This work is supported in part by NSF CNS-1910150 and CNS-2008026.

References

[1] 3GPP. TS36.213: Evolved Universal Terrestrial Radio Access (E-UTRA); Physical layer procedures, Sep. 2019.

[2] 3GPP. TS36.321: Evolved Universal Terrestrial Radio Access (E-UTRA); Medium Access Control (MAC) protocol specification, Sep. 2019.

[3] 3GPP. TS38.331: NR; Radio Resource Control (RRC); Protocol specification, Oct. 2019.

[4] 3GPP. TS36.331: Radio Resource Control (RRC), 2020.

[5] Michael Abrash. What VR Could, Should, and almost certainly Will be within two years. http://media.steampowered.com/apps/abrashblog/Abrash%20Dev%20Days%202014.pdf, 2014.

[6] Mikhail Afanasov, Alessandro Djordjevic, Feng Lui, and Luca Mottola. Flyzone: A testbed for experimenting with aerial drone applications. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, pages 67–78, 2019.

[7] Amazon. Amazon Alexa. https://www.amazon.com/Amazon-Echo-And-Alexa-Devices/b?ie=UTF8&node=9818047011, Mar. 2020.

[8] Pilar Andres-Maldonado, Pablo Ameigeiras, Jonathan Prados-Garzon, Jorge Navarro-Ortiz, and Juan M Lopez-Soler. Narrowband iot data transmission procedures for massive machine-type communications. *IEEE Network*, 31(6):8–15, 2017.

[9] Arjun Balasingam, Manu Bansal, Rakesh Misra, Kanthi Nagaraj, Rahul Tandra, Sachin Katti, and Aaron Schulman. Detecting if lte is the bottleneck with bursttracker. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–15, 2019.

[10] Tristan Braud, Farshid Hassani Bijarbooneh, Dimitris Chatzopoulos, and Pan Hui. Future networking challenges: The case of mobile augmented reality. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 1796–1807. IEEE, 2017.

[11] Nicola Bui, Foivos Michelinakis, and Joerg Widmer. Fine-grained lte radio link estimation for mobile phones. *Pervasive and Mobile Computing*, 49:76–91, 2018.

[12] Yi Cao, Javad Nejati, Aruna Balasubramanian, and Anshul Gandhi. Econ: Modeling the network to improve application performance. In *Proceedings of the Internet Measurement Conference*, pages 365–378, 2019.

[13] Jiasi Chen and Xukan Ran. Deep learning with edge computing: A review. *Proceedings of the IEEE*, 107(8):1655–1674, 2019.

[14] Tiffany Yu-Han Chen, Lenin Ravindranath, Shuo Deng, Paramvir Bahl, and Hari Balakrishnan. Glimpse: Continuous, real-time object recognition on mobile devices. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, pages 155–168, 2015.

[15] Open AR Cloud. Open AR Cloud. https://www.openarcloud.org/, Mar. 2020.

[16] Edith Cohen and Haim Kaplan. Prefetching the means for document transfer: A new approach for reducing web latency. In *INFOCOM 2000*, volume 2, pages 854–863. IEEE, 2000.

[17] Google Daydream. https://arvr.google.com/daydream/.

[18] WebRTC for Unity. https://github.com/Unity-Technologies/com.unity.webrtc.

[19] Bo Fu, Yang Xiao, Hongmei Julia Deng, and Hui Zeng. A survey of cross-layer designs in wireless networks. *IEEE Communications Surveys & Tutorials*, 16(1):110–126, 2013.

[20] Ismael Gomez-Miguelez, Andres Garcia-Saavedra, Paul D Sutton, Pablo Serrano, Cristina Cano, and Doug J Leith. srsLTE: An open-source platform for LTE evolution and experimentation. In *Proceedings of the Tenth ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation, and Characterization*, pages 25–32, 2016.

[21] Google. Google API for Android, Mar. 2020.

[22] Giulio Grassi, Kyle Jamieson, Paramvir Bahl, and Giovanni Pau. Parkmaster: An in-vehicle, edge-based video analytics service for detecting open parking spaces in urban environments. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, pages 1–14, 2017.

[23] Junxian Huang, Feng Qian, Alexandre Gerber, Z Morley Mao, Subhabrata Sen, and Oliver Spatscheck. A close examination of performance and power characteristics of 4g lte networks. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 225–238, 2012.

[24] Aman Jain, NS Sadagopan, Sunny Kumar Lohani, and Mythili Vutukuru. A comparison of sdn and nfv for redesigning the lte packet core. In *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 74–80. IEEE, 2016.

[25] Fabian Kaup, Foivos Michelinakis, Nicola Bui, Joerg Widmer, Katarzyna Wac, and David Hausheer. Assessing the implications of cellular network performance on mobile content access. *IEEE Transactions on Network and Service Management*, 13(2):168–180, 2016.

[26] Conor Kelton, Jihoon Ryoo, Aruna Balasubramanian, and Samir R Das. Improving user perceived page load times using gaze. In *14th USENIX Symposium on Networked Systems Design and Implementation*, pages 545–559, 2017.

[27] Ronny Krashinsky and Hari Balakrishnan. Minimizing energy for wireless web access with bounded slowdown. In *Proceedings of the 8th annual international conference on Mobile computing and networking*, pages 119–130, 2002.

[28] Kaspersky Lab. Rooting your android: Advantages, disadvantages, and snags. `https://www.kaspersky.com/blog/android-root-faq/17135/`, Jun. 2017.

[29] Zeqi Lai, Y Charlie Hu, Yong Cui, Linhui Sun, Ningwei Dai, and Hung-Sheng Lee. Furion: Engineering high-quality immersive virtual reality on today's mobile devices. *IEEE Transactions on Mobile Computing*, 2019.

[30] Jinsung Lee, Sungyong Lee, Jongyun Lee, Sandesh Dhawaskar Sathyanarayana, Hyoyoung Lim, Jihoon Lee, Xiaoqing Zhu, Sangeeta Ramakrishnan, Dirk Grunwald, Kyunghan Lee, et al. PERCEIVE: Deep Learning-based Cellular Uplink Prediction Using Real-Time Scheduling Patterns. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services (MobiSys'20)*, pages 377–390, 2020.

[31] Kyungmin Lee, David Chu, Eduardo Cuervo, Johannes Kopf, Yury Degtyarev, Sergey Grizan, Alec Wolman, and Jason Flinn. Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, pages 151–165, 2015.

[32] Yuanjie Li, Chunyi Peng, Zengwen Yuan, Jiayao Li, Haotian Deng, and Tao Wang. Mobileinsight: Extracting and analyzing cellular network information on smartphones. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking*, pages 202–215, 2016.

[33] Yuanjie Li, Zengwen Yuan, and Chunyi Peng. A control-plane perspective on reducing data access latency in lte networks. MobiCom '17, pages 56–69, New York, NY, USA, 2017. ACM.

[34] Ravi Netravali and James Mickens. Prophecy: Accelerating mobile page loads using final-state write logs. In *15th USENIX Symposium on Networked Systems Design and Implementation*, pages 249–266, 2018.

[35] Ravi Netravali, Vikram Nathan, James Mickens, and Hari Balakrishnan. Vesper: Measuring time-to-interactivity for web pages. In *15th USENIX Symposium on Networked Systems Design and Implementation*, pages 217–231, 2018.

[36] Ravi Netravali, Anirudh Sivaraman, James Mickens, and Hari Balakrishnan. Watchtower: Fast, secure mobile page loads using remote dependency resolution. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, pages 430–443, 2019.

[37] Akanksha Patel, Mythili Vutukuru, and Dilip Krishnaswamy. Mobility-aware vnf placement in the lte epc. In *2017 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 1–7. IEEE, 2017.

[38] Guillermo Pocovi, Ilaria Thibault, Troels Kolding, Mads Lauridsen, Rame Canolli, Nick Edwards, and David Lister. On the suitability of lte air interface for reliable low-latency applications. In *2019 IEEE Wireless Communications and Networking Conference (WCNC)*, pages 1–6. IEEE, 2019.

[39] Behnam Pourghassemi, Ardalan Amiri Sani, and Aparna Chandramowlishwaran. What-if analysis of page load time in web browsers using causal profiling. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 3(2):1–23, 2019.

[40] PUBG. PUBG. `https://www.pubg.com/`, Mar. 2020.

[41] Qualcomm. QxDM Professional - QUALCOMM eXtensible Diagnostic Monitor. `http://www.qualcomm.com/media/documents/tags/qxdm`.

[42] Unity Technologies Report. `https://www.tweaktown.com/news/74682/index.html`.

[43] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[44] Shu Shi, Varun Gupta, and Rittwik Jana. Freedom: Fast recovery enhanced vr delivery over mobile networks. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, pages 130–141, 2019.

[45] Apple Siri. Siri. https://www.apple.com/siri/, Mar. 2020.

[46] Unity Render Streaming. https://github.com/Unity-Technologies/UnityRenderStreaming.

[47] sysmocom. sysmoUSIM-SJS1 SIM + USIM Card (10-pack). http://shop.sysmocom.de/products/sysmousim-sjs1, 2016.

[48] Zhaowei Tan, Yuanjie Li, Qianru Li, Zhehui Zhang, Zhehan Li, and Songwu Lu. Supporting mobile vr in lte networks: How close are we? *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2(1):1–31, 2018.

[49] Uber. Uber. https://www.uber.com/, Mar. 2020.

[50] Waze. Waze. https://www.waze.com/, Mar. 2020.

[51] Xiufeng Xie, Xinyu Zhang, and Shilin Zhu. Accelerating mobile web loading using cellular link information. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, pages 427–439, 2017.

[52] Ming Yang and Tom Chin. Scheduling request during connected discontinuous reception off period, July 20 2017. US Patent App. 14/996,153.

[53] Zengwen Yuan, Yuanjie Li, Chunyi Peng, Songwu Lu, Haotian Deng, Zhaowei Tan, and Taqi Raza. A machine learning based approach to mobile network analysis. In *2018 27th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–9. IEEE, 2018.

[54] Anfu Zhou, Huanhuan Zhang, Guangyuan Su, Leilei Wu, Ruoxuan Ma, Zhen Meng, Xinyu Zhang, Xiufeng Xie, Huadong Ma, and Xiaojiang Chen. Learning to coordinate video codec with transport protocol for mobile video telephony. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–16, 2019.

APPENDIX

A Notations

Notation	Explanation
T_{drx_doze}	The DRX Doze latency for an uplink packet between it enters buffer during DRX OFF and DRX enters ON state
$T_{drx\_doze\_max}$	The maximum doze latency T_{drx_doze} measured under a cell
T_{sr_wait}	The latency of waiting for an uplink scheduling request (SR)
T_{sr_grant}	The time difference between an SR and sending data using the requested grant
T_{bsr_grant}	The time between the first segment of a packet being sent and the last segment being sent through grants via BSR
T_{retx}	The latency of uplink packet retransmission
$T_{scheduling}$	$T_{scheduling} = T_{sr\_wait} + T_{sr\_grant}$
$T_{inactivity\_timer}$	A new data transmission will restart this timer and keep the device in DRX ON state until this timer expires
$T_{sr\_periodicity}$	The periodicity of subframes where a device can initiate an SR
$T_{interval}$	The time interval between the last and the next expected packet

Table 9: Notation table.

B Discussion on the Corner Case

The corner case happens when the grant from an SR is sufficient for the data packet, but insufficient for the data packet and its *prefetcher*. In this section, we discuss the size of the grant from an SR and the probability of this corner case.

Since the size of a data packet and a *prefetcher* is fixed, the occurrence of the corner case depends on the grant from an SR. The BS assigns a grant for the SR according to 3GPP standard [1]. However, it has the freedom to determine the size of the grant. It can assign certain RBs to the user. The number of the RBs is denoted as N_{PRB}. The RB amount is not sufficient to determine the grant size, which is also affected by the modulation index, denoted as I_{MCS}. A BS sends a grant with I_{MCS}, which is affected by channel condition, device power, etc. N_{PRB} can be selected from a subset of discrete values from {1, ...,110}, depending on the channel bandwidth. I_{MCS} can be selected from a subset of discrete values {0, ..., 63}, depending on the modulation capability of the device. Multiple I_{MCS} can map to the same modulation scheme. The UL data that can be sent using this grant is a function of both N_{PRB} and I_{MCS}. This discrete function, denoted as $F(N_{PRB}, I_{MCS})$, is shown in a 110x44 table in 3GPP 36.213 [1].

$F()$ is monotonically increasing with either N_{PRB} or I_{MCS}. Suppose the selection of N_{PRB} and I_{MCS} are independent. Let the probability of the BS selecting $P(N_{PRB} = j) = p_j$, where $j \in \{1, ..., 110\}$ and $\sum p_j = 1$. Similarly, let $P(I_{MCS} = i) = q_i$, where $i \in \{0, ..., 63\}$ and $\sum q_i = 1$. Let the size of the data packet be a and the size of a *prefetcher* be a'. $(j,i) \in X_1$ if $a \leq F(j,i) < a + a'$. Otherwise, $(j,i) \in X_2$ Therefore, $p_{corner} = \sum_{(j,i)\in X_1} p_j \cdot q_i$.

From the operational traces, a BS tends to assign $N_{PRB} = 2$ or 3 in response to an SR. When $N_{PRB} = 2$, any $I_{MCS} \geq 3$ can guarantee $F(N_{PRB}, I_{MCS}) > 100$. When $N_{PRB} = 3$, any $I_{MCS} \geq 2$ can guarantee $F(N_{PRB}, I_{MCS}) > 100$. In our experiments, >99% of initial grants exceed 100B in all operators. This is sufficient for a small uplink sensory data packet and a small *prefetcher* message.

C Proof for Theorem 5.1

Proof. LRP operates based on the value of $T_{interval}$. When $T_{interval} \geq T_{inactivity\_timer}$, LRP sends a *rouser* to eliminate DRX doze latency. When the next packet arrives later than expected ($T > T_{interval}$), LRP is still very likely to reduce DRX doze latency as the *rouser* sent T_{drx_doze} before the next packet will keep the device in ON state for $T_{inactivity\_timer}$. Therefore, as long as $T \leq T_{interval} - T_{drx\_doze} + T_{inactivity\_timer}$, LRP still reduces DRX doze latency. If $T > T_{interval} - T_{drx\_doze} + T_{inactivity\_timer}$, the device might have already turned to DRX OFF when the next packet arrives. In this situation, the DRX doze latency still exists but LRP does not add extra latency source. Similarly, when the next packet arrives early ($T < T_{interval}$), LRP still reduces doze latency when the *rouser* precedes the data packet, namely $T \geq T_{interval} - T_{drx\_doze}$. LRP cannot eliminate the entire DRX doze latency as the optimal solution, but can still reduce doze latency to $T - (T_{interval} - T_{drx\_doze})$. Otherwise, LRP does not send any *rouser* and the latency is the same compared to no LRP. In summary, if the next packet actually arrives in T where $T_{interval} - T_{drx\_doze} \leq T \leq T_{interval} - T_{drx\_doze} + T_{inactivity\_timer}$, LRP still eliminates or reduces the DRX doze latency. The margin allowed for error (T_{drx_doze} and $T_{inactivity\_timer} - T_{drx\_doze}$) can be 30-80ms in reality depending on common LTE parameters. Otherwise, LRP does not increase the latency.

When $T_{interval} < T_{inactivity\_timer}$, LRP sends a *prefetcher* only (T_{sr_grant} before the next packet) to reduce scheduling latency. If data arrives later, the *prefetcher* is still possible to save its latency if its requested grant can be used by the next packet, which is $T < T_{interval} + T_{sr\_wait}$. Similarly, if data arrives earlier, the *prefetcher* reduces scheduling latency if it is sent before the real data packet, i.e., $T > T_{interval} - T_{sr\_grant}$. When the next scheduled packet arrives in $T_{interval}$ where $T_{interval} - T_{sr\_grant} < T < T_{interval} + T_{sr\_wait}$, LRP still reduces the LTE uplink scheduling latency. This margin allowed for error (T_{sr_grant} and T_{sr_wait}) is usually 8-20ms in reality depending on LTE parameters. Otherwise, the scheduling latency is not reduced but LRP does not incur extra latency. □

BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing

Yoann Ghigoff
Orange Labs
Sorbonne Université, LIP6, Inria

Julien Sopena
Sorbonne Université, LIP6

Kahina Lazri
Orange Labs

Antoine Blin
Gandi

Gilles Muller
Inria

Abstract

In-memory key-value stores are critical components that help scale large internet services by providing low-latency access to popular data. Memcached, one of the most popular key-value stores, suffers from performance limitations inherent to the Linux networking stack and fails to achieve high performance when using high-speed network interfaces. While the Linux network stack can be bypassed using DPDK based solutions, such approaches require a complete redesign of the software stack and induce high CPU utilization even when client load is low.

To overcome these limitations, we present BMC, an in-kernel cache for Memcached that serves requests before the execution of the standard network stack. Requests to the BMC cache are treated as part of the NIC interrupts, which allows performance to scale with the number of cores serving the NIC queues. To ensure safety, BMC is implemented using eBPF. Despite the safety constraints of eBPF, we show that it is possible to implement a complex cache service. Because BMC runs on commodity hardware and requires modification of neither the Linux kernel nor the Memcached application, it can be widely deployed on existing systems. BMC optimizes the processing time of Facebook-like small-size requests. On this target workload, our evaluations show that BMC improves throughput by up to 18x compared to the vanilla Memcached application and up to 6x compared to an optimized version of Memcached that uses the SO_REUSEPORT socket flag. In addition, our results also show that BMC has negligible overhead and does not deteriorate throughput when treating non-target workloads.

1 Introduction

Memcached [24] is a high-performance in-memory key-value store used as a caching-service solution by cloud providers [1] and large-scale web services [5, 40]. Memcached allows such services to reduce web request latency and alleviate the load on backend databases by using main memory to store and serve popular data over the network.

Memcached, however, is prone to bottlenecks introduced by the underlying operating system's network stack, including Linux's [16, 36], since the main goal of general purpose operating systems is to provide applications with flexible abstractions and interfaces. To achieve high throughput and low latency, user applications can give up using the standard kernel interfaces by using kernel-bypass technologies such as DPDK [6] which allow an application to program network hardware and perform packet I/O from userspace. The application that has control of the network hardware is then responsible for implementing a network stack that fits its specific needs [14, 28]. However, kernel-bypass comes with drawbacks. First, it eliminates security policies enforced by the kernel, such as memory isolation or firewalling. Specific hardware extensions, i.e. an IOMMU and SR-IOV [15, 20], or software-based isolation are then required to maintain standard security levels. Second, kernel-bypass relies on dedicating CPU cores to poll incoming packets, trading off CPU resources for low latency. This prevents the cores from being shared with other applications even when the client load is low. Third, kernel-bypass requires an extensive re-engineering of the existing application in order to achieve high performance with a dedicated network stack.

In this paper, we propose BPF Memcached Cache (BMC) to address the kernel bottlenecks impacting Memcached. BMC focuses on accelerating the processing of small GET requests over UDP to achieve high throughput as previous work from Facebook [13] has shown that these requests make up a significant portion of Memcached traffic. Contrary to hardware-specific accelerators, BMC runs on standard hardware and thus can be deployed on infrastructure with heterogeneous hardware. BMC relies on a *pre-stack processing* approach that consists in intercepting requests directly from the network driver, before they are delivered to the standard network stack, and processing them using an in-kernel cache. This provides the ability to serve requests with low latency and to fall back to the Memcached application when a request cannot be treated by BMC. BMC can leverage modern network card features such as multi-queues to process multiple requests in parallel (see Figure 1). In addition, BMC uses a separate lock for each cache entry to introduce minimal overhead and allow performance to scale with the number of cores.

Running BMC at the kernel level raises safety issues as a bug in its implementation could put the entire system at risk. To address this issue, BMC is implemented using eBPF. The Berkeley Packet Filter (BPF) [37], and its extended version, eBPF, is a bytecode and a safe runtime environment offered by the Linux kernel to provide userspace an approach to inject code inside the kernel. The Linux kernel includes a static analyzer to check that the injected code is safe before it can be executed, which limits the expressiveness of the injected code. We show how to circumvent this limitation by partitioning complex functionality into small eBPF programs and by bounding the data that BMC processes. Using eBPF also allows BMC to be run without requiring any modification to the Linux kernel or to the Memcached application, making it easy to deploy on existing systems. The eBPF bytecode of BMC is compiled from 513 lines of C code and is JIT compiled by the Linux kernel. This results in BMC introducing very little overhead into the OS network stack.

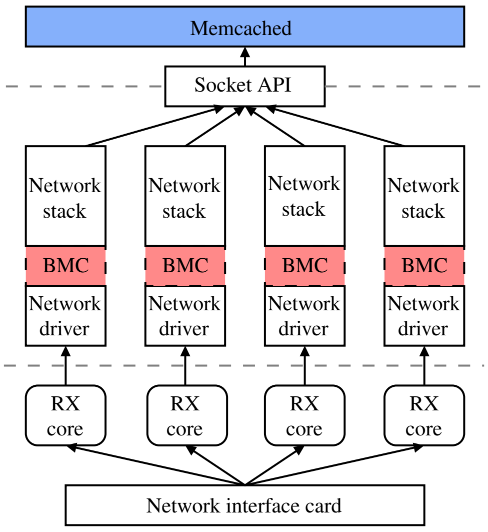


Figure 1: General architecture

The main results of this paper include:

- The identification of the bottlenecks of Memcached when processing requests over UDP. We propose MemcachedSR, a modified version of Memcached that uses the SO_REUSEPORT socket option to scale with the number of threads, improving throughput by 3x compared to the vanilla Memcached.
- The evaluation of BMC under our target workload consisting of small requests. In this setting, BMC improves the throughput by up to 6x with respect to MemcachedSR and by up to 18x with respect to vanilla Memcached.
- The evaluation of BMC under a non-target workload that consists of large requests not processed by BMC. In this setting, BMC has negligible overhead and does not deteriorate throughput with respect to MemcachedSR.
- The comparison of BMC with a dummy cache that shows that BMC's design is well suited for high throughput performance as it does not introduce unnecessary complexity.
- The comparison of Memcached running with BMC against a Memcached implementation based on Seastar, a networking stack for DPDK [6]. Our results show that Memcached with BMC achieves similar throughput to Seastar but uses 3x less CPU resources.

The rest of this paper is organized as follows. Section 2 provides background on Memcached and the OS network stack bottlenecks it suffers from. Section 3 describes BMC and its design. Section 4 discusses implementation details. Section 5 presents the experimental results. Section 6 discusses the generalization of BMC and its memory allocation challenges. Section 7 presents related work. Finally, Section 8 concludes the paper.

2 Background and motivation

This section describes the limitations of Memcached that motivate our work, and describes the eBPF runtime used to implement BMC.

2.1 Memcached

Memcached [10] is a mature in-memory key-value store traditionally used as a cache by web applications in a datacenter environment to speed up request processing and reduce the load on back-end databases. Because of its popularity, a lot of work has been put into optimizing it [35, 44].

A Memcached server operates on *items*, which are objects used to store a key and its associated value and metadata. Clients send requests to a Memcached server using a basic command-based protocol, of which GET and SET are the most important commands. A GET *key* command retrieves the value associated with the specified key if it is stored by Memcached and a SET *key value* command stores the specified key-value pair. A GET command can also be used as a multiget request when a client needs to retrieve multiple values at once. Requests can be sent using either the TCP or the UDP transport protocol.

The data management of Memcached has been well optimized and relies on slab allocation, a LRU algorithm and a hash table to allocate, remove and retrieve items stored in memory. Previous studies [16, 30] have shown that Memcached performance and scalability are heavily impaired by OS network stack bottlenecks, especially when receiving a large number of small requests. Since Facebook's production workloads show a 30:1 distribution between GET and SET commands, Nishtala et al. [40] proposed using UDP instead of TCP for GET commands to avoid the cost of TCP processing.

To gain additional insight into the performance of a Memcached server using UDP to receive GET requests, we profiled the CPU consumption while trying to achieve maximum throughput (the experimental setup is described in Section 5). As shown in Figure 2, more than 50% of Memcached's runtime is spent executing system calls. Moreover, the CPU usage of both sys_recvfrom and sys_sendmsg increases as more threads are allocated to Memcached. When eight threads are used by Memcached, the total CPU usage of these three system calls reaches 80%.

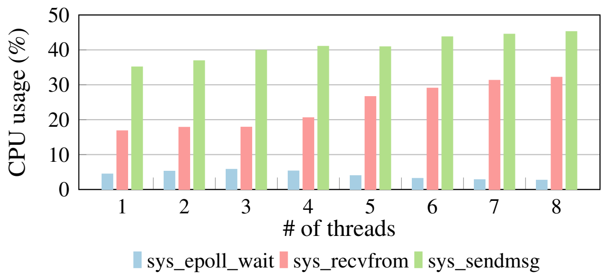


Figure 2: CPU usage of the three most used system calls by Memcached

Figure 3 shows the throughput of the vanilla Memcached application when varying the number of threads (and cores). The results show that vanilla Memcached does not scale and that its performance even deteriorates when more than four threads are used. Table 1 shows the top ten most time consuming functions measured by the *perf* tool while running Memcached with eight threads, all of them are kernel functions. The *native_queued_spin_lock_slowpath* and *__udp_enqueue_schedule_skb* functions account for a total of 28.63% of the processing time of our machine under test and are used to push packets to the UDP socket queue. The kernel's socket queues are data structures shared between the Memcached threads and the kernel threads responsible for the execution of the network stack, and therefore require lock protection. In the case of Memcached, a single UDP socket is used and its queue is shared between the cores receiving packets from the NIC and the cores running the application, leading to lock contention. This lock contention is then responsible for the decrease in Memcached throughput.

To allow Memcached to scale with the number of threads, we have modified the version 1.5.19 of Memcached to use the SO_REUSEPORT socket option. The SO_REUSEPORT option allows multiple UDP sockets to bind to the same port. We refer to this modified Memcached as *MemcachedSR* in the rest of the paper. We use this option to allocate a UDP socket per Memcached thread and bind each socket to the same port. Received packets are then equitably distributed between each socket queue by the Linux kernel which reduces lock contention. As shown in Figure 3, MemcachedSR scales with the number of threads and achieves a throughput that is up to 3 times higher than the vanilla version of Memcached.

Despite the scalability of MemcachedSR, there is still room for improvement as Memcached requests still have to go through the whole network stack before they can be processed by the application.

Function	% CPU utilization
native_queued_spin_lock_slowpath	17.68%
__udp_enqueue_schedule_skb	10.95%
clear_page_erms	5.04%
udp4_lib_lookup2	3.23%
_raw_spin_lock	3.04%
fib_table_lookup	2.90%
napi_gro_receive	2.27%
nfp_net_rx	1.97%
i40e_napi_poll	1.32%
udp_queue_rcv_one_skb	1.14%

Table 1: Top ten most CPU-consuming functions on a Memcached server under network load

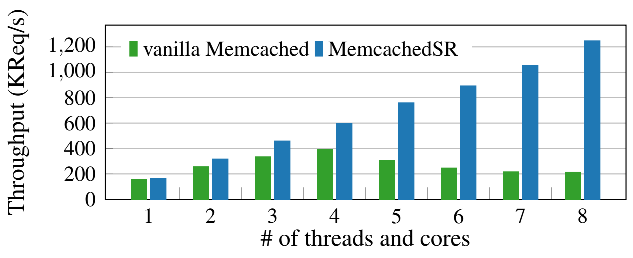


Figure 3: Vanilla Memcached vs. MemcachedSR

2.2 eBPF

The Berkeley Packet Filter (BPF) [37] is an in-kernel interpreter originally designed to run packet filters from userspace using a reduced instruction set. BPF has evolved into the extended BPF (eBPF), which introduces a new bytecode and just-in-time compilation for improved performance. An eBPF program can be loaded from userspace by the Linux kernel and triggered by a specific kernel event. The eBPF program is then run whenever the event is triggered.

eBPF programs can maintain and access persistent memory thanks to kernel data structures called *BPF maps*. Maps are designed to store arbitrary data structures whose size must be specified by the user application at creation time. They can be used for communicating between different eBPF programs or between eBPF programs and user applications. Furthermore, eBPF programs can call a restricted set of kernel functions, called helpers, allowing eBPF programs to interact with the system and access specific kernel data (e.g. map data, time since boot up). *Tail calls* allow an eBPF program to call another eBPF program in a continuation-like manner. The eBPF bytecode backend is supported by the Clang/LLVM compiler toolchain, which allows using the C language to write eBPF programs in a high-level language.

Because running user-space code inside the kernel can impact the system's security and stability, the Linux kernel calls the in-kernel eBPF verifier every time it loads an eBPF program to check if the program can be safely attached and executed. The goal of the verifier is to guarantee that the program meets two properties: safety, i.e., the program neither accesses unauthorized memory, nor leaks kernel information, nor executes arbitrary code, and liveness, i.e., the execution of the program will always terminate.

To analyze an eBPF program, the verifier creates an abstract state of the eBPF virtual machine [9]. The verifier updates its current state for each instruction in the eBPF program, checking for possible out-of-bounds memory accesses or jumps. All conditional branches are analyzed to explore all possible execution paths of the program. A particular path is valid if the verifier reaches a *bpf exit* instruction and the verifier's state contains a valid return value or if the verifier reaches a state that is equivalent to one that is known to be valid. The verifier then backtracks to an unexplored branch state and continues this process until all paths are checked.

Because this verification process must be guaranteed to terminate, a complexity limit is enforced by the kernel and an eBPF program is rejected whenever the number of explored instructions reaches this limit. Thus, the verifier incurs false positives, i.e. it can reject eBPF programs that are safe. In Linux 5.3, the kernel version used to implement BMC, this limit is set to 1 million instructions. Other parameters, such as the number of successive branch states, are also used to limit path explosion and the amount of memory used by the verifier. Since Linux 5.3, the verifier supports bounded loops in eBPF programs by analyzing the state of every iteration of a loop. Hence, the verifier must be able to check every loop iteration before hitting the previously-mentioned instruction complexity limit. This limits the number of loop iterations as well as the complexity of the instructions in the loop body. Moving data of variable lengths between legitimate memory locations requires a bounded loop and conditional instructions to provide memory bounds checking, which in turn increase the complexity of an eBPF program. Finally, eBPF does not support dynamic memory allocation, instead eBPF programs have to rely on eBPF maps (array, hashmap) to hold a fixed number of specific data structures.

Because of all these limitations, eBPF is currently mostly used to monitor a running kernel or to process low-layer protocols of network packets (i.e. L2-L4). Processing application protocols is more challenging but is required to allow the implementation of more complex network functions [38].

3 Design

In this section, we present the design of BMC, a safe in-kernel accelerator for Memcached. BMC allows the acceleration of a Memcached server by caching recently accessed Memcached data in the kernel and by relying on a *pre-stack processing* principle to serve Memcached requests as soon as possible after they have been received by the network driver. This approach allows BMC to scale to multicore architectures by leveraging modern NIC's multi-queue support to run BMC on each individual core for each received packet. The execution of BMC is transparent to Memcached, and Memcached does not need any modification to benefit from BMC. In the rest of this section, we first present the pre-stack processing approach. We then describe the BMC cache and how its coherence is insured.

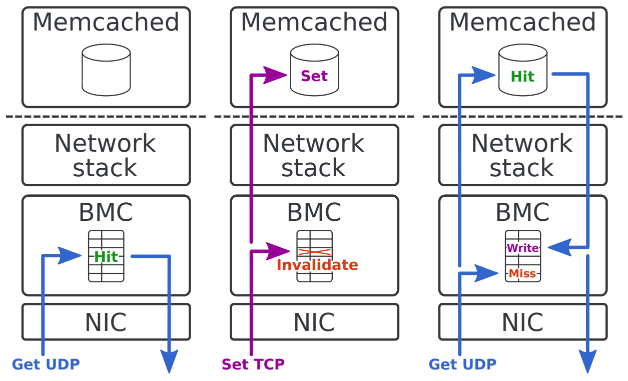
3.1 Pre-stack processing

BMC intercepts network packets at the network-driver level to process Memcached requests as soon as possible after they have been received by the NIC. BMC filters all network packets received by the network driver based on their destination port to only process Memcached network traffic. It focuses on processing GET requests using the UDP protocol and SET requests using the TCP protocol. Figure 4 illustrates how pre-stack processing allows BMC to leverage its in-kernel cache to accelerate the processing of Memcached requests.

When processing a GET request (4a), BMC checks its in-kernel cache and sends back the corresponding reply if it finds the requested data. In that case, the network packet containing the request is never processed by the standard network stack, nor the application, freeing CPU time.

SET requests are processed by BMC to invalidate the corresponding cache entries and are then delivered to the application (4b). After a cache entry has been invalidated, a subsequent GET request targeting the same data will not be served by BMC but rather by the Memcached application. BMC always lets SET requests go through the standard network stack for two reasons. First, it enables reusing the OS TCP protocol implementation, including sending acknowledgments and retransmitting segments. Second, it ensures SET requests are always processed by the Memcached application and that the application's data stays up-to-date. We choose not to update the in-kernel cache using the SET requests intercepted by BMC because TCP's congestion control might reject new segments after its execution. Moreover, updating the in-kernel cache with SET requests requires that both BMC and Memcached process SET requests in the same order to keep the BMC cache consistent, which is difficult to guarantee without a overly costly synchronization mechanism.

When a miss occurs in the BMC cache, the GET request is passed to the network stack. Then, if a hit occurs in Memcached, BMC intercepts the outgoing GET reply to update its cache (4c).



(a) Lookup (b) Invalidation (c) Update

Figure 4: BMC cache operations

Pre-stack processing offers the ability to run BMC on multiple cores concurrently. BMC can benefit from modern NIC features such as multi-queue and RSS to distribute processing among multiple CPU cores. The set of cores used to run BMC can also be fine-tuned in order to share a precise memory level (CPU caches, NUMA node, etc.). The performance of BMC can efficiently scale by configuring NICs to use multiple RX queues and mapping them to different cores. Pre-stack processing also enables running specialized code without having to modify existing software. Contrary to kernel-bypass, this approach does not require a whole NIC to be given to a userspace process and other applications can share the network hardware through the kernel network stack as usual.

3.2 BMC cache

The BMC cache is designed as a hash table indexed by Memcached keys. It is a direct-mapped cache, meaning that each bucket in the hash table can only store one entry at a time. BMC uses the 32-bit *FNV-1a* [21] hash function to calculate the hash value. Because this is a rolling hash function that operates on a single byte at a time, it allows BMC to compute the hash value of a key while parsing the Memcached request. The hash value is reduced to an index into the cache table by using the modulo operator. Each cache entry contains a valid bit, a hash value, a spin lock, the actual stored data, and the size of the data. This cache design offers constant-time complexity for lookup, insertion, and removal operations. To validate a cache hit, BMC checks that the valid bit of a cache entry is set and that the stored key is the same as that of the processed request.

The BMC cache is shared by all cores and does not require a global locking scheme since its data structure is immutable. However, each cache entry is protected from concurrent access using a spin lock.

4 Implementation

This section explains how BMC deals with the eBPF limitations to meet the required safety guarantees.

4.1 Bounding data

The verification of a loop contained in a single program may hit the maximum number of eBPF instructions the verifier can analyze. Loop complexity depends on the number of iterations and the complexity of the body. To make the verification of loops possible, BMC bounds the data it can process. It first limits the length of Memcached keys and values. BMC uses a loop to copy keys and values from a network packet to its cache, and vice-versa. For every memory copy, BMC must guarantee that it neither overflows the packet bounds nor overflows the cache memory bounds using fixed data bounds. Bounds checking then increases the loop complexity. To ensure the complexity of a single eBPF program does not exceed the maximum number of instructions the verifier can analyze, we empirically set the maximum key length BMC can process to 250 bytes and the maximum value length to 1000 bytes. Requests containing keys or values that exceed these limits are transmitted to the Memcached application. We also limit to 1500 the number of individual bytes BMC can read from a packet's payload in order to parse the Memcached data, bounding the complexity of this process. According to Facebook's workload analysis [13], about 95% of the observed values were less than 1000 bytes. Moreover, the Memcached protocol sets the maximum length of keys to 250 bytes. Hence, bounding the BMC data size does not have a big practical impact.

4.2 Splitting complex functions

In order to avoid reaching the limits of the eBPF verifier, BMC's functional logic is separated into multiple small eBPF programs, as each eBPF program is checked for safety independently. Each program then relies on *tail calls* to jump to the next program and continue packet processing without interruption. Linux limits the maximum number of successive tail calls to 33, preventing infinite recursion. However BMC uses at most three successive tail calls.

BMC is implemented using seven eBPF programs that are written in C code and are compiled to eBPF bytecode using Clang/LLVM version 9 and the default eBPF instruction set. The processing logic of BMC is split into two chains: one chain is used to process incoming Memcached requests and the other is used to process outgoing replies. Figure 5 illustrates how BMC's eBPF programs are divided. BMC's eBPF programs consist of a total of 513 lines of C code.

4.2.1 Incoming chain

The incoming chain is composed of five eBPF programs. It is attached to the XDP [27] driver hook and is executed

Program name	# of eBPF instructions	# of analyzed instructions	analysis time (µs)	# of CPU instructions
rx_filter	87	31 503	11 130	152
hash_keys	142	787 898	290 588	218
prepare_packet	178	181	47	212
write_reply	330	398 044	132 952	414
invalidate_cache	163	518 321	246 788	224
tx_filter	61	72	43	104
update_cache	125	345 332	95 615	188

Table 2: Complexity of BMC's programs. Column 2 represents the number of eBPF bytecode instructions of the program compiled from C code. Columns 3 and 4 respectively show the number of eBPF bytecode instructions processed by the Linux verifier and the time spent for this analysis. Column 5 shows the number of CPU instructions after JIT compilation.

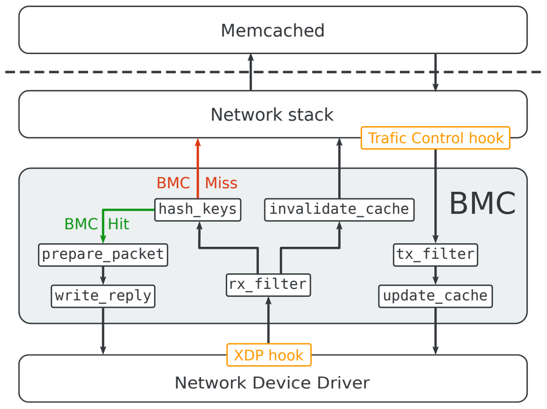


Figure 5: Division of BMC into seven eBPF programs

whenever a new packet is processed by the network driver. This hook is the earliest point in the network stack at which an eBPF program can be attached and allows BMC to use pre-stack processing to save the most CPU cycles by responding to Memcached requests as soon as possible.

rx_filter. The goal of this first eBPF program is to filter packets corresponding to the Memcached traffic using two rules. The first rule matches UDP packets whose destination port corresponds to Memcached's and whose payload contains a GET request. The second rule matches TCP traffic whose destination port also corresponds to Memcached's. The incoming chain branches based on which rule matches. If neither rule matches, the packet is processed by the network stack as usual.

hash_keys. This program computes hashes for every Memcached GET key contained in the packet. It then checks the corresponding cache entries for any cache hit or hash collision and saves the key hashes that have been hit in a per-cpu array used to store context data for the execution of the chain.

prepare_packet. This eBPF program increases the size of the received packet and modifies its protocol headers to prepare the response packet, swapping the source and destination Ethernet addresses, IP addresses and UDP ports. It then calls the last eBPF program of this branch of the chain. The maximum number of bytes BMC can add to the packet is limited by the network driver implementation. In our current implementation of BMC, this value is set to 128 bytes based on the different network drivers BMC attaches to, and it can be increased to a higher value to match other network driver implementations.

write_reply. This eBPF program retrieves a key hash saved in the per-cpu array to copy the corresponding cache entry to the packet's payload. If the table contains multiple key hashes, this eBPF program can call itself to copy as many items as possible in the response packet. Finally, this branch of the incoming chain ends by sending the packet back to the network.

invalidate_cache. The second branch of the incoming chain handles Memcached TCP traffic and contains a single eBPF program. This program looks for a SET request in the packet's payload and computes the key hash when it finds one to invalidate the corresponding cache entry. Packets processed by this branch of the incoming chain are always transmitted to the network stack so that Memcached can receive SET requests and update its own data accordingly.

4.2.2 Outgoing chain

The outgoing chain is composed of two eBPF programs to process Memcached responses. It is attached to the Traffic Control (TC) egress hook and is executed before a packet is sent to the network.

tx_filter. The first eBPF program of this chain serves as a packet filter and applies a single rule on outgoing packets. The rule matches UDP packets whose source port corresponds to Memcached's. In this case the second eBPF program of the chain is called, otherwise the packet is sent to the network as usual.

update_cache. The second eBPF program checks if the packet's payload contains a Memcached GET response. If positive, its key is used to index the BMC cache and the response data is copied in the corresponding cache entry. The network stack then carries on its execution and the Memcached response is sent back to the network.

Table 2 provides complexity metrics for each eBPF program. For the most complex ones, the number of eBPF instructions the Linux verifier has to analyze to ensure their safety is a thousand times higher than their actual number of instructions. The table also shows that it is necessary to divide BMC's logic into multiple eBPF programs to avoid reaching the limit of 1,000,000 instructions that can be analyzed by the Linux verifier.

5 Evaluation

In this section, we evaluate the performance of MemcachedSR running with BMC. We aim to evaluate the throughput gain offered by BMC and how performance scales with the number of cores when processing a target workload that consists of small UDP requests. We also evaluate MemcachedSR with BMC on a non-target workload to study the overhead and impact of BMC on throughput when it intercepts Memcached requests but does not cache them. We show that the increase in throughput can be obtained without allocating additional memory, and that the cache memory can be partitioned between the Memcached application and BMC. We compare BMC with a dummy cache implementation and show that its design is efficient for high performance. Finally, we compare MemcachedSR running with BMC to Seastar, an optimized networking stack based on DPDK. We study their performance using our target workload and a workload that uses both TCP and UDP requests. We also measure their CPU resource consumption for an equivalent client load and show that BMC allows saving CPU resources.

5.1 Methodology

Platform. Our testbed consists of three machines: one acting as the Memcached server under test, and two as the clients. The server machine is equipped with a dual socket motherboard and two 8-core CPUs (Intel Xeon E5-2650 v2 @ 2.60 GHz) with HyperThreading disabled, 48 GB of total memory and two NICs (one Intel XL710 2x40GbE and one Netronome Agilio CX 2x40GbE). The other two machines are used as clients to send traffic and are equipped with the same Intel Xeon CPU and an Intel XL710 2x40GbE NIC. One client is connected back to back to the server using its two network ports while the other client is connected using a single port. In total, the server machine uses three network ports to receive traffic from the clients. In all experiments, the server machine runs Linux 5.3.0.

Target workload and Method. Our target workload is the following: the client applications generate skewed workloads based on established Memcached traffic patterns [13]. Clients use a non-uniform key popularity that follows a Zipf distribution of skewness 0.99, which is the same used in Yahoo Cloud Serving Benchmark (YCSB) [19]. MemC3 [23] is an in-memory key-value store that brings carefully designed algorithms and data structures to Memcached to improve both its memory efficiency and scalability for read-mostly workloads. Similarly to the evaluations performed in the MemC3 paper, our workload consist of a population of 100 million distinct 16-byte keys and 32-byte values. By default, we allocate 10 GB of memory for the Memcached cache and 2.5 GB for the BMC cache. With this amount of memory, the Memcached cache can hold about 89 million items while the BMC cache can hold 6.3 million. Hence, some items can be missing from both the BMC and the Memcached cache. The memory allocated to both Memcached and BMC is not only used for keys and values but also stores metadata. For each cache entry in BMC, 17 bytes are used as metadata. Before each experiment, the clients populate Memcached's cache by sending a SET request for every key in the population. Note that this does not populate BMC's cache as it is only updated when the application replies to GET requests.

The client applications send requests at the rate of 12 million requests per second (Req/s) in an open-loop manner in order to achieve the highest possible throughput and highlight bottlenecks. A total of 340 clients are simulated to efficiently distribute requests among multiple cores on the server by leveraging the NICs' multi-queue and RSS features. We further refer to these cores as *RX cores*. We limit our evaluations to a maximum of 8 RX cores on a single CPU to enforce NUMA locality with the NICs.

Table 3 summarizes this target workload as well as other default evaluation settings that are used in the following experiments unless otherwise specified.

Key distribution	Zipf (0.99)
Key size	16 bytes
Value size	32 bytes
Key population	100 million
BMC to Memcached cache size ratio	25%
Number of Memcached application threads	8
Number of RX cores	8

Table 3: MemC3-like evaluation settings

5.2 Throughput

Target workload. We evaluate the throughput of Memcached under three configurations: vanilla Memcached alone, MemcachedSR alone and MemcachedSR with BMC. We also evaluate how these three configurations scale with the number of cores. We allocate the same CPU resources for all three configurations. For all configurations, we vary the number of threads the application uses to process requests simultaneously and dedicate cores to the application by pinning its threads. For MemcachedSR with BMC, we also vary the number of queues configured on each of the server's NICs and use the same cores to handle interrupts. This allows BMC to be executed by each core serving interrupts in parallel. For the vanilla Memcached application alone and MemcachedSR alone, 8 cores are used to execute the network stack.

Figure 6 shows the throughput achieved by these three configurations. As mentioned in Section 2.1, the vanilla Memcached application does not scale due to the socket lock contention; at best it achieves 393K requests per second using 4 cores. MemcachedSR offers better scalability and achieves 1.2M requests per second when using 8 cores. For MemcachedSR with BMC, the overall system throughput is split between requests answered using the BMC cache and requests handled by the application. When running on a single core, MemcachedSR with BMC achieves 1M requests per second, which is 6x the throughput of both vanilla Memcached and MemcachedSR. When BMC runs on 8 cores, the server achieves a throughput of 7.2M requests per second, 6.3 million being processed by BMC, the rest being processed by Memcached. This is 18x better performance with respect to vanilla Memcached and 6x with respect to MemcachedSR.

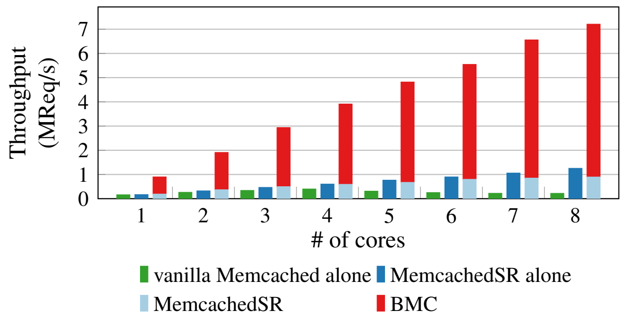


Figure 6: Throughput of BMC

Worst-case workload. We now change our workload to use 8KB Memcached values instead of 32 bytes. This is BMC's worst-case workload since the Memcached requests are still analyzed by BMC but the values are too large to be stored in its cache, thus the Memcached application is always used to serve the requests. To study the impact of the additional processing of BMC, we compare the throughput of the MemcachedSR application alone and that of the MemcachedSR application running with BMC. Figure 7 shows that BMC's additional processing has negligible overhead and does not significantly deteriorate the application throughput.

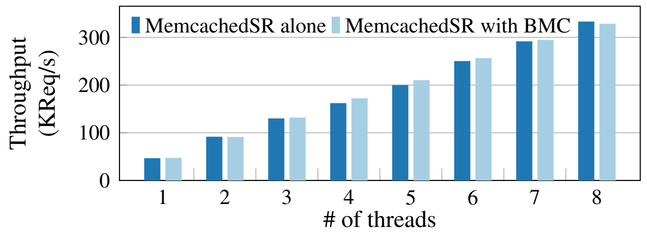


Figure 7: Throughput under a worst-case workload

We now modify the workload so that part of it are requests that BMC targets while the rest is for 8KB values. We then and evaluate how varying the ratio of the target requests affects throughput. As shown in Figure 8, BMC improves throughput by 4x compared to MemcachedSR alone when the workload consists of 25% of targeted requests even though it does not speed up the majority of requests received. This shows that BMC is valuable even when the workload contains few target requests and that BMC further improves throughput as the ratio of target requests increases.

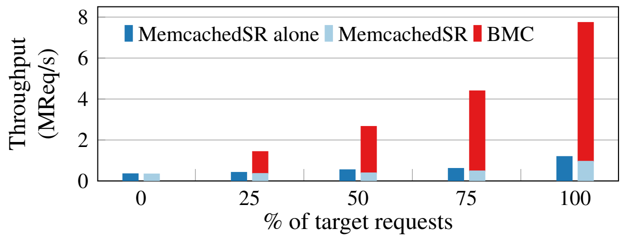


Figure 8: MemcachedSR vs. MemcachedSR with BMC throughput for varying request size distributions.

5.3 Cache size

We then evaluate the impact of BMC's cache size on throughput. In this experiment, we use a total of 10 GB of memory and split it between the Memcached cache and the BMC cache, varying the distribution from 0.1% of memory allocated to BMC to a maximum of 40%. The latter corresponds to a size of 4 GB for the BMC cache, which is the maximum size accepted by the Linux kernel for the allocation of a single eBPF map [7]. The results are shown in Figure 9 where the total system throughput is broken down into hits in the BMC cache, and hits and misses in the application cache. For all distribution schemes tested, there is an increase in performance compared to running MemcachedSR alone. The best throughput is achieved when BMC uses 25% of the total memory. In this case, the BMC cache size is well-suited to store the hottest items of the Zipf distribution. Throughput decreases from 25% to 40% because the Memcached cache hit rate shrinks from 89% to 43% as its cache gets smaller. This causes the BMC cache hit rate to diminish as well because only responses from Memcached cache hits allow entries of the BMC cache to be updated. When only 0.1% (10 MB) of the memory is used by the BMC cache, throughput is multiplied by 2.3 compared to the best throughput achieved by MemcachedSR alone, showing that BMC offers good performance even with minimal memory resources.

5.4 BMC processing latency

We now evaluate the overhead induced by a BMC cache miss, which characterizes the worst-case scenario since BMC processes a request and executes additional code that does not lead to any performance benefit. To characterize this overhead,

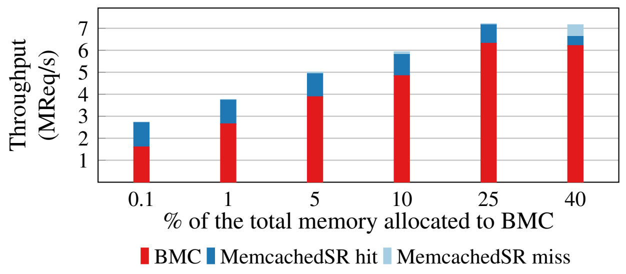


Figure 9: System throughput under various memory partition schemes

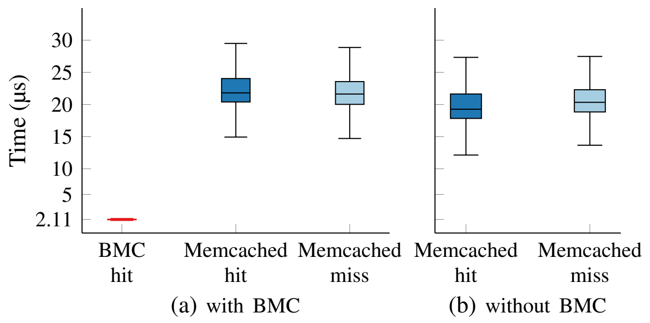


Figure 10: Time to receive, process and reply to a request.

we use *kprobes* to measure the total time required to receive, process and reply to a Memcached request. The *nfp_net_rx* and *nfp_net_tx* driver functions are respectively instrumented to record the time at which a request is received and the corresponding reply is sent back to the network. In this experiment, a single client machine is used to send requests to the server and a single Memcached key is used to ensure that a cache result is always either a hit or a miss. After sending a request, the client always waits for the server's reply to make sure the server does not process more than one request at a time.

Figure 10 shows the time distribution of 100,000 measurements for cache hits and misses separately. Figure 10a shows the distributions of MemcachedSR running with BMC as well as BMC hits. For Memcached hits, the valid bit of BMC's cache entries is never set to ensure BMC lookups result in a cache miss and that the request is always processed by the Memcached application. However, the BMC cache is still updated to measure additional data copies. The median of the distribution of BMC cache hits is 2.1 µs and that of Memcached cache hits and misses are respectively 21.8 and 21.6 µs. Hence, a BMC cache hit can reduce by 90% the time required to process a single request. Running the same experiment on MemcachedSR without BMC (Figure 10b) shows that the processing time of both Memcached hits and misses is lower by about 1 µs. This shows that BMC has a negligible processing overhead compared to the total time required for the execution of the Linux network stack and the Memcached application. Moreover, this additional processing time is entirely recovered by a single BMC cache hit.

Next we study the impact of the processing time of the kernel cache on its throughput. To do so we have implemented a dummy cache that always replies the same response and whose processing time can be parameterized using a empty loop. This dummy cache is implemented using a single eBPF program and is attached to the XDP network-driver hook just like BMC. Figure 11 shows the throughput of the dummy cache while varying its processing time and compares it with the actual BMC cache perfoming cache hits. This experiment demonstrates that the cache throughput is highly dependent on its processing time: increasing the processing time from 100 ns to 2000 ns decreases throughput by a factor of 4.5. The average time to perform a cache hit in BMC is fairly close to the time of the dummy cache with no additional processing time; this shows that choosing simple and fast algorithms for BMC's cache design introduces little processing overhead and contributes to its high throughput performance. Implementing overly complex algorithms may lead to a sharp drop in performance. Hence, adding new features to BMC, such as an eviction algorithm, must be well thought out to result in an improved hit rate that compensates for the additional processing time.

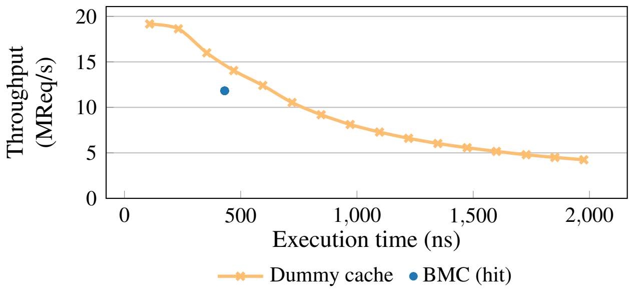


Figure 11: Execution time impact on throughput.

5.5 Impact on concurrent network applications.

As BMC intercepts every network packet before the OS network stack, its execution may have a performance impact on other networking applications running on the same host. To study this impact, we use *iperf* to transfer 10GB of data over TCP from one client machine to the server machine while serving Memcached requests, and measure the time required to complete this transfer. This experiment is conducted for MemcachedSR alone and MemcachedSR with BMC, while varying clients' request throughput. Figure 12 shows the results. When there is no load, the baseline transfer time is 4.43 seconds. When the Memcached load rises, the time required to complete the data transfers increases since the cores are busy processing incoming Memcached traffic in addition to the *iperf* traffic, which leads to an increase in TCP retransmission rate and UDP packet drops. When using BMC, the transfer time is lowered when the server is under load as CPU resources are saved when BMC processes Memcached requests in place of the application. For a Memcached load of 5000K Req/s, BMC allows *iperf* to complete its transfer 83% faster compared to the configuration in which Memcached processes the requests alone and the OS network stack is always executed.

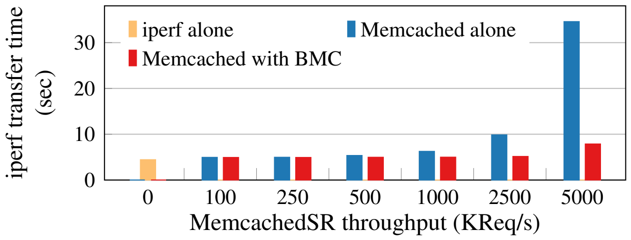


Figure 12: Time required to transfer 10GB of data using *iperf* under client load.

5.6 Kernel-bypass comparison

We now compare MemcachedSR with BMC against the Memcached implementation from Seastar [4] 20.05, a framework for building event-driven applications that comes with its own network stack built on top of DPDK.

Target workload. In this experiment, we evaluate the throughput of Seastar and compare the results with MemcachedSR running with BMC. We perform the experiment using a single client to generate our target UDP workload as Seastar does not support multiple network interfaces. This single client alone generates 4.5 million requests per second. Figure 13 shows the throughput of Seastar and MemcachedSR with BMC when varying the number of cores. BMC is able to process the workload generated by a single client machine using 4 cores. Using the same number of cores, Seastar achieves 443K requests per second. Seastar's throughput increases to 946K requests per second when using 8 cores. We are not sure why Seastar's throughput drops when using 2 cores; our investigations revealed that this only happens when Seastar receives UDP packets and that Seastar performs best when it processes Memcached requests over TCP.

Workload mixing UDP and TCP requests. As our preliminary investigation shows that Seastar performs best on TCP, we change our workload to send half of the Memcached requests with TCP while the other half keeps using UDP. This workload coincides with a Memcached deployment for which the protocol used by clients cannot be anticipated. Figure 14 shows that the throughput of both configurations scales with the number of cores. Seastar's high-performance TCP stack enables its Memcached implementation to process 2.3 million requests per second when using 8 cores. Accelerating the processing of UDP requests allows MemcachedSR with BMC

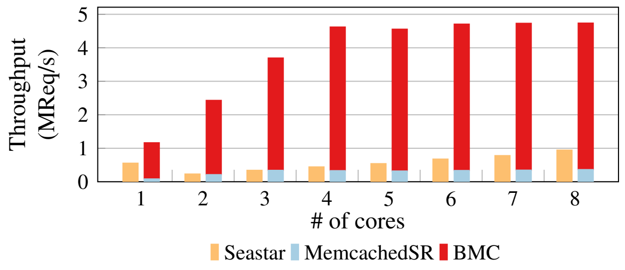


Figure 13: Seastar vs. BMC throughput

to achieve similar throughput when using 3 cores. Increasing the number of cores does not increase the throughput of MemcachedSR with BMC as the client TCP workload generation becomes the bottleneck.

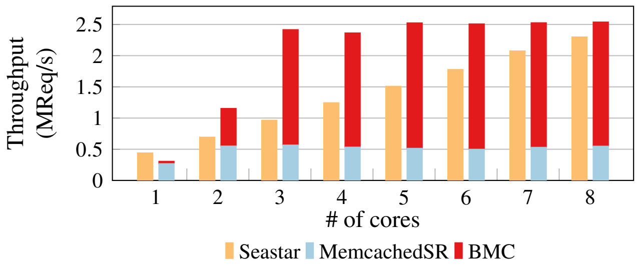


Figure 14: Seastar vs. BMC throughput when mixing TCP and UDP requests

CPU usage. We then measure the CPU usage of both MemcachedSR with BMC and Seastar for different client loads. In both configurations we use a total of 8 CPU cores to process the workload. For MemcachedSR with BMC, we use 6 RX cores and pin the Memcached threads to the two remaining cores. This configuration offers the best performance and allows us to measure the CPU usage of the Memcached application and the network stack (including BMC) separately. The CPU usage is measured on each core for 10 seconds using the *mpstat* tool. Figure 15 shows the average CPU core usage per core type (Seastar, MemcachedSR and BMC). The results show that Seastar always uses 100% of its CPU resources, even when throughput is low. This is because DPDK uses poll mode drivers to reduce the interrupt processing overhead when packets are received by the NIC. The CPU usage of MemcachedSR with BMC scales with the load thanks to the interrupt-based model of the native Linux drivers BMC builds upon. As shown in Figure 14, Seastar can process 2.3 million requests per second when using 8 cores, Figure 15 shows that MemcachedSR with BMC consumes 33% of the CPU (91% of the two Memcached cores and 13% of the six RX cores) to achieve similar throughput. Therefore, using Memcached with BMC saves CPU resources that can be used by other tasks running on the same system when the workload is low.

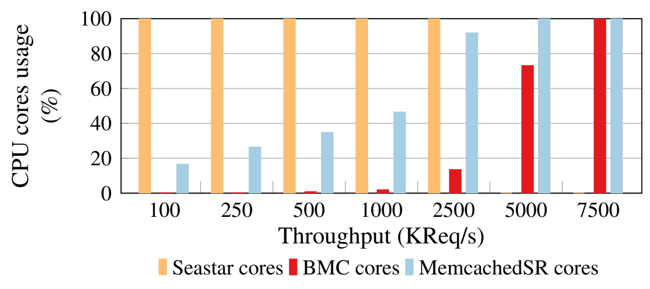


Figure 15: CPU usage of BMC compared to Seastar

6 Discussion

Although the BMC cache is fairly generic and can store any datatype, most of BMC is specialized to filtering and processing Memcached requests. Applying in-kernel caching to another key-value store like Redis [11] would then require specific eBPF programs to process Redis's RESP protocol. Because Redis requests are only transmitted over TCP, adapting BMC to Redis requires the support of the TCP protocol. This can be done by either sending acknowledgements from an eBPF program or reusing the existing TCP kernel implementation by intercepting packets past the TCP stack. As Redis is more focused on functionality and Memcached on performance, an in-kernel cache for Redis will require more eBPF programs to implement the minimal subset of Redis commands required to ensure cache coherence with the application. Just like BMC, some of the functionalities can be left to the application to focus on accelerating the most frequent request types. Redis usually performs worse than Memcached when processing a large volume of requests because it is single-threaded, hence we expect the throughput speed-up to be even higher than for Memcached.

Although static memory allocation enables the verification of BMC's eBPF programs, it also wastes memory. BMC suffers from internal fragmentation because each cache entry is statically bounded by the maximum data size it can store and inserting data smaller than this bound wastes kernel memory. The simplest approach to reduce memory fragmentation would be to fine-tune the bound of the cache entries to minimize the amount of fragmented memory. A more flexible approach would be to reuse fragmented memory to cache additional data. Each cache entry would then be able to store multiple data, making BMC a set-associative cache for which the number of slots varies according to the size of the stored data to reduce memory fragmentation. The fact that BMC is a non-exclusive cache also leads to memory loss since some data is duplicated between the BMC cache and Memcached. This duplication occurs for the majority of data that is not frequently accessed. On the other hand, frequently accessed data are eventually discarded from the application by Memcached's LRU algorithm because the BMC cache is used instead. Ideally, BMC should be able to directly access the application's cache memory to avoid any duplication, however, this requires browsing Memcached data structures which could create security vulnerabilities.

The XDP driver hook leveraged by BMC requires support in the NIC driver. With no XDP driver support, BMC can still be used with the generic Linux kernel hook but its performance will not be as high. However, this is not a critical concern as most of the drivers for high-speed network interfaces support XDP.

7 Related Work

This section discusses the most relevant related work in the field of optimization of the network stack and Memcached.

Programmable hardware switches. Recent advances in programmable hardware switches with languages like P4 [41] have raised significant interest on offloading network processing operations into the network. NetCache [29] implements an in-network key-value cache on Barefoot Tofino switches [12]. NetCache uses switch lookup tables to store, update and retrieve values. To access the key-value store, clients have to use a specific API to translate client requests into NetCache queries. Switch KV [32] and FlairKV [43] leverage programmable ASICs to implement a caching solution also acting as a load balancer. Switch KV uses an efficient routing algorithm to forward client queries to the right server. The OpenFlow protocol is used to install routes to cached objects and invalidate routes to recently modified ones. FlairKV accelerates GET queries by intercepting every SET query and the corresponding reply to detect unmodified objects. While these approaches leverage ASIC optimizations to offer high throughput and low latency, they consume switch memory, TCAM and SRAM, which is an expensive resource primary reserved for packet forwarding. Thus, using lookup table resources to store key-value data exposes the entire network to bottlenecks and failures.

FPGA and NIC offloading. Examples of key-value store applications offloaded to FPGA include TSSP [34] which implements part of the Memcached logic, i.e., the processing of GET requests over UDP, on a Xilinx Zynq SoC FPGA. A more complete FPGA Memcached implementation [17] supports processing of SET and GET requests over both TCP and UDP protocols. Similar work [18] deployed and evaluated an FPGA Memcached application within the public Amazon infrastructure. These approaches achieve high throughput with up to 13.2 million RPS with 10GbE link but they are constrained by the FPGA programming model, which requires replacing the Memcached protocol by more FPGA-compatible algorithms. KV-Direct [31] leverages Remote Direct Memory Access (RDMA) technology on NICs to update data directly on the host memory via PCIs. With this approach, KV-Direct alleviates CPU bottlenecks at the expense of PCI resources. NICA [22] introduces a new hardware-software co-designed framework to run application-level accelerators on FPGA NICs. NICA enables accelerating Memcached by serving GETs directly from hardware using a DRAM-resident cache and achieves similar performance to BMC since it still requires host processing to handle cache misses. NICached [42] proposes to use Finite State Machines as an abstract programming model to implement key-value store applications on programmable NICs. With this abstraction, NICached can be implemented with different languages and platforms: FPGA, eBPF, P4 and NPU-based NICs. NICached is the closest work to BMC but it targets NICs and does not propose an implementation of this model.

Compared to hardware approaches, BMC offers competitive performance, does not make use of expensive hardware resources such as SRAM and does not require hardware investment and software re-engineering.

Kernel-bypass. A kernel-bypass version of Memcached has been built on top of StackMap [46], an optimized network stack that achieves low latency and high throughput by dedicating hardware NICs to userspace applications. Using StackMap improves vanilla Memcached throughput by 2x in the most favorable scenario. MICA [33] employs a full kernel-bypass approach to process all key-value store queries in user space. MICA avoids synchronization by partitioning hash-maps among cores. MICA relies on a specific protocol that requires client information to map queries to specific cores and is not compatible with Memcached. To the best of our knowledge, MICA is the fastest software key value store application with a throughput of 77 million RPS on a dual-socket server with Intel Xeon E5-2680 processors. MICA is built with the DPDK library making MICA inherit most of DPDK's constraints: dedicated CPU cores to pull incoming packets, reliance on the hardware for isolation and requiring entirely re-engineering existing applications.

Compared to MICA, BMC achieves lower throughput but keeps the standard networking stack, does not implement any modification of clients and saves CPU resources.

Memcached optimizations Prior works [35, 44] have proposed to change the locking scheme of a former Memcached version to remove bottlenecks that impacted performance when running a large number of threads. To scale Memcached network I/O, MegaPipe [26] replaces socket I/O by a new channel-based API. Hippos [45] uses the Netfilter hook with a kernel module to serve Memcached requests from the Linux kernel, but does not ensure the safety of its kernel module and requires modifications to Memcached's source code to update its kernel cache.

eBPF verification. Gershuni et al. [25] have proposed a new verifier based on abstract interpretation in order to scale the verification of eBPF programs with loops. The authors showed that they could verify programs with small bounded loops and eliminate some false positives, however, their implementation has a time complexity about 100 times higher than the current Linux verifier, and uses from 100 to 1000 times more memory. Serval [39] introduces a general purpose and reusable approach to scale symbolic evaluation by using symbolic profiling and domain knowledge to provide symbolic optimizations. However, Serval does not consider domain specific symbolic evaluation. For example, the Linux verifier is capable of inferring register types based on the attach type of an eBPF program. Type inference then allows the Linux verifier to check the type correctness of the parameters passed to a helper function. Without this specific symbolic evaluation, Serval cannot ensure a precise analysis of eBPF programs and therefore cannot be used in place of the Linux verifier.

eBPF usage. eBPF is extensively used in industry for fast packet processing. Cilium [2] uses eBPF as a foundation to provide networking and security to Linux containers. Cloudflare uses eBPF to replace their complex infrastructure filtering rules by eBPF programs. As an example, Cloudflare's DDoS mitigation solution uses XDP in L4Drop, a module that transparently translates iptable DDoS mitigation rules into eBPF programs [8]. These eBPF programs are pushed to the edge servers located in Cloudflare's Points of Presence (PoPs) for automatic packet filtering. Facebook developed Katran [3], an XDP based L4 Load balancer. Katran consists of a C++ library and an XDP program deployed in backend servers in Facebook's infrastructure PoPs.

8 Conclusion

We present BMC, an in-kernel cache designed to improve performance of key-value store applications. BMC intercepts application queries at the lowest point of the network stack just as they come out of the NIC to offer high throughput and low latency with negligible overhead. When compared to user space alternatives, BMC shows comparable performance while saving computing resources. Moreover, BMC retains the Linux networking stack and works in concert with the user space application for serving complex operations. We believe that the BMC design can motivate the emergence of new system designs that make it possible to maintain the standard Linux networking stack while offering high performance.

BMC focuses on the optimization of Memcached because it is a performance-oriented key-value store. As a future work, we plan to apply the design of BMC to other popular key-value store applications such as Redis.

BMC is publicly available at `https://github.com/Orange-OpenSource/bmc-cache`.

Acknowledgments

We thank Julia Lawall, Paul Chaignon, Antonio Barbalace, and the anonymous reviewers for their valuable feedback.

References

[1] Amazon ElastiCache - In-memory data store and cache, 2020. URL: https://aws.amazon.com/elasticache.

[2] Cilium - Linux Native, API-Aware Networking and Security for Containers, 2020. URL: https://cilium.io.

[3] GitHub - facebookincubator/katran: A high performance layer 4 load balancer, 2020. URL: https://github.com/facebookincubator/katran.

[4] GitHub - scylladb/seastar: High performance server-side application framework, 2020. URL: https://github.com/scylladb/seastar.

[5] GitHub - twitter/twemcache: Twemcache is the Twitter Memcached, 2020. URL: https://github.com/twitter/twemcache.

[6] Home - DPDK, 2020. URL: https://www.dpdk.org/.

[7] kernel/bpf/syscall.c - Linux source code (v5.6) - Bootlin, 2020. URL: https://elixir.bootlin.com/linux/v5.6/source/kernel/bpf/syscall.c#L373.

[8] L4Drop: XDP DDoS Mitigations, 2020. URL: https://blog.cloudflare.com/l4drop-xdp-ebpf-based-ddos-mitigations.

[9] Linux Socket Filtering aka Berkeley Packet Filter (BPF) — The Linux Kernel documentation, 2020. URL: https://www.kernel.org/doc/html/latest/networking/filter.html#ebpf-verifier.

[10] memcached - a distributed memory object caching system, 2020. URL: https://memcached.org/.

[11] Redis, 2020. URL: https://redis.io/.

[12] Tofino Page - Barefoot Networks, 2020. URL: https://barefootnetworks.com/products/brief-tofino/.

[13] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '12, London, United Kingdom, June 11-15, 2012*, pages 53–64. ACM, 2012. doi:10.1145/2254756.2254766.

[14] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, pages 49–65. USENIX Association, 2014. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/belay.

[15] M. Ben-Yehuda, J. Mason, J. Xenidis, O. Krieger, L. Van Doorn, J. Nakajima, A. Mallick, and E. Wahlig. Utilizing IOMMUs for virtualization in Linux and Xen. In *OLS'06: The 2006 Ottawa Linux Symposium*, pages 71–86. Citeseer, 2006.

[16] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. T. Morris, and N. Zeldovich. An Analysis of Linux Scalability to Many Cores. In *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada*, pages 1–16. USENIX Association, 2010. URL: http://www.usenix.org/events/osdi10/tech/full_papers/Boyd-Wickizer.pdf.

[17] S. R. Chalamalasetti, K. T. Lim, M. Wright, A. AuYoung, P. Ranganathan, and M. Margala. An FPGA memcached appliance. In *The 2013 ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA '13, Monterey, CA, USA, February 11-13, 2013*, pages 245–254. ACM, 2013. doi:10.1145/2435264.2435306.

[18] J. Choi, R. Lian, Z. Li, A. Canis, and J. H. Anderson. Accelerating Memcached on AWS Cloud FPGAs. In *Proceedings of the 9th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies, HEART 2018, Toronto, ON, Canada, June 20-22, 2018*, pages 2:1–2:8. ACM, 2018. doi:10.1145/3241793.3241795.

[19] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, pages 143–154. ACM, 2010. doi:10.1145/1807128.1807152.

[20] Y. Dong, X. Yang, J. Li, G. Liao, K. Tian, and H. Guan. High performance network virtualization with SR-IOV. *J. Parallel Distributed Comput.*, 72(11):1471–1480, 2012. doi:10.1016/j.jpdc.2012.01.020.

[21] D. Eastlake, T. Hansen, G. Fowler, K.-P. Vo, and L. Noll. The FNV Non-Cryptographic Hash Algorithm. 2019.

[22] H. Eran, L. Zeno, M. Tork, G. Malka, and M. Silberstein. NICA: an infrastructure for inline acceleration of network applications. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, pages 345–362. USENIX Association, 2019. URL: https://www.usenix.org/conference/atc19/presentation/eran.

[23] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, pages 371–384. USENIX Association, 2013. URL: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/fan.

[24] B. Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[25] E. Gershuni, N. Amit, A. Gurfinkel, N. Narodytska, J. A. Navas, N. Rinetzky, L. Ryzhyk, and M. Sagiv. Simple and precise static analysis of untrusted Linux kernel extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 1069–1084. ACM, 2019. doi:10.1145/3314221.3314590.

[26] S. Han, S. Marshall, B. Chun, and S. Ratnasamy. MegaPipe: A New Programming Interface for Scalable Network I/O. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 135–148. USENIX Association, 2012. URL: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/han.

[27] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller. The eXpress data path: fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies, CoNEXT 2018, Heraklion, Greece, December 04-07, 2018*, pages 54–66. ACM, 2018. doi:10.1145/3281411.3281443.

[28] E. Jeong, S. Woo, M. A. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, pages 489–502. USENIX Association, 2014. URL: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/jeong.

[29] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 121–136. ACM, 2017. doi:10.1145/3132747.3132764.

[30] J. Leverich and C. Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014*, pages 4:1–4:14. ACM, 2014. doi:10.1145/2592798.2592821.

[31] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 137–152. ACM, 2017. doi:10.1145/3132747.3132756.

[32] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman. Be Fast, Cheap and in Control with SwitchKV. In *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016*, pages 31–44. USENIX Association, 2016. URL: https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/li-xiaozhou.

[33] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, pages 429–444. USENIX Association, 2014. URL: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/lim.

[34] K. T. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch. Thin servers with smart pipes: designing SoC accelerators for memcached. In *The 40th Annual International Symposium on Computer Architecture, ISCA'13, Tel-Aviv, Israel, June 23-27, 2013*, pages 36–47. ACM, 2013. doi:10.1145/2485922.2485926.

[35] J. Lozi, F. David, G. Thomas, J. L. Lawall, and G. Muller. Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications. In *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, pages 65–76. USENIX Association, 2012. URL: https://www.usenix.org/conference/atc12/technical-sessions/presentation/lozi.

[36] I. Marinos, R. N. M. Watson, and M. Handley. Network stack specialization for performance. In *ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014*, pages 175–186. ACM, 2014. doi:10.1145/2619239.2626311.

[37] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the Usenix Winter 1993 Technical*

Conference, San Diego, California, USA, January 1993, pages 259–270. USENIX Association, 1993. URL: https://www.usenix.org/conference/usenix-winter-1993-conference/bsd-packet-filter-new-architecture-user-level-packet.

[38] S. Miano, M. Bertrone, F. Risso, M. Tumolo, and M. V. Bernal. Creating Complex Network Services with eBPF: Experience and Lessons Learned. In *IEEE 19th International Conference on High Performance Switching and Routing, HPSR 2018, Bucharest, Romania, June 18-20, 2018*, pages 1–8. IEEE, 2018. doi:10.1109/HPSR.2018.8850758.

[39] L. Nelson, J. Bornholt, R. Gu, A. Baumann, E. Torlak, and X. Wang. Scaling symbolic evaluation for automated verification of systems code with Serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 225–242. ACM, 2019. doi:10.1145/3341301.3359641.

[40] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, pages 385–398. USENIX Association, 2013. URL: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala.

[41] B. Pat, D. Dan, I. Martin, M. Nick, R. Jennifer, T. Dan, V. Amin, V. George, and W. David. Programming Protocol-Independent Packet Processors. *ACM SIGCOMM Computer Communication Review*, 44, 12 2013. doi:10.1145/2656877.2656890.

[42] G. Siracusano and R. Bifulco. Is it a SmartNIC or a Key-Value Store?: Both! In *Posters and Demos Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*, pages 138–140. ACM, 2017. doi:10.1145/3123878.3132014.

[43] H. Takruri, I. Kettaneh, A. Alquraan, and S. Al-Kiswany. FLAIR: Accelerating Reads with Consistency-Aware Network Routing. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 723–737. USENIX Association, 2020. URL: https://www.usenix.org/conference/nsdi20/presentation/takruri.

[44] A. Wiggins and J. Langston. Enhancing the scalability of memcached. *Intel document, unpublished*, 2012.

[45] Y. Xu, E. Frachtenberg, and S. Jiang. Building a high-performance key-value cache as an energy-efficient appliance. *Perform. Evaluation*, 79:24–37, 2014. doi:10.1016/j.peva.2014.07.002.

[46] K. Yasukata, M. Honda, D. Santry, and L. Eggert. StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*, pages 43–56. USENIX Association, 2016. URL: https://www.usenix.org/conference/atc16/technical-sessions/presentation/yasukata.

Segcache: a memory-efficient and scalable in-memory key-value cache for small objects

Juncheng Yang
Carnegie Mellon University

Yao Yue
Twitter

K. V. Rashmi
Carnegie Mellon University

Abstract

Modern web applications heavily rely on in-memory key-value caches to deliver low-latency, high-throughput services. In-memory caches store small objects of size in the range of 10s to 1000s of bytes, and use TTLs widely for data freshness and implicit delete. Current solutions have relatively large per-object metadata and cannot remove expired objects promptly without incurring a high overhead. We present Segcache, which uses a segment-structured design that stores data in fixed-size segments with three key features: (1) it groups objects with similar creation and expiration time into the segments for efficient expiration and eviction, (2) it approximates some and lifts most per-object metadata into the shared segment header and shared information slot in the hash table for object metadata reduction, and (3) it performs segment-level bulk expiration and eviction with tiny critical sections for high scalability. Evaluation using production traces shows that Segcache uses 22-60% less memory than state-of-the-art designs for a variety of workloads. Segcache simultaneously delivers high throughput, up to 40% better than Memcached on a single thread. It exhibits close-to-linear scalability, providing a close to 8× speedup over Memcached with 24 threads.

1 Introduction

In-memory caches such as Memcached [54] and Redis [13] are widely used in modern web services such as Twitter [18, 46], Facebook [21, 55], Reddit [3] to reduce service latency and improve system scalability. The economy of cache lies within supporting data retrieval more cheaply, and usually more quickly, compared to the alternatives. The usefulness of in-memory caches is judged by their efficiency, throughput, and scalability, given certain hardware resource constraints. Memory efficiency determines the amount of memory a cache needs to achieve a certain miss ratio. Throughput is typically measured in queries per second (QPS) per CPU core. Scalability reflects how well a cache can use multiple cores on a host. There have been several efforts to reduce miss ratio via better eviction algorithms [22, 26, 36, 37]. Many other works focus on improving throughput [41, 51]. However, several other aspects of in-memory caching also play import roles in memory efficiency.

Web services tend to cache small key-value objects in memory, typically in the range of 10s to 1000s of bytes [21, 46].

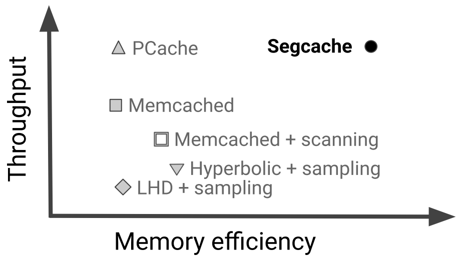


Figure 1: How Segcache compares to state-of-the-art caches

However, most popular production caching systems store a relatively large amount of metadata. For example, both Memcached and Redis impose over 50 bytes of memory overhead per object. Furthermore, research aimed at reducing miss ratio typically ends up expanding object metadata even further [22, 26, 28, 36, 60], as shown in Table 1.

Time-to-live (TTL) is widely used in caching to meet data freshness and feature requirements, or comply with regulations such as GDPR [42, 65–67]. Twitter mandates the use of TTLs in cache, with values ranging from a few minutes to a month. Existing caching systems either remove expired objects lazily or incur high overhead [70] when they attempt to expire more proactively. We summarize the techniques for removing expired objects in Table. 2 and discuss them in §2.

Most production in-memory caches use an external memory allocator such as `malloc` or a slab-based memory allocator. The former often subjects the cache to external fragmentation, and the latter to internal fragmentation [63]. In addition, slab-based allocators often suffer from the so-called slab calcification problem [18, 44], or introduce extra cache misses due to slab migration [29, 34].

One way to reduce memory fragmentation is to adopt a log-structured design. This approach has been widely used for file systems [62] and durable key-value stores [14, 31, 33, 57, 58, 63] for their simplicity, high write throughput [51, 62], low fragmentation, and excellent space efficiency [63]. There have been several in-memory caches built with log-structured design. MICA [51], a throughput-oriented system based on one giant log per thread, limits its eviction algorithm to FIFO or CLOCK, both of which are sub-optimal for many

Table 1: Comparison of research systems (all comparisons are with corresponding baselines)

System	Memory Allocator	Memory fragmentation	Improve TTL expiration	Object metadata size	Throughput	Memory efficiency improvement approach
MICA	Log	No	No	Decrease	Higher	Worse
Memshare	Log	No	No	Increase	Lower	Memory partitioning and sharing
pRedis	Malloc	External	No	Increase	Lower	Better eviction
Hyperbolic	Malloc	External	No	Increase	Lower	Better eviction
LHD	Slab	Internal	No	Increase	Lower	Better eviction
MemC3	Slab	Internal	No	Decrease	Higher	Small metadata
Segcache	Segment	No	Yes	Minimal	Higher	Holistic redesign

workloads. Memshare [37], a multi-tenant caching system, divides DRAM into small logs (called segments) and uses segments to enforce memory partitioning between tenants. However, its computation of miss ratio curve for each tenant and object migration are relatively expensive, which result in reduced throughput compared to Memcached.

As modern servers become denser with CPU cores over time, thread-scalability becomes essential in modern cache design. Several techniques have been proposed to improve scalability in key-value caches and key-value stores, such as static DRAM and data partitioning [50, 51], opportunistic concurrency control with lock-free data structures [32, 41, 52], and epoch-based design [31]. However, each technique comes with its own problems. Static partitioning uses memory inefficiently. Opportunistic concurrency control works better on read-heavy workloads, whereas some caching workloads are write-heavy [46]. An epoch-based system requires a log-structured design with a sub-optimal eviction algorithm.

Achieving high memory efficiency, high throughput, and high scalability simultaneously in caching systems is challenging. Previous works tend to trade one for the other (Fig. 1 and Table 1). In this paper, we present Segcache, a cache design that achieves all the three desired properties. Segcache is a TTL-indexed, dynamically-partitioned, segment-structured[1] cache where objects of similar TTLs are stored in a small fixed-size log called a segment. Segments are first grouped by TTL and then naturally sorted by creation time. This design makes timely removal of expired objects both simple and cheap. As a cache, Segcache performs eviction by merging a few segments into one, retaining only the most important objects, and freeing the rest. Managing the object life cycles at the segment level allows most metadata to be shared within a segment. It also allows metadata bookkeeping to be performed with a limited number of tiny critical sections. These decisions improve memory efficiency and scalability without sacrificing throughput or features.

Below are some highlights of our contributions:

- To the best of our knowledge, Segcache is the first cache design that can efficiently remove all objects immediately after expiration. This is achieved through TTL-indexed, time-sorted segment chains.

[1]Since segments are small-sized logs, Segcache can be viewed as a log-structured cache with special properties; see §6 for in-depth comparisons.

Table 2: Techniques for removing expired objects

Technique	Remove all expired?	Is removal cheap?
Deletion on access	No	Yes
Checking LRU tail	No	Yes
Transient item pool	No	Yes
Full cache scan	Yes	No
Random sampling	No	No

- We propose and demonstrate "object sharing economy", a concept that reduces per-object metadata to just 5 bytes per object, a 91% reduction compared to Memcached, without compromising on features.
- Our single pass, merge-based eviction algorithm uses an approximate and smoothed frequency counter to achieve a balance between retaining high value objects and effectively reclaiming memory.
- We demonstrate that a "macro management strategy", replacing per-request bookkeeping with batched operations on segments, improves throughput. It also delivers close-to-linear CPU scalability.
- Segcache is designed for production on top of Pelikan, and is open sourced (see §4).
- We evaluated Segcache using a wide variety of production traces, and compared results with multiple state-of-the-art designs. Segcache reduces memory footprint by 42-88% compared to Twitter's production system, and 22-58% compared to the best of state-of-the-art designs.

2 Background and Motivation

As a critical component of the real-time serving infrastructure, caches prefer to store data, especially small objects, in DRAM. DRAM is expensive and energy-hungry. However, existing systems do not use the costly DRAM space efficiently. This inefficiency mainly comes from three places. First, existing solutions are not able to quickly remove expired objects. Second, metadata overhead is considerable compared to typical object sizes. Third, internal or external memory fragmentation is common, leading to wasted space. While improvements of admission [24, 25, 39, 40, 43], prefetching [73, 74], and eviction algorithms [22, 23, 26, 30, 36, 37, 49, 68] have been the main focus of existing works on improving memory efficiency [22, 25, 26, 39], little attention has been paid to addressing expiration and metadata reduction [41, 63]. On the contrary, many systems add more per-object metadata to make smarter decisions about what objects to keep [22, 36, 39, 73].

We summarize recent advancements of in-memory caching systems in Table 1 and discuss them below.

2.1 TTL and expiration in caching

TTLs are extremely common in caching. As a result, object expiration is an integral part of all existing solutions.

2.1.1 The prevalence of TTL

TTLs are used by users of Memcached and Redis [4, 8, 10, 12], Facebook [17], Reddit [3], Netflix [6]. In Twitter's production, *all* in-memory cache workloads use TTLs between one minute and one month. A TTL is specified at write time to determine how long an object should remain accessible in the caching system. An expired object cannot be returned to the client, and a cache miss is served instead.

Cache TTLs serve three purposes. First, clients use TTLs to limit data inconsistency [4, 46]. Writing to the cache is usually best-effort, so it is not uncommon for data in cache and database to fall out of sync. Second, some services use TTLs to prompt periodic re-computation. For example, a recommendation system may only want to reuse cached results within a time window, and recompute periodically to incorporate new activities and content. Third, TTLs are used for implicit deletion. A typical scenario is rate-limiting. Rate limiters are counters associated with some identities. Services often need to cap requests from a particular identify within a predefined time window to prevent denial-of-service attacks. Services store rate limiters in distributed caches with TTLs, so that the counts can be shared among stateless services and reset periodically. Another increasingly common scenario is using TTLs to ensure data in caches comply with privacy laws [42, 67].

2.1.2 Lazy expiration

Lazy expiration means expiration only happens when an object is reaccessed. *Deletion on access* is the most straightforward approach adopted by many production caching systems. If a system uses lazy expiration only, an object that's no longer accessed can remain in memory long past expiration.

2.1.3 Proactive expiration

Proactive expiration is used to reclaim memory occupied by expired objects more quickly. Although there has been no academic research on this topic to the best of our knowledge, we identified four approaches introduced into production systems over the years, as summarized in Table. 2.

Checking LRU tail is used by Memcached. Before eviction is considered, the system checks a fixed number of objects at the tail of the LRU queue and removes expired objects. Operations on object LRU queues reduce thread scalability due to the extensive use of locking for concurrent accesses [22, 26]. Additionally, this approach is still opportunistic and therefore doesn't guarantee the timely removal of expired objects. Many production caches track billions of objects over a few LRU queues, so the time for an object to percolate through the LRU queue is very long.

Transient object pool was introduced by Facebook [55]. It makes a special case for the timely removal of objects with small TTLs. The main idea is to store such objects separately, and only allow them to be removed via expiration. However, choosing the TTL threshold is non-trivial and can have side effects [46]. Although Memcached supports it, it is disabled by default.

Full cache scan is a popular approach adopted by Memcached and CacheLib [17]. As the name indicates, this solution periodically scans all the cached objects to remove expired ones. Full cache scan is very effective if the scan is frequent, but it wastes resources on objects that are not expired, which can be the vast majority.

Random sampling is adopted by Redis. The key idea is to periodically sample a subset of objects and remove expired ones. In Redis, if the percentage of the expired objects in the sample is above a threshold, this process continues. While sampling is cheaper per run, the blind nature of sampling decides that it is both inefficient and not very effective. Users have to accept that the sampling can only keep the percentage of expired objects at a pre-configured threshold. Meanwhile, the cost can be higher than full cache scan due to random memory access. There have been some production incidents where Redis could not remove enough expired objects and caused unexpected evictions [70].

Despite the various flaws, proactive expiration is highly regarded by developers of production systems. When asked to replace LRU for a better eviction strategy in Memcached, the maintainer states that "*pulling expired items out actively is better than almost any other algorithmic improvement (on eviction) I could think of.*" [10] Meanwhile, Redis' author mentioned that "*Redis 6 expiration will no longer be based on random sampling but will take keys sorted by expiration time in a radix tree.*" [2]

In summary, efficiently and effectively removing expired objects is an urgent problem that needs to be solved in current caching systems.

2.2 Object metadata

We observe that the objects stored in in-memory caches are small [46], and the mean object sizes (key+value) of Twitter's top four production clusters are 230, 55, 294, 72 bytes, respectively. This observation aligns with the observations at Facebook [21], and Reddit [3].

Existing systems are not efficient in storing small objects because they store considerable amount of metadata per object. For example, Memcached stores *56 bytes* of metadata with each object [3], which is a significant overhead compared

[2]As of Redis v6.0.6, this change is not implemented yet.

[3]2×8 bytes LRU pointers, 8 bytes hash pointer, 4 bytes access time, 4 bytes expire time, 8 bytes object size, 8 bytes `cas` (compare-and-set, used for atomic update).

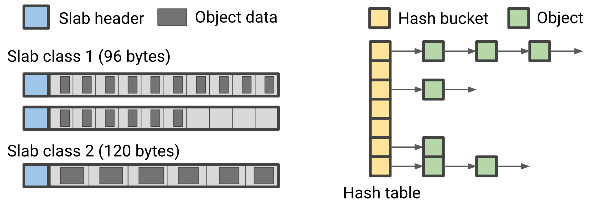


Figure 2: Slab memory allocation (left) and object-chained hash table (right) in Memcached.

to typical object size. All of the metadata fields are critical for Memcached's operations, and cannot be dropped without first removing some functionalities or features.

There have been several attempts at Twitter to cut metadata overhead. For example, Pelikan's slab-based storage removes object LRU queues and reuses one pointer for both hash chain and free object chain. As a result, it reduces object metadata to 38 bytes. However, this prevents Pelikan from applying the LRU algorithm to object eviction, and results in higher miss ratio compared to Memcached in our evaluation. Pelikan also introduced Cuckoo hashing [59] as a storage module for fixed-size objects, only storing 6 bytes (or 14 bytes with `cas`) of metadata per key.

Several academia works have also looked at reducing metadata size. RAMCloud [63] and FASTER [31] use a log-structured design to reduce object metadata. However, their designs target *key-value stores* instead of key-value caches (See discussion in §5). MemC3 [41] redesigns the hash table with Cuckoo hashing and removes LRU chain pointers. However, it does not consider some operations such as `cas` for atomic updates, does not support TTL expiration or other advanced eviction algorithms.

2.3 Memory fragmentation

Memory management is one of the fundamental design aspects of an in-memory caching system. Systems that directly use external memory allocators (e.g., *malloc*) such as Redis are vulnerable to external memory fragmentation and OOM.

To avoid this problem, other systems such as Memcached use a slab-based memory allocator, allotting a fixed-size slab at a time, which is then explicitly partitioned into smaller chunks for storing objects, as shown in Fig. 2 (left). The chunk size is decided by the `class id` of a slab and configured during startup. A slab-based memory allocator is subjected to internal memory fragmentation at the end of each chunk and at the end of each slab.

Using a slab-based allocator also introduces the slab calcification problem, a phenomenon where some slab classes cannot obtain enough memory and exhibit higher miss ratios. Slab calcification happens because slabs are assigned to classes using the first-come-first-serve method. When popularity among slab classes change over time, the newly popular slab classes cannot secure more memory because all slabs have been assigned. This has been studied in the previous works [29, 44, 46]. Memcached automatically migrates slabs between classes to solve this problem, however, it is not always effective [2, 5, 9, 15]. Re-balancing slabs may increase the miss ratio because all objects on the outgoing slab are evicted. Moreover, due to workload diversity and complexity in slab migration, it is prone to errors and sometimes causes crash in production [7, 16].

Overall, existing production systems have not yet entirely solved the memory fragmentation problem. Among the research systems, log-structured designs such as MICA [50, 51], memshare [37] and RAMCloud [63] do not have this problem. However, they cannot perform proactive expiration and are limited to using basic eviction algorithms (such as FIFO or CLOCK) with low memory efficiency.

2.4 Throughput and scalability

In addition to memory efficiency, throughput and thread-scalability are also critical for in-memory key-value caches. Memcached's scalability limitation is well documented in various industry benchmarks [11, 55]. The root cause is generally attributed to the extensive locking in the object LRU queues, free object queues, and the hash table. Several systems have been proposed to solve this problem. Some of them remove locking by using simpler eviction algorithms and sacrificing memory efficiency [41, 51]. Some introduces opportunistic concurrency control [41], which does not work well with write-heavy workloads. Some other works use random eviction algorithms to avoid concurrent reads and writes [22, 26], which do not address all the locking contention. Moreover, they reduce throughput due to the large number of random memory accesses.

3 Design principles and overview

The design of Segcache follows three principles.

Be proactive, don't be lazy. Expired objects offer no value, so Segcache eagerly removes them for memory efficiency.
Maximize metadata sharing for economy. To reduce the metadata overhead without loss of functionality, Segcache maximizes metadata sharing across objects.
Perform macro management. Segcache operates on segments to expire/evict objects in bulk with minimum locking.

At a high level, Segcache contains three components: a hash table for object lookup, an object store comprised of segments, and a TTL-indexed bucket array (Fig. 3).

3.1 TTL buckets

Indexing on TTL facilitates efficient removal of expired objects. To achieve this, Segcache first breaks the spectrum of possible TTLs into ranges. We define the time-width of a TTL range t_1 to t_2 ($t_1 < t_2$) as $t_2 - t_1$. All objects in range t_1 to t_2 are treated as having TTL t_1, which is the *approximate TTL* of this range. Rounding down guarantees an object can only be expired early, and no object will be served beyond expiration. Objects are grouped into small fix-sized groups called segments (see next section), and all the objects stored in the same segment have the same approximate TTL. Second, Segcache

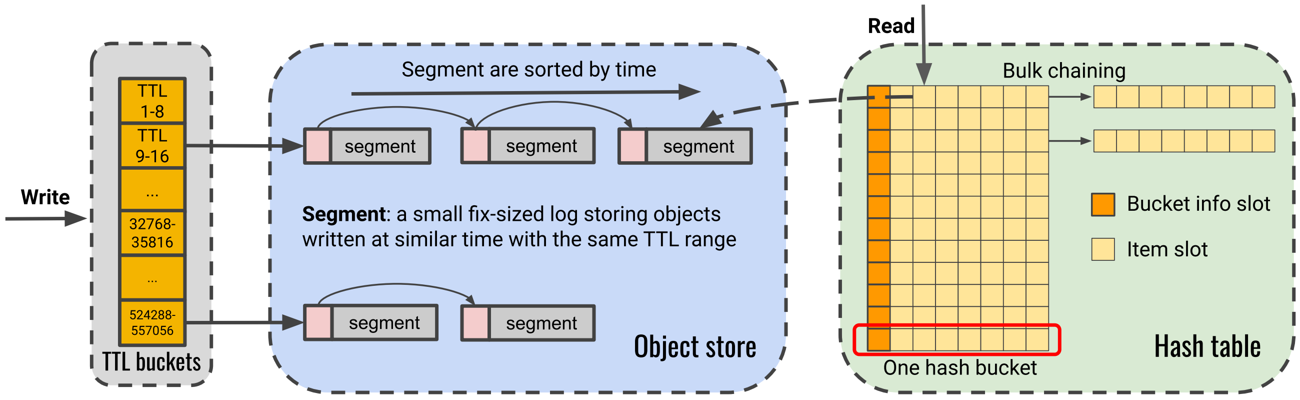


Figure 3: Overview of Segcache. A read request starts from the hash table (right), a write request starts from the TTL buckets (left).

uses an array to index segments based on *approximate TTL*. Each element in this array is called a *TTL bucket*. A segment with a particular approximate TTL value is associated with the corresponding TTL bucket. Within each bucket, segments are chained and sorted by creation time.

To support a wide TTL range from a few second to at least one month without introducing too many buckets or losing resolution on the lower end, Segcache uses 1024 TTL buckets, divided into four groups. From one group to the next, the time-width grows by a factor of 16. In other words, Segcache uses increasingly coarser buckets to efficiently cover a wide range of TTLs without losing relative precision for typical TTL buckets. The boundaries of the TTL buckets are chosen in a way that finding the TTL bucket only requires a few bit-wise operations. We show that this design allows Segcache to efficiently and effectively remove expired objects in §3.5.

3.2 Object store: segments

Segcache uses segments as the basic building blocks for storing objects. All segments are of a configurable size, default to 1 MB. Unlike slabs in Memcached, Segcache group objects stored in the same segment by approximate TTL, not by size. A segment in Segcache is similar to a small log in log-structured systems. Objects are always appended to the end of a segment, and once written, the objects cannot be updated (except for `incr`/`decr` atomic operations). However, unlike other log-structured systems [37, 51, 57, 58, 62, 63], where available DRAM is either used as one continuous log or as segments withou no relationship between each other, segments in Segcache are sorted by creation time, linked into chains, and indexed by approximate TTLs.

In Segcache, each non-empty TTL bucket stores pointers to the head and tail of a time-sorted segment chain, with the head segment being the oldest. A write in Segcache first finds the right TTL bucket for the object, and then appends to the segment at the tail of the segment chain. When the tail segment is full, a new segment is allocated. If there is no free segment available, eviction is triggered (§3.6).

3.3 Hash table

As shown in previous works [41], the object-chained hash tables (Fig. 2 (right)) limits the throughput and scalability in the existing production systems [54,71]. Segcache uses a bulk-chaining hash table similar to MICA [51] and Faster [31].

An object-chained hash table uses object chaining to resolve hash collisions. The throughput of such a design is sensitive to hash table load. Collision resolution requires walking down the hash chain, incurring multiple random DRAM accesses and string comparisons. Moreover, object chaining imposes a memory overhead of an 8-byte hash pointer per object, which is expensive compared to the small object sizes.

Instead of having just one slot per hash bucket, Segcache allocates 64 bytes of memory (one CPU cache line) as eight slots in each hash bucket (Fig. 3). The first slot stores the bucket information, the following six slots store object information. The last slot stores either object information or a pointer to the next hash bucket (when more than seven objects hash to the same bucket). This chaining of hash buckets is called bulk chaining. Bulk chaining removes the need to store hash pointers in the object metadata and improves the throughput of hash lookup by minimizing random accesses.

The bucket information slot stores an 8-bit spin lock, an 8-bit slot usage counter, a 16-bit last-access timestamp , and a 32-bit `cas` value. Each item slot stores a 24-bit segment id, a 20-bit offset in the segment, an 8-bit frequency counter (described in §3.6.3), and a 12-bit tag. The tag of a key is a hash used to reduce the number of string comparisons when hash collisions happen.

3.4 Object metadata

Segcache achieves low metadata overhead by *sharing metadata across objects*. Segcache facilitates metadata sharing at two places: the hash table bucket and the segment. Objects in the same segment share creation time, TTL, reference counter, while objects in the same hash bucket share last-access timestamp, spinlock, `cas` value, and hash pointer.

Because objects in the same segment have the same approximate TTL and are written around the same time, Segcache computes the approximate expiration time of the whole segment based on the oldest object in the segment and approximate TTL of the TTL bucket. This approximation skews the clock and incurs early expiration for objects later in the segment. As we will show in our evaluation, early expiration has negligible impact on miss ratio.

Segcache also omits object-level hash chain pointers and LRU chain pointers. Bulk chaining renders hash chain pointer unnecessary. The LRU chain pointers are not needed because because both expiration and eviction are performed at the segment level. Segcache further moves up metadata needed for concurrent accesses (reference counter) into the segment header. In addition, to support `cas`, Segcache maintains a 32-bit `cas` value per hash bucket and shares it between all objects in the hash bucket. While sharing this value may increase false data race between different objects hashed to the same hash bucket, in practice, the impact of this compromise is negligible due to two reasons. First, `cas` traffic is usually orders of magnitude lower than simple read or write, as observed in production environment [46]. Second, one `cas` value is shared only by a few keys, the chance of concurrent updates on different keys in the same hash bucket is small. In the case of a false data race, the client usually retries the request.

The final composition of object metadata in Segcache contains one 8-bit key size, one 24-bit value size, and one 8-bit flag. And Segcache stores only 5 bytes[4] of metadata with each object, which is a 91% reduction compared to Memcached.

3.5 Proactive expiration

In Segcache, all objects in one segment are written sequentially and have the same approximate TTL, which makes it feasible to remove expired objects in bulk. Proactive removal of expired objects starts with scanning the TTL buckets. Because segments linked in each TTL bucket are ordered by creation time and share the same approximate TTL, they are also ordered by expiration time. Segcache uses a background thread to scan the first segment's header in each non-empty TTL buckets. If the first segment is expired, the background thread removes all the objects in the segment, then continues down the chain until it runs into one segment that is not yet expired, at which point it will move onto the next TTL bucket.

Segcache's proactive expiration technique uses memory bandwidth efficiently. Other than reading the expired objects, each full scan only accesses *a small amount of consecutive metadata* — the TTL bucket array. This technique also ensures that memory occupied by expired objects are promptly and completely recycled, which improves memory efficiency.

As mentioned before, objects are subject to early expiration. However, objects are usually less useful near the end of their TTL. Our analysis of production traces at Twitter shows that a small TTL reduction makes negligible difference (if any) in the miss ratio.

[4]The 5-byte does not include the shared metadata, which is small per object. And it also does not include the one-byte frequency counter, which is stored as part of object pointer in the hash table.

3.6 Segment eviction

While expiration removes objects that cannot be used in the future and is preferred over eviction, cache cannot rely on expiration alone. All caching systems support eviction when necessary to make room for new objects.

Eviction decisions can affect the effectiveness of cache in terms of the miss ratio, thus have been the main focus of many previous works [22, 26, 35, 49, 56, 69]. Segcache does not update objects in-place. Instead, it appends new objects and marks the old ones as deleted. Therefore, better eviction becomes even more critical.

Unlike most existing systems performing evictions by object, Segcache performs eviction by segments. Segment eviction could evict popular objects, increasing the miss ratio. To address this problem, Segcache uses a *merge-based* eviction algorithm. The basic idea is that by combining multiple segments into one, Segcache selectively retains a relatively small portion of the objects that are more likely to be accessed again and discards the rest. This design brings out several finer design decisions. First, we need to pick the segments to be merged. Second, there needs to be an algorithm making per-object decisions while going through these segments.

3.6.1 Segment selection

The segments merged during each eviction are always from a single TTL bucket. Within this bucket, Segcache merges the first N *consecutive*, *un-expired*, and *un-merged* (in current iteration) segments (Fig. 4). The new segment created from the eviction inherits the creation time of the oldest evicted segment. This design has the following benefits. First, the created segment can be inserted in the same position as the evicted segments in the segment chain, and maintains the time-sorted segment chain property. Second, objects in the created segment still have relatively close creation/expiration time, and the merge distorts their expiration schedules minimally.

While within one TTL bucket, the segment selection is limited to consecutive ones, across TTL buckets, Segcache uses round-robin to choose TTL bucket.

3.6.2 One-pass merge and segment homogeneity

When merging N consecutive segments into one, Segcache uses a dynamic threshold for retaining objects to achieve merge in a single pass. This threshold is updated after scanning every $\frac{1}{10}$ of a segment and aims to retain $\frac{1}{N}$ bytes from each segment being evicted.

The rationale for retaining a similar number of bytes from each segment is that objects and segments created at a similar time are homogeneous with similar properties. Therefore, no segment is more important than others. Fig. 5a shows the relative standard deviation (RSD, $\frac{std}{mean}$) of the mean object size in consecutive segments and across random segments, and Fig. 5b compares the RSD of live bytes in consecutive and random segments. Both figures demonstrate that consecutive segments are more homogeneous (similar) than random segments. As a result, retaining a similar number of bytes from

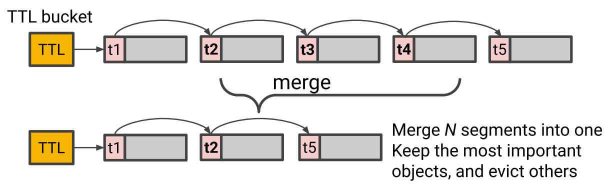


Figure 4: Merge-based segment eviction.

each is reasonable. However, we remark that the current segment selection and merge heuristics may not be the optimal solution in some cases, and deserve more exploration.

3.6.3 Selecting objects

So far, one question remains unsolved: what objects should be retained in an eviction? An eviction algorithm's effectiveness is determined by its ability to predict future access based on past information. Under the independent reference model (IRM), a popular model used for cache workloads, an object with a higher frequency is more likely to be re-accessed. Moreover, it has been shown in theory that under IRM and for fix-sized objects, the least frequently used (LFU) is k-competitive and the best policy [27, 39, 61, 64].

Similar to greedy dual size frequency [35], Segcache uses the frequency-over-size ratio to rank objects. Therefore it needs a frequency counter that is memory-efficient, computationally-cheap, and scalable. Meanwhile, it should allow Segcache to be burst-resistant and scan-resistant. Moreover, The counter needs to provide *higher accuracy for less popular* objects (opposite of the counter-min sketch). This is critical for cache eviction because the highly-popular objects are always retained (cached), and the less popular objects decide the miss ratio of a cache. Segcache uses a novel one-byte counter (stored in hash table), which we call *approximate and smoothed frequency counter* (ASFC), to track frequencies.

Approximate counter. ASFC has two stages. When frequency is smaller than 16 (last four bits of the counter), it always increases by one for every request. In the second stage, it counts frequency similar to a Morris counter [1], which increases with a probability that is the inverse of current value.

Smoothed counter. Segcache uses the last access timestamp, which is shared by objects in the same hash bucket, to rate-limit updates to the frequency counters. The frequency counter for each object is incremented at most once per second. This technique is effective in absorbing sudden request bursts.

Simple LFU is susceptible to cache pollution due to request bursts and non-constant data access patterns. While several approaches such as dynamic aging [20, 39, 61], and window-based frequency [38, 47] have been proposed to address this issue, they require additional parameters and/or extensive tuning [19]. To avoid extra parameters, Segcache resets the frequency of retained objects during evictions, which has a similar effect as window-based frequency.

The linear increase at low frequency and probabilistic increase at high frequency allow ASFC to achieve a higher accuracy for less popular objects. Meanwhile, the approximate design allows ASFC to be memory efficient, using one byte to count up to $2^8 \times 2^8$ requests. The smoothed design of ASFC allows Segcache to be burst-resistant and scalable.

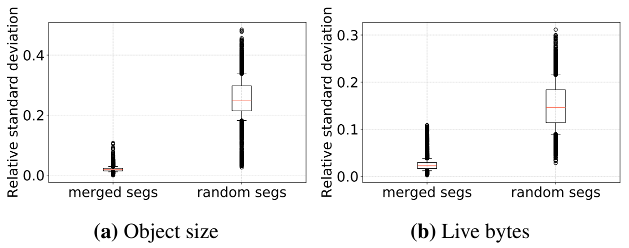


Figure 5: a) Relative standard deviation of mean object size in consecutive segments and random segments. b) Relative standard deviation of live bytes in consecutive segments and random segments.

3.7 Thread model and scalability

Segcache is designed to scale linearly with the number of threads by using a combination of techniques such as minimal critical sections, optimistic concurrency control, atomic operations, and thread-local variables. Most notably, because object life cycle management is at the segment level, only modifications to the segment chains require locking, which avoids common contention spots related to object-level bookkeeping, such as maintaining free-object queues. This macro management strategy reduces locking frequency by four orders of magnitude in our default setting compared to what would be needed in a Memcached-like system.

More specifically, no locking is needed on the read path except to increment object frequency, which is at most once every second. On the write path, because segments are append-only, inserting objects can take advantage of atomic operations. However, we observe that relying on atomic operation is insufficient to achieve near-linear scalability with more than eight threads. To solve this, each thread in Segcache maintains a local view of active segments (the last segment of each segment chain), and the active segments in each thread can be written only by that thread. Although the segments are local to each thread for writes, the objects that have been written are immediately available for reading by other threads. During eviction, locking is required when segments are being removed from the segment chain. However, the critical section of removing a segment from the chain is very tiny compared to object removal, which is *lock-free*. Moreover, evicting one segment means evicting thousands of objects, so segment eviction is infrequent compared to object writes.

4 Implementation and Evaluation

In this section, we compare the memory efficiency, throughput, and scalability of Segcache against several research and production solutions, using traces from Twitter's production. Specifically, we are interested in the following questions,

- Is Segcache more memory efficient than alternatives?

Table 3: Traces used in evaluation

Trace	Workload type	# requests	TTLs (TTL: percentage)	Write ratio	Mean object size	Production miss ratio
c	content	4.2 billion	1d: 65%, 14d: 27%, 12h: 7%	7%	230 bytes	1-5%
*u*1	user	6.5 billion	5d:1.00	1%	290 bytes	<1%
*u*2	user	4.5 billion	12h:1.00	3%	55 bytes	<1%
n	negative cache	1.6 billion	30d:1.00	2%	45 bytes	~1%
mix	content + user + negative cache + transient item	11.88 billion	30d: 14%, 14d:11%, 24h: 23%, 12h: 38%, 2min:12%	7%	243 bytes	NA

- Does Segcache provide comparable throughput to state-of-the-art solutions? Does it scale well with more cores?
- Is Segcache sensitive to design parameters? Are they easy to pick or tune?

4.1 Implementation

Segcache is implemented as a storage module in the open-sourced Pelikan project. Pelikan is a cache framework developed at Twitter. The Segcache module can both work as a library or be setup as a Memcached-like server. Our current implementation supports multiple worker threads, with a dedicated background thread performing proactive expiration. For our evaluation, eviction is performed by worker threads as-needed, but it is easy to use the same background thread to facilitate background eviction. We provide configurable options to change the number of segments to merge for eviction and segment size. The source code can be accessed at http://www.github.com/twitter/pelikan and archived at http://www.github.com/thesys-lab/segcache.

4.2 Experiment setup

4.2.1 Traces

Single tenant traces. We used week-long unsampled traces from production cache clusters at Twitter (Table. 3, the same as in previous work [46])[5]. Trace *c* comes from a cache storing tweets and their metadata, which is the largest cache cluster at Twitter. Trace *u*1 and *u*2 are both user related, but the access patterns of the two workloads are different, so different TTLs are used. Notably, they are separated into two caches in production because effective and efficient proactive expiration was not achievable prior to Segcache. Trace *n* is a negative result cache, which stores the keys that do not exist in the database, a common way of using cache to shield databases from unnecessary high loads.

Multi tenant trace. Although Twitter's production deployments are single-tenant, multi-tenant deployments are also common because of better resource utilization [21]. To evaluate the performance under multi-tenant workloads, we merged workloads from four types of caches: user, content, negative cache, and transient item cache.

4.2.2 Baselines

Memcached used in our evaluation is version 1.6.6 with segmented LRUs. It supports lazy expiration and checks LRU tail for expiration. We ran Memcached in two modes, one with cache scanning enabled (s-Memcached), which scans the entire cache periodically to remove expired objects; the other with scanning disabled (Memcached). Other expiration techniques are enabled in both modes. Our evaluation also includes pelikan_twemcache (PCache), Twitter's Memcached equivalent and successor to Twemcache [18]. Compared to Memcached, PCache has a smaller object metadata without LRU queues, and only performs slab eviction [75]. We implemented LHD [22] and Hyperbolic [26] on top of PCache since original implementations are not publicly available. These systems do not consider object expiration. To make the comparisons fairer, we add random sampling to remove expired objects in these two systems, which is also how Redis performs expiration. In the following sections, r-LHD and r-Hyperbolic refer to these enhanced versions. Note that adding random sampling to remove expired objects does not significantly impact the throughput, and we observe less than a 10% difference.

Because we do not modify the networking stack, we focus our evaluation on the storage subsystem. We performed all evaluations by close-loop trace replay on dedicated hosts in Twitter's production fleet using the traces described in §4.2.1. The hosts have dual-socket Intel Xeon Gold 6230R CPU, 384 GB DRAM with one 100 Gbps NIC.

4.2.3 Metrics

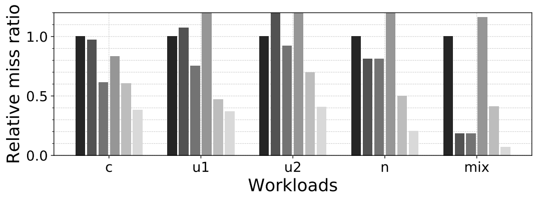
We use three metrics in our evaluation to measure the memory efficiency, throughput, and scalability of the systems.

Relative miss ratio. Miss ratio is the most common metric in evaluating memory efficiency. Because workloads have dramatically different miss ratios in production (from a few percent to less than 0.1%) and compulsory miss ratios, directly plotting miss ratio is less readable. Therefore, we use *relative miss ratio* (defined as $\frac{mr}{mr_{baseline}}$ where *mr* stands for miss ratio and the baseline is PCache) in the presentation.

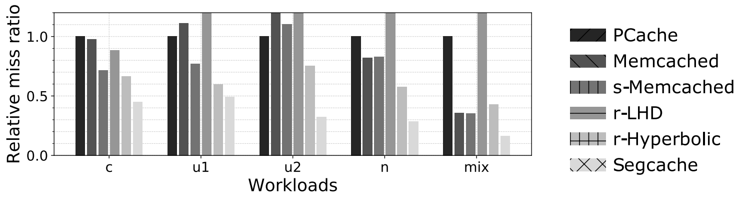
Relative memory footprint. Although miss ratio is a common metric, a sometimes more useful metric is how much memory footprint can be reduced at a certain miss ratio. Therefore, in §4.3, we show this metric using PCache memory footprint as the baseline.

Throughput and scalability. Throughput is measured in million queries per second (MQPS) and used to quantify a caching system's performance. Scalability measures the throughput running on a multi-core machine with the number of hardware threads from 1 to 24 in our evaluations.

[5]The traces are available at http://www.github.com/twitter/cache-trace.

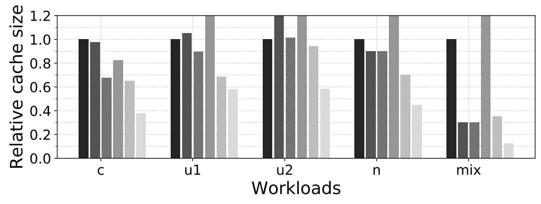


(a) Large cache size

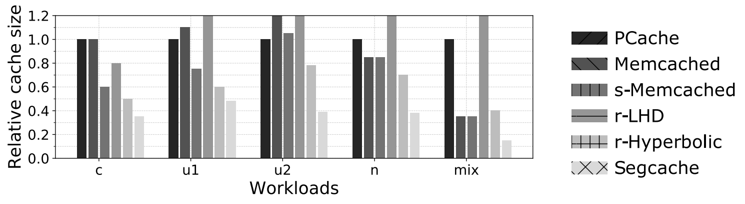


(b) Small cache size

Figure 6: Relative miss ratio of different systems (baseline Pelikan is 1), lower is better.



(a) Production miss ratio



(b) Miss ratio of Pelikan at the small cache size in Fig.6b

Figure 7: Relative memory footprint to achieve a certain miss ratio, lower is better.

4.3 Memory efficiency

In this section, we compare the memory efficiency of all systems. We present the relative miss ratio at two cache sizes[6] in Fig.6. (1) The "large cache" is the cache size when the miss ratio of Segcache reaches the plateau (<0.05% miss ratio reduction when the cache size increases by 5%). Miss ratios achieved at large cache sizes are similar to production miss ratios. (2) we choose the "small cache" size as 50% of the large cache size.

Compared to the best of the five alternative systems, Segcache reduces miss ratios by up to 58%. Moreover, it performs better on both the single-tenant and the multi-tenant workloads. This large improvement is the cumulative effect of having timely proactive expiration, small object metadata, no memory fragmentation, and a merge-based eviction strategy.

We observe that Memcached and PCache have comparable miss ratios in most workloads (except workload *mix* because PCache is not designed for multi-tenant workloads). While comparing Memcached and s-Memcached, we observe that adding full cache scanning capability significantly reduces the miss ratio by up to 40%, which indicates the importance of proactive expiration. However, as we show in §4.4.1, cache scanning is expensive and reduces throughput by almost half for some workloads. Moreover, we observe that workload *n* and *mix* do not benefit from full cache scanning. Workload *n* shows no benefit because it uses a single TTL of 30 days and no objects expire in the evaluation. Although workload *mix* has a mixture of short and long TTLs, it shows no benefit because the objects of different TTLs are from different workloads with different object sizes, and are stored in different slab classes with different LRU queues. As a result, checking LRU tail for expiration is effective at removing expired objects and scanning provides little benefit. Overall, we observe that proactively removing expired objects can effectively reduce miss ratio and improve memory efficiency.

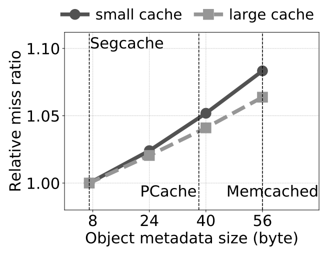
[6]We experimented with twenty cache sizes, and the two set of results presented here are representative.

State-of-the-art research caching systems, r-LHD and r-Hyperbolic use ranking to select eviction candidates and often reduce miss ratio compared to LRU. In our evaluation, r-Hyperbolic shows lower miss ratio compared to Memcached and PCache, while r-LHD is only better on workload *c*. r-LHD is designed for workloads with a mixture of scan and LRU access patterns (such as block access in storage systems), while in-memory caching workloads rarely show scan requests. This explains why it has higher miss ratios. We have also evaluated r-LHD and r-Hyperbolic without sampling for expiration (not shown), and as expected, they have higher miss ratios due to the wasted cache space from expired objects.

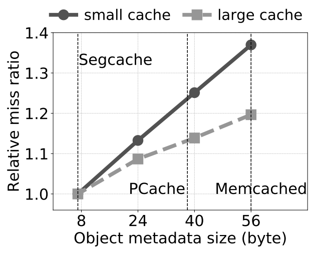
An alternative way of looking at memory efficiency is to determine the cache size required to achieve a certain miss ratio. We show the relative memory footprints of different systems in Fig. 7, using PCache as the baseline. The figures show that for both the production miss ratio and a higher miss ratio, Segcache reduces memory footprint by up to 88% compared to PCache, 60% compared to Memcached, 56% compared to s-Memcached, and 64% compared to r-Hyperbolic.

4.3.1 Ablation study

In Fig. 6, we observe that s-Memcached reduces miss ratio by up to 35% compared to Memcached, which demonstrates the importance of proactive expiration, one of the key design features of Segcache. Besides proactive expiration, another advantage of Segcache over previous systems is smaller object metadata. To understand its impact, we measure the relative miss ratio of increasing object metadata in Segcache (Fig. 8). It shows that reducing object metadata size can have a large miss raito impact for workloads with small object sizes. Workload *c* has relatively large object sizes (230 bytes), and reduc-



(a) Workload *c*



(b) Workload *n*

Figure 8: Impact of object metadata size on miss ratio. Workload *n* has smaller object sizes as compared to workload *c* and hence enjoys larger benefit from reduction from object metadata.

ing the metadata from 56 bytes to 8 bytes reduces the miss ratio by 6-8%. While workload *n* has small object sizes (45 bytes) and reducing object metadata size provides a 20-38% reduction in miss ratio. This result indicates reducing object metadata size is very important, and it is a critical component contributing to Segcache's high memory efficiency.

4.4 Throughput and scalability

4.4.1 Single-thread throughput

Besides memory efficiency, the other important metric of a cache is the throughput. Fig. 9 shows the throughput of different systems. Compared to other systems, PCache and Segcache achieve higher throughput, up to 2.5× faster than s-Memcached, up to 3× faster than r-Hyperbolic, and up to 4× faster than r-LHD. The reason is that PCache performs slab eviction only, and Segcache performs merge-based segment eviction. Both systems perform batched and sequential bookkeeping for evictions, which significantly reduces the number of random memory accesses and makes good use of the CPU cache. In addition, PCache and Segcache do not maintain an object LRU chain, which leads to less bookkeeping and also contributes to the high throughput.

Although r-LHD and r-Hyperbolic have lower miss ratios than Memcached, their throughput is also lower. The reason is that both systems use random sampling during evictions, which causes a large number of random memory accesses. One major bottleneck of a high-throughput cache is the poor CPU cache hit ratio, and optimizing CPU cache utilization has been one focus of improving the throughput [51, 52]. Although r-LHD proposes to segregate object metadata for better locality [22], it requires adding more object metadata, and hence would further decrease memory efficiency.

4.4.2 Thread scalability

We show the scalability results in Fig. 10a, where we compare Segcache with Memcached and s-Memcached. Fig. 10a shows that compared to Memcached, Segcache has a higher throughput and close-to-linear scalability. With 24 threads, Segcache achieves over 70 MQPS, a 19.9× boost compared to using a single-thread, while Memcached only achieves 9 MQPS, 3.4× of its single-thread throughput. The reason why Segcache can achieve close-to-linear scalability is the effect of multiple factors as discussed in §3.7. While there is not much throughput difference between Memcached and s-Memcached, s-Memcached is deadlocked when running with more than 8 threads.

Note that we do not present the result of PCache in this figure because it does not support multi-threading. We also do not show the result of r-LHD and r-Hyperbolic because we could not find any simple way to implement a better locking than the one in Memcached. Although r-LHD and r-Hyperbolic removes the object LRU chain and lock, the slab memory allocator still requires heavy locking.

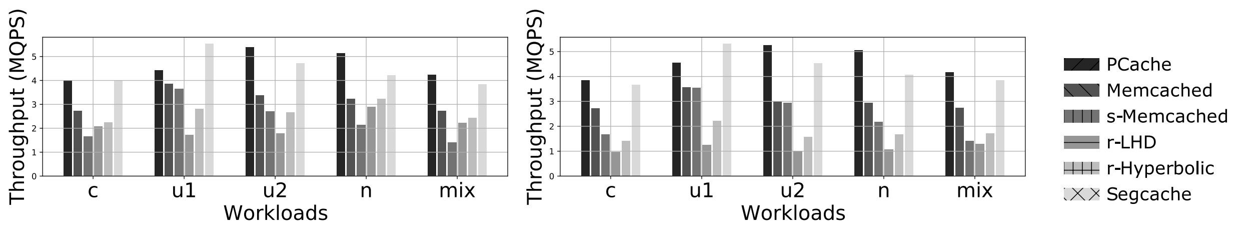
4.5 Sensitivity

In this section, we study the effects of parameters in Segcache using workload *c* (from Twitter's largest cache cluster). The most crucial parameter in Segcache is the number of segments to merge for eviction, which balances between processing overhead and memory efficiency. Fig. 10b shows how the miss ratio is affected by the number of merged segments. Compared to retaining no objects (the bar labeled eviction), using merge-based eviction reduces the miss ratio by up to 20%, indicating the effectiveness of merge-based eviction. Moreover, it shows that the point for the minimal miss ratio is between 3 and 4. Merging two segments or more than four segments increases the miss ratio, but not significantly.

There are two reasons why merging too few segments leads to a high miss ratio. First, merging too few segments can lead to unfilled segment space. For example, when merging only two segments, 50% of the bytes are retained from each segment in one pass. If the second segment does not have enough live objects, the new segment will have space wasted. Second, the fidelity of predicting future accesses on unpopular objects is low. Merging fewer segments means retaining more objects, so it requires distinguishing unpopular objects, and the decision can be inaccurate. Meanwhile, merging fewer segments means triggering eviction more frequently, giving objects less time to accumulate hits.

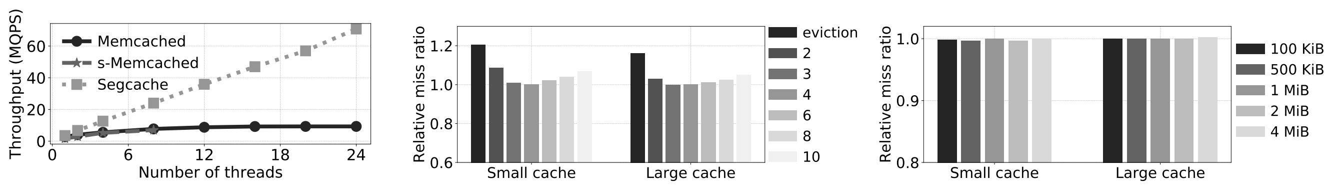
On the other hand, merging too many segments increases the miss ratio as well. Because merging more segments means setting a higher bar for retained objects, some important objects can be evicted. In our evaluation, we observe three and four are, in general, good options. However, merging more or fewer segments does not adversely affect the miss ratio significantly and still provides a lower miss ratio than current production systems. Therefore, we consider this parameter a stable one that does not require tuning per workload.

Besides the number of segments to merge, another parameter in Segcache is the segment size. We use the default 1 MB in our evaluation; Fig. 10c shows the impact of different segment sizes. It demonstrates that segment size has little impact on the miss ratio, which is expected. Because the fraction of objects retained from each segment does not depend on the segment size, thus not affecting the miss ratio.



(a) Large cache size **(b)** Small cache size

Figure 9: Throughput of different systems, the higher the better.



(a) Scalability **(b)** Number of segments to merge **(c)** Segment size

Figure 10: CPU scalability and sensitivity analysis.

5 Discussion

5.1 Alternative proactive expiration designs

Besides the TTL bucket design in Segcache, there are other possible solutions for proactive expiration. For example, a radix tree or a hierarchical timing wheel can track object expiration time. However, neither is as memory efficient as Segcache. In fact, any design that builds an expiration index strictly at the object level requires two pointers per object, an overhead with demonstrated impact for our target workloads. The radix tree may also use an unbounded amount of memory to store the large and uncertain number of expiration timestamps. In addition, performing object-level expiration and eviction requires more random memory access and locking than bulk operations, limiting throughput and scalability.

5.2 In-memory key-value cache vs store

In the literature, we observe several instances where there is a mix-up of *volatile key-value caches* (such as Memcached) and *durable key-value stores* (such as RAMCloud and RocksDB). However, from our viewpoint, these two types of systems are significantly different in terms of their usage, requirements, and design. Indeed, one of the main contributions is to identify the opportunity to approximate object metadata and share them (time, pointers, reference counters, version/cas number) across objects. Time approximation in particular is not as tolerated in a traditional key-value store. Below we discuss the differences between caches and stores.

TTL. TTLs are far more ubiquitous in caching than in key-value store [3, 6, 12, 17, 55, 70]. We described Twitter's use of TTLs in detail in [46]. In comparison, many datasets are kept in key-value stores indefinitely.

Eviction. Eviction is unique to caching. In addition, eviction is extremely common in caching. A production cache running at 1M QPS with 10% writes, which can be new objects or on-demand fill from cache misses, will evict 100K objects every second. Re-purposing compaction and cleaning techniques in log-structured storage may not be able to keep up with the write rate needed in caching. On the other hand, caches have considerable latitude in deciding what to store, and can choose more efficient mechanisms.

Design requirements. In-memory caches are often used in front of key-value stores to absorb most read requests, or to store transient data with high write rates. Production users expect caches to deliver much higher throughput and/or much lower tail latencies. In contrast, key-value stores are often considered sources of truth. As such they prioritize durability (crash recovery) and consistency over latency and throughput.

The differences between cache and store allow us to make some design choices in Segcache that are not feasible for durable key-value stores (even if they are in-memory).

6 Related work

6.1 Memory efficiency and throughput

Approaches for improving memory efficiency fall broadly in the two categories: improving eviction algorithms and adding admission algorithms.

Eviction algorithm. A vast number of eviction algorithms have been proposed in different areas starting from the early 90s [45, 53, 56, 61, 76]. However, most of them focus on the cache replacement of databases or page cache, which are different from a distributed in-memory cache because cached contents in databases and page cache are typically fix-sized blocks with spatial locality. In recent years, several algorithms have been proposed to improve the efficiency of in-memory caching, such as LHD [22], Hyperbolic caching [26], pRedis [60], and mPart [28]. However, all of them add more object metadata and computation, which reduces usable cache size and reduces throughput, which has significant repercussions for caches with small objects.

Segcache uses a merge-based eviction strategy that retains high-frequency small-sized objects from evicted segments, which is similar to a frequency-based eviction algorithms such as LFU [20,47] and GDSF [35]. However, unlike some of these systems that require parameters tuning, Segcache uses ASFC that avoids these problems. In addition to the eviction algorithm, two major components that contribute to Segcache's low miss ratio is efficient, proactive expiration, and object metadata sharing, which are unique to Segcache.

Admission control. Adding admission control to decide which object should be inserted into the cache is a popular approach for improving efficiency. For example, Adaptsize [25], W-TinyLFU [39], flashshield [40] are designed in the recent years. Admission control is effective for CDN caches, which usually have high one-hit-wonder ratios (up to 30%) with a wide range of object sizes (100s of bytes to 10s of GB). Segcache does not employ an admission algorithm because most of the in-memory cache workloads have low one-hit-wonder ratios (<5%) and relatively small object size ranges. Moreover, adding admission control often add more metadata and extra computation, hurting efficiency and throughput.

Other approaches. There are several other approaches in improving efficiency, such as optimizing slab migration strategy in Memcached [29,44], compressing cached data [72], and prefetching data [73]. Reducing object metadata size has also been considered in previous works [41]. However, for supporting the same set of functions (including expirations, deletions, `cas`), these approaches need more than twice as much object metadata as Segcache.

Throughput and scalability. A large fraction of works on improving throughput and scalability focus on durable key-value stores [31,50,52], which are different from key-value caches as discussed in §5. Segcache is inspired by these works and further improves throughput and scalability by macro management using approximate and shared object metadata.

6.2 Log-structured designs

Segcache's segment-structured design is inspired by several existing works that employ log-structured design [31, 33, 51, 57, 58, 62, 63] in storage and caching systems. The log-structured design has been widely adopted in storage systems to reduce random access and improve throughput. For example, log-structured file system [62] and LSM-tree databases [14,48] transform random disk writes to sequential writes. Recently log-structured designs have also been adopted in in-memory key-value store [31,33,57,58,63] to improve both throughput and scalability.

For in-memory caching, MICA [51] uses DRAM as one big log to improve throughput, but it uses FIFO for eviction and does not optimize for TTL expiration. Memshare [37] also uses log-structured design and has the concept of segments. However, Memshare optimizes for multi-tenant cache by moving cache space between tenants to minimize miss ratio based on each tenant's miss ratio curve. Memshare uses a cleaning process to scan N segments, evict one segment, and keep $N-1$ segments where the goal is to enforce memory partitioning between tenants. In terms of performance, scanning N ($N = 100$ in evaluation) segments and *evicting one* incurs a high computation overhead and negatively affects the throughout. Moreover, to compute the miss ratio of different tenants, Memshare adds more metadata to the system, which reduces memory efficiency.

Systems employing a log-structured design benefit from reduced metadata size and memory fragmentation, and increased write throughput, for example, several of the existing works [14,41,63] and including Segcache. Compared to these existing works, Segcache achieves a higher memory efficiency by *approximating* and *sharing* object metadata, *proactive TTL expiration*, and using ASFC to retain fewer bytes during eviction while providing a low miss ratio (10% - 25% bytes from each segment are retained in Segcache compared to 75% in RAMCloud [63] and 99% in Memshare [37]).

In a broad view, Segcache can be described as a *dynamically-partitioned* and *approximate-TTL-indexed* log-structured cache. However, one of the key differences between Segcache and log-structured design is that Segcache is centered around the indexed and sorted segment chain. Both objects in a segment and segments in the chains are time-sorted and indexed by approximate TTLs for metadata sharing, macro management, and efficient TTL expiration.

7 Conclusion

Segcache stems out of our insights from production workloads, in particular, the observation that object expiration and metadata play an important role in improving memory efficiency. We chose a TTL-indexed segment-structured design to achieve both high throughput, high scalability and memory efficiency. Our evaluation against state-of-the-art designs from both research and production projects shows that Segcache comes out ahead on our stated goals. Its efficient use of memory bandwidth, near linear scalability, and low-touch configuration poise it favorably as a practical production caching solution suitable for contemporary and future hardware.

Acknowledgements. We thank our shepherd Ryan Stutsman and the anonymous reviewers for their valuable feedback. We also thank Rebecca Isaacs, Xi Wang and Dan Luu for providing feedback, and the Cache, IOP and HWEng teams at Twitter for their support in evaluating Segcache. This work was supported in part by a Facebook PhD fellowship, and in part by NSF grants CNS 1901410 and CNS 1956271.

References

[1] Approximate counting algorithm. https://en.wikipedia.org/wiki/Approximate_counting_algorithm. Accessed: 2020-08-06.

[2] bit vector + backoff timer + simpler implementation for faster slab reassignment. https://github.com/memcached/memcached/pull/542. Accessed: 2020-08-06.

[3] Caching at reddit. https://redditblog.com/2017/1/17/caching-at-reddit/. Accessed: 2020-05-06.

[4] database caching strategy using redis. https://d0.awsstatic.com/whitepapers/Database/database-caching-strategies-using-redis.pdf. Accessed: 2020-05-06.

[5] Enhance slab reallocation for burst of eviction. https://github.com/memcached/memcached/pull/695. Accessed: 2020-08-06.

[6] Ephemeral volatile caching in the cloud. https://netflixtechblog.com/ephemeral-volatile-caching-in-the-cloud-8eba7b124589. Accessed: 2020-05-06.

[7] Experiencing slab ooms after one week of uptime. https://github.com/memcached/memcached/issue/689. Accessed: 2020-08-06.

[8] Expiration in redis 6. https://news.ycombinator.com/item?id=19664483. Accessed: 2020-08-06.

[9] Faster slab reassignment. https://github.com/memcached/memcached/pull/524. Accessed: 2020-08-06.

[10] Lfu eviction policy. https://github.com/memcached/memcached/issues/543. Accessed: 2020-08-06.

[11] Memcached benchmark. https://github.com/scylladb/seastar/wiki/Memcached-Benchmark. Accessed: 2020-08-06.

[12] Memory used only grows despite unlink/delete of keys. https://github.com/redis/redis/issues/7482. Accessed: 2020-08-06.

[13] Redis. http://redis.io/. Accessed: 2020-05-06.

[14] Rocksdb. https://rocksdb.org/. Accessed: 2020-08-06.

[15] slab auto-mover anti-favours slab 2. https://github.com/memcached/memcached/issue/677. Accessed: 2020-08-06.

[16] slabs: fix crash in page mover. https://github.com/memcached/memcached/pull/608. Accessed: 2020-08-06.

[17] Cachelib: The general-purpose caching engine. 2020.

[18] Chris Aniszczyk. Caching with twemcache. https://blog.twitter.com/engineering/en_us/a/2012/caching-with-twemcache.html. Accessed: 2020-08-06.

[19] Martin Arlitt, Ludmila Cherkasova, John Dilley, Rich Friedrich, and Tai Jin. Evaluating content management techniques for web proxy caches. *ACM SIGMETRICS Performance Evaluation Review*, 27(4):3–11, 2000.

[20] Martin Arlitt, Rich Friedrich, and Tai Jin. Performance evaluation of web proxy cache replacement policies. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 193–206. Springer, 1998.

[21] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, pages 53–64, 2012.

[22] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. Lhd : Improving cache hit rate by maximizing hit density. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 389–403, 2018.

[23] Nathan Beckmann and Daniel Sanchez. Talus: A simple way to remove cliffs in cache performance. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 64–75. IEEE, 2015.

[24] Daniel S. Berger. Towards lightweight and robust machine learning for cdn caching. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, HotNets '18, page 134–140, New York, NY, USA, 2018. Association for Computing Machinery.

[25] Daniel S Berger, Ramesh K Sitaraman, and Mor Harchol-Balter. Adaptsize: Orchestrating the hot object memory cache in a content delivery network. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 483–498, 2017.

[26] Aaron Blankstein, Siddhartha Sen, and Michael J Freedman. Hyperbolic caching: Flexible caching for web applications. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 499–511, 2017.

[27] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and zipf-like distributions: Evidence and implications. In *IEEE INFOCOM'99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now*

(Cat. No. 99CH36320), volume 1, pages 126–134. IEEE, 1999.

[28] Daniel Byrne, Nilufer Onder, and Zhenlin Wang. mpart: miss-ratio curve guided partitioning in key-value stores. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on Memory Management*, pages 84–95, 2018.

[29] Daniel Byrne, Nilufer Onder, and Zhenlin Wang. Faster slab reassignment in memcached. In *Proceedings of the International Symposium on Memory Systems*, pages 353–362, 2019.

[30] Pei Cao and Sandy Irani. Cost-aware www proxy caching algorithms. In *Usenix symposium on internet technologies and systems*, volume 12, pages 193–206, 1997.

[31] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. Faster: A concurrent key-value store with in-place updates. In *Proceedings of the 2018 International Conference on Management of Data*, pages 275–290, 2018.

[32] Jiqiang Chen, Liang Chen, Sheng Wang, Guoyun Zhu, Yuanyuan Sun, Huan Liu, and Feifei Li. Hotring: A hotspot-aware in-memory key-value store. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 239–252, Santa Clara, CA, February 2020. USENIX Association.

[33] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. Flatstore: An efficient log-structured key-value storage engine for persistent memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1077–1091, New York, NY, USA, 2020. Association for Computing Machinery.

[34] Yue Cheng, Aayush Gupta, and Ali R. Butt. An in-memory object caching framework with adaptive load balancing. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery.

[35] Ludmila Cherkasova. *Improving WWW proxies performance with greedy-dual-size-frequency caching policy*. Hewlett-Packard Laboratories, 1998.

[36] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 379–392, 2016.

[37] Asaf Cidon, Daniel Rushton, Stephen M Rumble, and Ryan Stutsman. Memshare: a dynamic multi-tenant key-value cache. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 321–334, 2017.

[38] Gil Einziger, Ohad Eytan, Roy Friedman, and Ben Manes. Adaptive software cache management. In *Proceedings of the 19th International Middleware Conference*, pages 94–106, 2018.

[39] Gil Einziger, Roy Friedman, and Ben Manes. Tinylfu: A highly efficient cache admission policy. *ACM Transactions on Storage (ToS)*, 13(4):1–31, 2017.

[40] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. Flashield: a hybrid key-value cache that controls flash write amplification. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 65–78, 2019.

[41] Bin Fan, David G Andersen, and Michael Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 371–384, 2013.

[42] gdpr. Art. 17 gdpr right to erasure ('right to be forgotten'). https://gdpr-info.eu/art-17-gdpr/. Accessed: 2020-05-06.

[43] Yu Guan, Xinggong Zhang, and Zongming Guo. Caca: Learning-based content-aware cache admission for video content in edge caching. In *Proceedings of the 27th ACM International Conference on Multimedia*, MM '19, page 456–464, New York, NY, USA, 2019. Association for Computing Machinery.

[44] Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo, Chen Ding, Song Jiang, and Zhenlin Wang. {LAMA}: Optimized locality-aware memory allocation for key-value cache. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 57–69, 2015.

[45] Song Jiang, Feng Chen, and Xiaodong Zhang. Clock-pro: An effective improvement of the clock replacement. In *USENIX Annual Technical Conference, General Track*, pages 323–336, 2005.

[46] Rashmi Vinayak Juncheng Yang, Yao Yue. A large scale analysis of hundreds of in-memory caching clusters at twitter. In *OSDI'20*, 2020.

[47] George Karakostas and D Serpanos. Practical lfu implementation for web caching. *Technical Report TR-622–00*, 2000.

[48] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[49] Conglong Li and Alan L Cox. Gd-wheel: a cost-aware replacement policy for key-value stores. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–15, 2015.

[50] Sheng Li, Hyeontaek Lim, Victor W Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G Andersen, O Seongil, Sukhan Lee, and Pradeep Dubey. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 476–488, 2015.

[51] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. Mica : A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, 2014.

[52] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 183–196, 2012.

[53] Nimrod Megiddo and Dharmendra S Modha. Arc: A self-tuning, low overhead replacement cache. In *Fast*, volume 3, pages 115–130, 2003.

[54] memcached. memcached - a distributed memory object caching system. http://memcached.org/. Accessed: 2020-05-06.

[55] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, 2013.

[56] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. *Acm Sigmod Record*, 22(2):297–306, 1993.

[57] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, et al. The case for ramclouds: scalable high-performance storage entirely in dram. *ACM SIGOPS Operating Systems Review*, 43(4):92–105, 2010.

[58] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, et al. The ramcloud storage system. *ACM Transactions on Computer Systems (TOCS)*, 33(3):1–55, 2015.

[59] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[60] Cheng Pan, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. predis: Penalty and locality aware memory allocation in redis. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 193–205, 2019.

[61] John T Robinson and Murthy V Devarakonda. Data cache management using frequency-based replacement. In *Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 134–142, 1990.

[62] Mendel Rosenblum and John K Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[63] Stephen M Rumble, Ankita Kejriwal, and John Ousterhout. Log-structured memory for dram-based storage. In *12th {USENIX} Conference on File and Storage Technologies ({FAST} 14)*, pages 1–16, 2014.

[64] Dimitrios N Serpanos and Wayne H Wolf. Caching web objects using zipf's law. In *Multimedia Storage and Archiving Systems III*, volume 3527, pages 320–326. International Society for Optics and Photonics, 1998.

[65] Aashaka Shah, Vinay Banakar, Supreeth Shastri, Melissa Wasserman, and Vijay Chidambaram. Analyzing the impact of GDPR on storage systems. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, Renton, WA, July 2019. USENIX Association.

[66] Supreeth Shastri, Melissa Wasserman, and Vijay Chidambaram. The seven sins of personal-data processing systems under GDPR. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association.

[67] Supreeth Shastri, Melissa Wasserman, and Vijay Chidambaram. Gdpr anti-patterns. *Commun. ACM*, 64(2):59–65, January 2021.

[68] Zhenyu Song, Daniel S Berger, Kai Li, and Wyatt Lloyd. Learning relaxed belady for content distribution network caching. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 529–544, 2020.

[69] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. Ripq : Advanced photo caching on flash for facebook. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 373–386, 2015.

[70] Matthew Tejo. Improving key expiration in redis. https://blog.twitter.com/engineering/en_us/topics/infrastructure/2019/improving-key-expiration-in-redis.html. Accessed: 2020-08-06.

[71] Twitter. twitter twemcache. https://github.com/twitter/twemcache. Accessed: 2020-05-06.

[72] Xingbo Wu, Li Zhang, Yandong Wang, Yufei Ren, Michel Hack, and Song Jiang. zexpander: A key-value cache with both high performance and fewer misses. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–15, 2016.

[73] Juncheng Yang, Reza Karimi, Trausti Sæmundsson, Avani Wildani, and Ymir Vigfusson. Mithril: mining sporadic associations for cache prefetching. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 66–79, 2017.

[74] Qiang Yang, Haining Henry Zhang, and Tianyi Li. Mining web logs for prediction models in www caching and prefetching. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 473–478, 2001.

[75] Yao Yue. Eviction strategies. https://github.com/twitter/twemcache/wiki/Eviction-Strategies. Accessed: 2020-08-06.

[76] Yuanyuan Zhou, Zhifeng Chen, and Kai Li. Second-level buffer cache management. *IEEE Transactions on parallel and distributed systems*, 15(6):505–519, 2004.

When Cloud Storage Meets RDMA

Yixiao Gao♠♡, Qiang Li♡, Lingbo Tang♡, Yongqing Xi♡, Pengcheng Zhang♡, Wenwen Peng♡, Bo Li♡, Yaohui Wu♡, Shaozong Liu♡, Lei Yan♡, Fei Feng♡, Yan Zhuang♡, Fan Liu♡, Pan Liu♡, Xingkui Liu♡, Zhongjie Wu♡, Junping Wu♡, Zheng Cao♡, Chen Tian♠, Jinbo Wu♡, Jiaji Zhu♡, Haiyong Wang♡, Dennis Cai♡, and Jiesheng Wu♡

♠Nanjing University, ♡Alibaba Group

Abstract

A production-level cloud storage system must be high performing and readily available. It should also meet a Service-Level Agreement (SLA). The rapid advancement in storage media has left networking lagging behind, resulting in a major performance bottleneck for new cloud storage generations. Remote Direct Memory Access (RDMA) running on lossless fabrics can potentially overcome this bottleneck. In this paper, we present our experience in introducing RDMA into the storage networks of Pangu, a cloud storage system developed by Alibaba. Since its introduction in 2009, it has proven to be crucial for Alibaba's core businesses. In addition to the performance, availability, and SLA requirements, the deployment planning of Pangu at the production scale should consider storage volume and hardware costs. We present an RDMA-enabled Pangu system that exhibits superior performance, with the availability and SLA standards matching those of traditional TCP-backed versions. RDMA-enabled Pangu has been demonstrated to successfully serve numerous online mission-critical services across four years, including several important shopping festivals.

1 Introduction

Alibaba Group [12] is a China-based multinational technology company specializing in e-commerce, e-finance, and cloud computing. Numerous companies, including Alibaba, have moved their core business systems onto clouds. As a fundamental part of information technology (IT) infrastructure, a cloud storage provides a storage service to tenants both inside and outside the cloud provider. In 2009, Alibaba introduced Pangu [18], a cloud storage system that has subsequently played a crucial role in many Alibaba core businesses. As of 2020, Pangu has been deployed in hundreds of clusters, and it has been managing hundreds of thousands of storage nodes. Furthermore, it supports the real-time access to exabyte-level data in numerous production environments.

In order to ensure comparability to local physical storage clusters, a cloud storage system must meet the following requirements:

(i) *High performance:* Small latency and high throughput provide competitive advantages across many scenarios.

(ii) *High availability:* System disruptions incur significant financial/reputation loss for both tenants and their cloud providers.

(iii) *Service-Level Agreement (SLA):* A cloud storage system must be resilient, and thus its performance should gracefully downgrade when various software/hardware failures happen.

The rapid advancement in storage media has left networking lagging behind, resulting in a major performance bottleneck for new cloud storage generations. Networking is not a problem for traditional storage systems built with Hard Disk Drives (HDDs). However, the access latency of current Non-Volatile Memory Express (NVMe) disks is at the microsecond level [50] and the total throughput of a storage node can exceed 100Gbps. In contrast, the latency of traditional network stacks (*e.g.*, TCP/IP) can reach milliseconds [13], while the bandwidth per kernel TCP thread is only tens of Gbps at most [51].

Remote Direct Memory Access (RDMA) running on lossless fabrics offers a promising solution to the network bottleneck in cloud storage. By implementing its entire protocol stack on host NICs, RDMA is able to provide both microsecond level access latency and a per-connection throughput of approximately 100Gbps with almost zero CPU consumption [23]. The application of RDMA over Commodity Ethernet (RoCE) in data centers relies on the Priority Flow Control (PFC) mechanism to provide a lossless fabric.

In this paper, we present our experience in introducing RDMA into Pangu's storage networks (*i.e.*, the network among storage nodes). Our objective is to provide an RDMA-enabled Pangu system that exhibits superior performance, with availability and SLA standards equal to that of traditional TCP-backed versions. Our experience spans 4 years and will continue with the development of RDMA. We faced a number of challenges specifically related to cloud storage, with additional problems associated with RDMA. We have developed a number of solutions to allow for RDMA to function in a production-level cloud storage, several of which are engineering-level work-arounds. However, overcoming the aforementioned RDMA issues proves to be a complicated task. Here, we expose the practical limitations of the production systems in order to facilitate innovative research and applications in this area.

In addition to the performance, availability, and SLA requirements, the deployment planning of Pangu at the production scale should consider storage volume and hardware costs. Following the *availability-first* principal, RDMA communication is enabled only inside each *podset* [13]. Such a podset contains a group of leaf switches, and all Top-of-Rack (ToR) switches connected to these leaf switches. The podsets are connected via spine switches. This setting is currently the optimal balance between application demands, performance, and availability/SLA control. Storage node configurations are carefully planned to match the disk throughput with the network bandwidth. We adopt the hybrid deployment of RDMA/TCP in Pangu to exploit TCP as the last resort for the system (§3).

The performance optimization aims to minimize latency while maximizing throughput. We leverage *software-hardware co-design* to minimize performance overhead. We build a software framework in Pangu that integrates RDMA with Pangu's private user-space storage platform designed for new storage media. By eliminating data-copy operations, the latency of a typical block service request is reduced to tens of microseconds. We observed that the memory bandwidth becomes a bottleneck when upgrading Pangu to a 100Gbps network. By exploiting the RDMA features and offloading critical computations, Pangu is able to saturate the underlying networks. Furthermore, we leverage a new thread communication mode in Pangu to reduce the performance pitfall caused by a large number of Queue Pairs (QPs, RDMA connection abstraction) per node (§4).

Previous studies have reported the risks of large-scale RDMA deployment [13]. RDMA-enabled Pangu clusters do encounter such problems, including PFC deadlocks [13], PFC pause frame storms, and head-of-line blocking [27, 44]. We determined several PFC storms to be attributed to a previously unexplored source that consequently invalidates an earlier solution [13]. In order to guarantee availability, we apply the *escape-as-fast-as-possible* design principle to handle PFC storms. We bring up a fine-grained switching mechanism between RDMA/TCP traffic in Pangu and it handles PFC storms regardless of their causes (§5).

In order to meet the SLA standards, we adopt the design principal of *exploiting storage semantics whenever useful* in Pangu. By taking advantage of its ability to control the application layer, Pangu performs the real-time checking and alarming for a large number of storage service and network

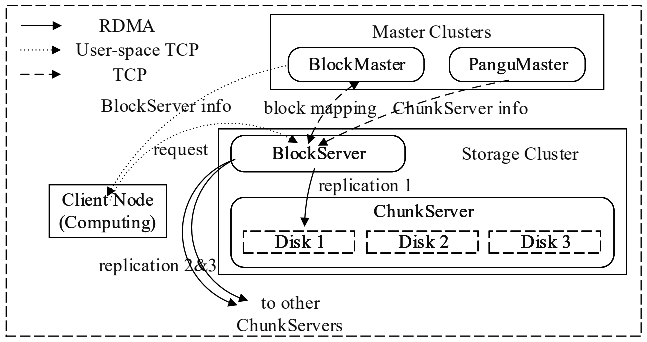


Figure 1: Pangu block storage service framework.

metrics. With the help of the dual-home topology feature, we optimize the fail-over performance of Pangu by reducing the connection recovery time. We also fix network problems by exploiting application controls, for example, blacklisting problematically connected nodes (§6).

We share our experience in adopting the RDMA-enabled Pangu system and discuss several potential research directions (§7). This system has successfully served numerous online mission-critical services under the scope of Alibaba over the past four years, including several important shopping festivals (*e.g.*, Double-11 [8]). Sharing our experience in integrating RDMA into Pangu can be helpful for other RDMA-enabled systems.

2 Background

2.1 Pangu in Alibaba Cloud

Pangu Framework. Pangu is a distributed file system developed by Alibaba Cloud. Released in 2009, it plays a major role in the core Alibaba businesses (*e.g.*, e-business and online payment, cloud computing, enhanced solid state drive backed cloud disk, elastic compute service, MapReduce-like data processing, and distributed database). In this paper, we focus on the network features of Pangu.

Pangu provides numerous storage services, including elastic block service, object storage service, store service, *etc.* We take the block service as an example to demonstrate the system framework. Fig. 1 presents the I/O workflows of Pangu. Virtual block devices contain continuous address spaces that can be randomly accessed by applications. A Pangu client in a computing node organizes data into fixed-sized (*e.g.*, 32 GB) *segments*, while the BlockServers and ChunkServers run on storage nodes. Each segment is aligned to a BlockServer for I/O processing. On the BlockServers, a segment is divided into blocks and replicated to the ChunkServers, which are in charge of the standalone back-end storage of the blocks and device management.

The BlockMasters manage metadata such as the mapping between a segment and its located BlockServer and the BlockServer's living states. The PanguMasters manage the states of the ChunkServers. These master nodes are synchronized using consistent protocols, such as Raft [36].

All data communication in Pangu is in the form of Remote Procedure Calls (RPCs). Each ChunkServer initiates the RPC clients/servers, and storage operations are performed by issuing pre-registered RPCs. An RPC client can simultaneously use different RPC channels (*i.e.*, connections via RDMA, kernel TCP, user-space TCP, or shared memory)according to the required RPCs.

Cloud Storage Requires RDMA. The principal performance metrics for storage services are read/write throughput and access latency. Low latency and high throughput prove to be advantageous for numerous application scenarios. Many customers expect similar performance of the cloud storage to that of the local physical storage. For example, the Alibaba e-commerce database requires extremely low latency in order to ensure fast responses due to the potentially large peak number of transactions per second (*e.g.*, 544,000 orders per second at peak hours [8]). Moreover, the enhanced SSD service promises 1 million IOPS, 4GB/s throughput, and 200µs latency for 4KB random writes [17].

The latency of traditional network stack (*e.g.*, TCP/IP) is generally within hundreds of microseconds [13]. The maximum achievable TCP bandwidth per kernel thread can reach tens of Gbps [51]. In contrast, the access latency of current NVMe SSDs is only at the microsecond level, while the read/write bandwidth of a single device is at the GB/s level [49]. The total throughput of each storage node (generally with 8-16 NVMe disks) can exceed 100Gbps and the incoming Storage Class Memory (SCM, *e.g.*, Intel 3D-XPoint) can even achieve nanosecond level latency [35]. Thus, networking is currently the primary performance bottleneck for cloud storage.

RDMA is an alternative networking choice for cloud storage. By implementing its entire protocol stack on host NICs, RDMA provides both microsecond level access latency and a per-connection throughput close to 100Gbps with almost no CPU consumption [23]. RDMA has successfully been integrated into numerous network-bottlenecked systems, for example, key-value stores [22, 33], distributed transactions [6, 24, 48], and graph queries [40], demonstrating an improved performance compared with non-RDMA predecessors.

2.2 Challenges

Besides performance, availability and SLA are also critical for a successful cloud storage system.

Availability. System disruptions incur significant financial/reputation loss for both tenants and their cloud providers. In 2018, Amazon S3 experienced a system disruption that lasted for 4 hours [2], affecting Amazon Elastic Compute Cloud , Amazon Elastic Block Store volumes, and AWS Lambda [3]. This disruption also had an impact on tens of thousands of websites built on the Amazon storage service, including Netflix [34], Spotify [43], Pinterest [37], and Buzzfeed [5]. Similar events have occurred with Google Cloud and Microsoft Azure [4, 9].

Service-Level Agreement. Software and hardware failures are extremely common in distributed systems. A cloud storage system should exhibit graceful performance downgrade with the occurrence of various failures. Distributed storage systems include mature node monitoring and fail-over mechanisms. A single storage node failure has a minimal impact on the service quality. In our experience, the most challenging aspect of ensuring a stable performance lies in the storage networks. Network failures generally result in a larger affected range compared to storage node failures.

In addition to its superior performance, customers of our RDMA-enabled Pangu require the same levels of availability and SLA standards to that of traditional TCP-backed versions.

2.3 State-of-the-art Work Do Not Fit

Unknown PFC Storm Sources. PFC runs under a hop-by-hop mechanism, with the possibility of PFC storms, spreading into the whole cluster. A PFC storm can seriously affect cluster availability and is the most well-known issue of RDMA. In 2016, Microsoft presented its experience in the deployment of RDMA [13], where they revealed that a bug in the receiving pipeline of an RDMA-capable NICs (RNICs) causes PFC storms. The problem was fixed by building watchdogs on the NICs and switches. However, we identified an additional type of PFC storms that originates from switches, implying the complexity of PFC storms with multifarious sources. The Microsoft solution [13] fails to solve this new problem (§5).

Practical Concerns that Limit Design Options. We are not able to simply treat RDMA as a black-box and wait for future research and technical advances to solve the current problems. Despite the large number of recent studies [10, 22, 29, 33, 40, 48], a production-level comprehensive PFC-free solution is still premature. The application of RDMA over lossy Ethernet has been explored in previous work [7, 11, 15, 26], allowing for the bypass of the PFC mechanism. However, such solutions rely on new hardware features.

The deployment of new hardware is a long process, with several months or even years of testing, followed by the subsequent introduction to business applications. For example, the process of testing Pangu with CX-4 RNICs, a joint collaboration with NIC providers, lasted for over two years. There is a tension between the fast growth of new RDMA demands and the long update cycles of new hardware. To date, these PFC-free proposals are not mature enough for large-scale business deployment, particularly for the availability and SLA standard requirements of cloud storage systems.

Furthermore, large-scale industry deployment is generally associated with multiple generations of legacy RDMA NICs. For example, we have already deployed several Mellanox NIC generations (*e.g.*, CX-4, CX-5), with the number of each reaching tens of thousands. It is operationally infeasible and costly to replace all legacy NICs in the running nodes,

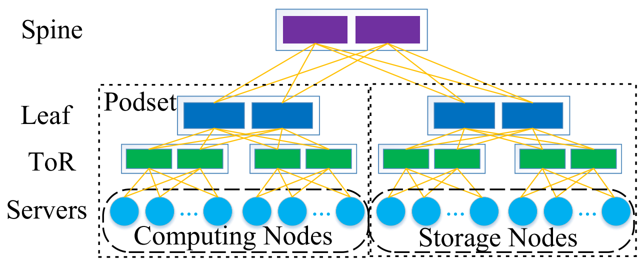


Figure 2: Topology of Pangu.

Hardware	25Gbps	100Gbps
CPU	Xeon 2.5GHz, 64 cores	Xeon 2.5GHz, 96 cores
Memory	DDR4-2400, 128GB	DDR4-2666, 128GB $\times3$
Storage	1.92TB SSD$\times$12	3.84TB SSD$\times$14
Network	CX-4 Lx Dual-port	CX-5 Dual-port
PCIe	PCIe Gen 3.0	PCIe Gen 3.0

Table 1: Example configurations of 25/100Gbps nodes.

while upgrading the firmware of tens of thousands of running servers is both time-consuming and error-prone. Thus, the need for new hardware features or firmware should be minimized.

Domain Knowledge of Distributed Storage Should be Exploited. Existing work largely ignores potential help from the application layer. Storage service metrics, rather than networking metrics, are a key concern for cloud service applications. We take into account such storage semantics in the design of Pangu when improving the engineering trade-off and the decision making process for various networking problems.

3 RDMA Deployment

3.1 Consideration in Deployment Planning

The deployment planning of storage clusters governs the network topology, RDMA communication scope, storage node configurations, *etc.* Multiple factors must be considered, including matching the storage volume with demands, controlling hardware costs, optimizing performance, and minimizing availability and SLA risks. The final outcome is a trade-off among all these factors.

For example, Microsoft deploys RDMA at the scale of an entire Clos network [13]. Thus, if not prevented, PFC storms could spread across the whole network and bring down an entire cluster. This amount of risk is unacceptable in a production-level storage system.

3.2 Deployment Choices of Pangu

The key principle employed by our RDMA deployment is *availability-first*.

Network and Node Configurations. Fig. 2 displays the Clos-based network topology of Pangu. Consistent with the common dual-home practice, we deploy Mellanox CX series dual-port RNICs to connect a host with two distinct ToR switches. In particular, two physical ports are bonded to a single IP address. Network connections (*e.g.*, QPs in RDMA) are balanced over two ports following a round-robin fashion.

Total bandwidth	TCP bandwidth ratio	TX pauses
25Gbps	40%	0
30Gbps	45%	1Kpps
32Gbps	50%	8Kpps
35Gbps	46%	15Kpps

Table 2: TX pauses in hybrid RDMA/TCP traffic.

When one port is down, the connections on this port can be migrated to another port.

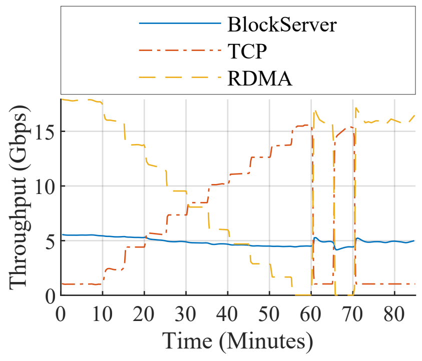
Table 1 reports typical hardware configurations for 25Gbps and 100Gbps RNIC storage nodes. The number of SSD per node is determined by the total RNIC bandwidth versus the throughput of a single SSD, allowing the I/O throughput to match the network bandwidth. Note that the SSD types in the 25Gbps and 100Gbps configurations are distinct, resulting in disproportional numbers. Computing and storage nodes are deployed in different racks within a single podset. The numbers of computing and storage nodes are then calculated according to the computational demands.

RDMA Scope. In order to minimize the failure domain, we only enable RDMA communication within each podset and among storage nodes. The communication between computing and storage nodes is performed via a private user-space TCP protocol (Fig. 1). This is attributed to the complex hardware configurations of computing nodes, which update rapidly. Thus, TCP can be effectively applied as a hardware-independent transport protocol. User-space TCP is more convenient for upgrade and management compared to kernel TCP, while kernel TCP is selected for cross-podset communication due to its generality.

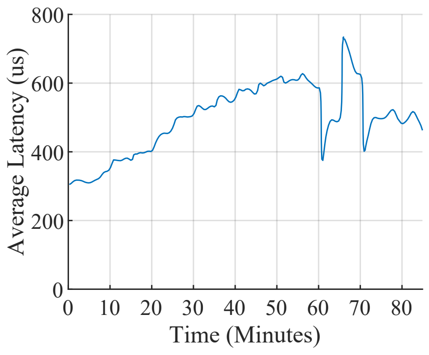
The production deployment is an additional concern for podset-level RDMA. In many datacenters, podsets are located in different buildings. For cross-building RDMA links, the base link delay is much larger, while the PFC mechanism requires much larger headroom buffer. In order to enable RDMA, the PFC/ECN thresholds located on the spine switches must be carefully adapted and tested. This is a tough task and at present, does not result in sufficient gains.

RDMA/TCP Hybrid Service. To the best of our knowledge, previous research on RDMA deployment does not explore RDMA and TCP hybrid services. We keep TCP as the last resort in Pangu following the *availability-first* principal. Despite current progress, RDMA devices are far from flawless. Thus, when either availability or SLA are threatened, switching affected links from RDMA to TCP can maintain the available bandwidth. This escape plan does not impact the unaffected RDMA links.

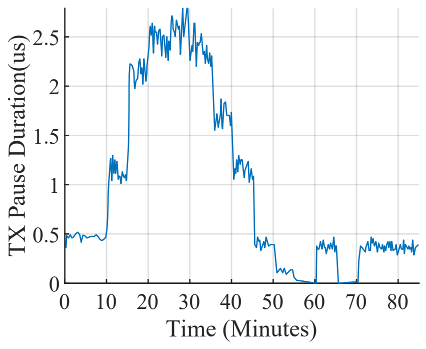
However, during the hybrid deployment process, we determined that coexistent TCP traffic provoked a large number of TX pauses (*i.e.*, PFC pause frames sent by NICs), even if RDMA/TCP traffic are isolated in two priority queues. Table 2 reports the TX pause generation rate in Pangu under different loads with approximately 50% TCP traffic. The tests are performed on Mellanox CX-4 25Gbps dual-port RNICs. Such a large number of TX pauses are detrimental to the



(a) Throughput of RDMA/TCP/BlockServer.



(b) Latency of BlockServer requests.



(c) Average TX pause duration.

Figure 3: RDMA/TCP hybrid deployment tests at different ratios (from 0% to 100% TCP).

performance and may result in PFC storms. We investigated this problem together with Mellanox and determined that the processing of TCP in the Linux kernel is highly I/O-intensive. Kernel TCP initiates too many partial writes on NICs' PCIe bus. As the PCIe bandwidth is consumed, the receiving pipeline of a NIC is slowed down. The buffer overflows and the NIC subsequently begins to transmit PFC pause frames.

In order to optimize the memory access of TCP, we make several adjustments on the data access procedure. First, disabling the Large Receive Offset (LRO) can reduce the memory bandwidth usage. This is attributed to the access of multiple cache lines when the LRO is enabled. Furthermore, enabling NUMA also improves the efficiency of memory accesses, which subsequently aids in relieving the pressure of PCIe. We also allocate a larger buffer on the RNICs for RDMA traffic to prevent TX pauses. Finally, making application data cacheline-aligned is a common optimization practice that improves memory efficiency [23].

3.3 Evaluation

We test several RDMA/TCP traffic ratios to investigate the effects of RDMA/TCP hybrid deployment. Each computing node runs FIO with 8 depths (inflight I/O requests), 8 jobs (working threads), and 16 KB block size in order to write virtual disks. Note that one write request on a BlockServer generates three data replicas. We enable all optimizations approaches detailed in §3.2 for the TCP kernel.

Fig. 3(a) depicts the BlockServer bandwidth with varying RDMA/TCP ratios. The workload starts at 10 minutes with 100% RDMA traffic. Afterwards, in every 5 minutes, the workload contains 10% more TCP traffic and 10% less RDMA traffic. At 60, 65, 70 minutes we change the TCP traffic ratio to 0%, 100%, and 0% respectively in order to explore the performance of Pangu with quick traffic switching between RDMA and TCP. The average BlockServer throughput exhibits minimal reduction as the RDMA traffic ratio decreases.

Fig. 3(b) presents the BlockServers' average request latency for the same workload as that in Fig. 3(a). The average latency under 100% RDMA traffic is approximately half of the latency under 100% TCP traffic, while the tail latency under 100% TCP is more than 10× larger compared to 100% RDMA traffic. RDMA presents great latency advantages compared to TCP. Fig. 3(c) demonstrates the average TX pause duration per second for this workload. Only a limited number of TX pauses are observed. When the TCP bandwidth ratio is around 50% at 30 minutes, the pause duration reaches a peak value.

Overall, these results demonstrate the stable performance of our RDMA/TCP hybrid mechanism.

4 Performance Optimization

4.1 Performance Hurdles

The performance optimization of Pangu aims to *minimize latency while maximizing throughput*.

RDMA-Storage Co-Design. Integrating the RDMA protocol stack with the storage backend is challenging. It must cover key performance points such as thread modeling, memory management, and data serialization. The thread model directly affects latency due to communication costs among threads. Well-designed memory management and data serialization are key to achieving zero-copy during data access. Here we present a brief introduction on the design of these components for storage purposes.

The User Space Storage Operating System (USSOS) is a unified user-space storage software platform that aims to support new storage media such as NVMe SSD and persistent memory. Its design principles (*e.g.*, memory management, shared memory mechanism, and user-space drivers) are based on well-known user-space technologies (*e.g.*, DPDK [19] and SPDK [42]). Related tests reveal that enabling USSOS in Pangu can improve CPU efficiency by more than 5× on average.

As a central part of USSOS, the User Space Storage File System (USSFS) is a high-performance local file system designed for SSDs. By running in the user space, USSFS is able to bypass the kernel to avoid user-/kernel-space-crossing overhead. USSOS divides disks into "chunks" which ChunkServer uses in its APIs (*e.g.*, create, seal, and delete).

Components	Average Utilization	Peak Utilization	Maximum Physical Capacity
Physical CPU utilization ratio	66%	70%	100%
Memory read/write throughput	28GB/s / 29GB/s	33GB/s / 32GB/s	61GB/s in total (1:1 read/write)
SSD PCIe throughput (socket 0 + socket 1)	550MB/s + 550MB/s	1000MB/s + 1000MB/s	3.938GB/s + 3.938GB/s
Network PCIe RX throughput	10GB/s	11GB/s	15.754GB/s
Network PCIe TX throughput	8GB/s	9GB/s	15.754GB/s

Table 3: Measured resource utilization of Pangu in 100Gbps network with 1:1 read/write ratio.

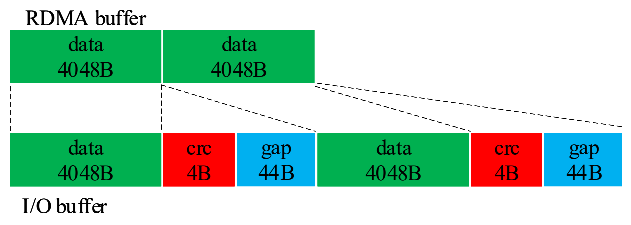


Figure 4: Potential triggering of data copying by CRC.

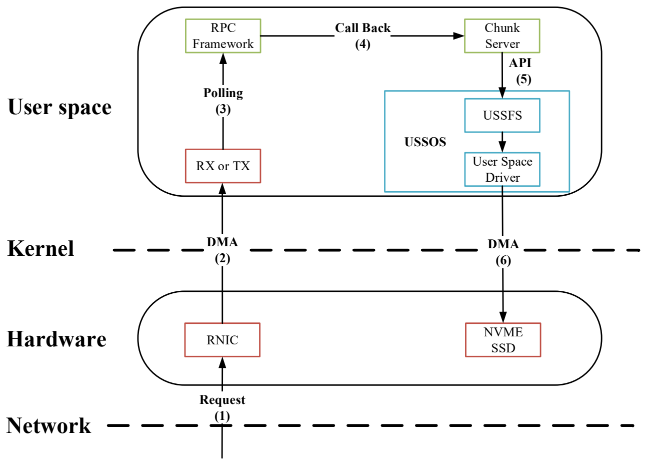


Figure 5: Integrated network/storage processing.

USSOS directly writes data and metadata to disks and uses polling to perceive completion events. For different block sizes, USSFS is able to improve IOPS by 4-10× compared to the Ext4 file system.

A run-to-completion model is considered as the optimal approach for the integration of the RDMA network stack with the storage stack. This model has previously been explored in studies discussing disaggregated storage (*e.g.*, Reflex [25], i10 [16]). However, these studies were published after the introduction of RDMA to Pangu in 2017. Reflex and i10 focus on remote direct I/O while a ChunkServer in Pangu is applied as a local storage engine for distributed storage. Google's Snap [31] leverages a separate network process to unify network functionalities and reduce the number of network connections.

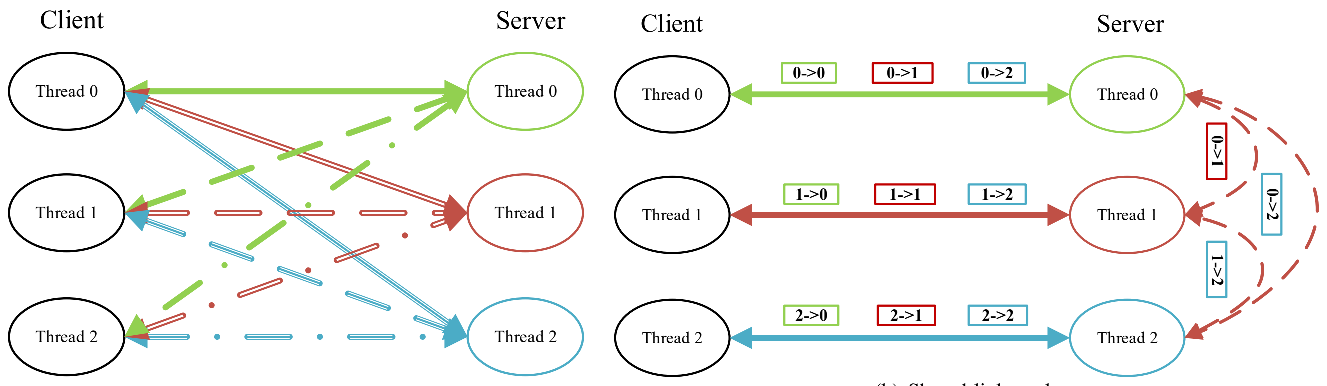
Memory Bottleneck with 100Gbps networks. Deploying 100Gbps networks can achieve lower latency and higher throughput. With faster network, now the memory throughput becomes a bottleneck in our system.

In order to obtain the upper bounds of the memory access throughput, we test the memory throughput using the Intel Memory Latency Checker (MLC) tool [20]. Table 3 details the measured usage of the hardware resources. In our test, the maximal achievable memory bandwidth is 61GB/s with a 1:1 read/write ratio. However, the average memory throughput with Pangu's workload is already $29\text{GB/s}+28\text{GB/s} = 57\text{GB/s}$. This indicates the memory to be the bottleneck rather than the network.

By monitoring the memory usage in Pangu, we determined that both the verification and data copy processes require optimization. Data integrity is one of the most significant features of distributed storage. We adopt Cyclic Redundancy Check (CRC) for application-level data verification in Pangu. As shown in Fig. 4, the received data is split into chunks of 4KB, with a 4B CRC value and a 44B gap added to each chunk. This operation is a memory- and computation-intensive operation as the calculations are applied to the entire dataset. The data are also copied when they are written into the disks in order to include CRC footers. Copying is not performed in other components due to the remote-memory access semantic of RDMA.

Large Number of QPs. We used to adopt the full-mesh link mode among running threads in Pangu in order to maximize throughput and minimize latency (Fig. 6(a)). Assume that each ChunkServer has 14 threads, each BlockServer has 8 threads, and each node contains both ChunkServers and BlockServers. For the full-mesh mode in a cluster of 100 storage nodes, there could be $14 \times 8 \times 2 \times 99 = 2,2176$ QPs in each node. RNICs' performance drop dramatically for large numbers of QPs due to cache miss [21]. In particular, the number of RX pauses (*i.e.*, PFC pause frames received) is very high.

Previous studies have demonstrated the same issue [10, 23, 47]. In order to solve this problem, FaSST [24] shares QPs among threads, which subsequently lowers the CPU efficiency and performance due to the lock contention of QPs between threads. An alternative heuristic is the inclusion of a dedicated proxy thread that manages all receive and send requests [41]. However, switching to/from a dedicated proxy thread increases latency. Furthermore, it is difficult to saturate the full network bandwidth with a single thread. Moreover, the proxy solution is not transparent to the underlying RDMA libraries.



(a) Full-mesh mode

(b) Shared-link mode

Figure 6: Different link modes for send/receive RPC requests

4.2 Designs

The designs related to performance in Pangu are based on the principle of *software-hardware co-design to minimize performance overhead.*

Storage-RDMA Unified Run-to-Completion Stack. We adopt a run-to-completion thread model for both storage and network to achieve low latency. Fig. 5 demonstrates the procedure used to process requests. When a write RPC is received by a node, the RNIC posts it to the user space via DMA. The RPC framework obtains the request using polling and subsequently hands it over to a ChunkServer module for processing. The ChunkServer then informs USSFS to allocate a "chunk" resource to the request. Finally, a user-space driver interacts with NVMe SSDs to store the data. These operations are generally performed in a single server thread without thread switching. This run-to-completion model minimizes the latency. In order to reduce the blocking time caused by large jobs, large I/O requests are split into smaller requests when submitted by applications. This optimization ensures a quick response to I/O signals. An additional optimization strategy for large I/O requests involves the passing of auxiliary work (*e.g.*, formatting and CRC calculation) to non-I/O threads, where they are subsequently processed. These optimizations reduce the average latency of a typical storage request (*e.g.*, 4KB size) to less than 30μs.

The data formats are unified as I/O vectors. An I/O vector is transmitted without copying via a single RDMA verb using scatter-gather DMA (the transfer of discontinuous data through a single interruption) in network. Serialization is not necessary due to RDMA semantics.

Zero-Copy & CRC Offloading. As discussed in §4.1, in Pangu, data has to be copied once on the I/O path as each 4KB chunk is verified and attached with a CRC footer. Here, we leverage the User-Mode Memory Registration (UMR) [32] feature of RNICs to avoid such data copy. UMR can scatter RDMA data on the remote side through the definition of appropriate memory keys. Thus, data can be formatted and organized according to storage application formats. We use UMR to remap the continuous data from the sender into an I/O buffer at the receiver, which contains 4KB data, a 4B footer, and a 44B gap in each unit. Following the CRC calculation, the filled I/O buffer can be directly applied for disk writing. Besides, the CRC calculation is able to be offloaded to capable RNICs (*e.g.*, Mellanox CX-5), thus lowering CPU and memory usage. The 4KB data are posted to the RNIC and the 4B CRC checksum is then generated.

Shared Link. We adopt the shared link mode, an effective solution for reducing the number of QPs in Pangu. The shared link mode is implemented in the application layer and leaves RDMA libraries untouched. A *correspondent* thread in the destination node is assigned to each thread in the source node (Fig. 6(b)). The thread's requests to the node are sent to its correspondent thread, which subsequently dispatches requests to correct target threads.

Consider a daemon with N threads, each thread polls N request/response queues to obtain the requests/responses. Note that there is only a single producer/consumer for each request/response queue. Thus we use lock-free queues for each request/response queue to avoid contention. According to our test, this design adds approximately 0.3 μs latency.

In the shared link mode, there is resource overhead at the correspondent thread during request dispatching when the source thread sends too many requests. Pangu supports shared groups, where threads in a node can be divided into several groups. A correspondent thread only relays requests for its group members. Returning to the previous example, the number of QPs in the *All Shared* mode is now reduced to $(8+8)\times 99 = 1,584$. If the threads are divided into 2 shared groups, the number of QPs will be $(8\times 2+8\times 2)\times 99 = 3,168$.

4.3 Evaluation

Zero-Copy & CRC Offloading. We use FIO with 16 jobs and 64 I/O depth to test a virtual I/O block device on a single ChunkServer. Fig. 7(a) demonstrates the memory bandwidth usage (including read/write tasks) when UMR zero copy and CRC offloading are used. The memory bandwidth usage is reduced by more than 30%, revealing that these measures are able to relieve the pressure of memory usage. Fig. 7(b) depicts

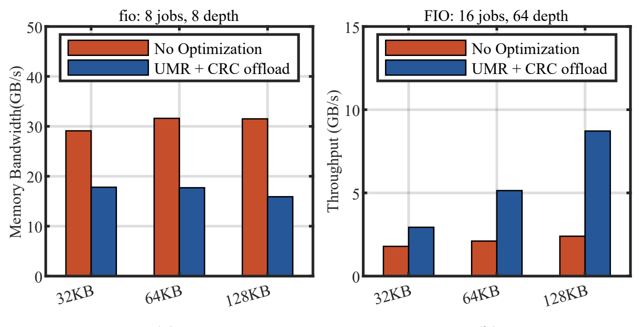


Figure 7: Performance of UMR zero copy + CRC offloading

the improvement in throughput following the optimization. The throughput of a single ChunkServer thread is improved by approximately 200% for a block size of 128KB.

Shared Link. We tested the shared link mode with several shared QP groups in a cluster of 198 computing nodes and 93 storage nodes. The background workload compromises 4KB random writes with 8 threads and 8 I/O depths. Fig. 8(a) presents the throughput in the All Shared, 2 Shared Groups, and 4 Shared Groups modes, whereby a performance trade-off can be observed. The All Shared mode exhibits slightly lower throughput but generates the lowest number of PFC pauses. Note that the reduction in bandwidth at 5 and 24 minutes in the All Shared mode is attributed to the garbage collection mechanism in Pangu. Fig. 8(b) presents the TX pause duration with 1, 2, and 4 Shared Groups, respectively. The lower the number of groups, the fewer the PFC pauses are generated due to the reduction in QP number. We use the All Shared Group mode in our scale and configuration framework.

5 Availability Guarantee

5.1 PFC Storms

A New Type of PFC Storm. The PFC storm previously discussed in [13] originates from the NICs, with a bug in the receiving pipeline acting as the root cause. Fig. 9(a) depicts the phases of this PFC storm: (1) The bug slows down the NIC receive processing, filling its buffer; (2) the NIC transmits the PFC pauses to its ToR switch in order to prevent packet drop; (3) the ToR switch pauses the transmission; (4) the ToR switch's buffer becomes full and starts to transmit the PFC pauses; and (5) the victim ports are paused and are unable to transmit.

We encountered a different type of PFC storm when operating RDMA in Pangu. The root cause is a bug in the switch hardware of a specific vendor. The bug reduces the switching rate of the lossless priority queues to a very low rate. Fig. 9 compares the two types of PFC storms. As an example, we assume that the bug occurs in a down port of a ToR switch: (1) due to the low transmitting rate, the switch's buffer becomes full; (2) the switch transmits the PFC pauses to the connected ports; and (3) the additional switches and NICs stop the transmissions. The leaf switches and NICs connected to this ToR switch receive continuous pause frames and thus the storm spreads.

State-of-the-Art Solutions. Guo *et al.* [13] built a NIC-based watchdog to continuously monitor transmitted pause frames, disabling the PFC mechanism if necessary. In addition, watchdogs were also deployed on the switches for disabling the PFC mechanism when switches receive continuous pause frames and are unable to drain the queuing packets. The switches can subsequently re-enable PFC in the absence of pause frames over a specific period of time. Thus, PFC storms can be controlled via these two watchdogs during phase (2).

However, this solution is unable to completely solve the PFC storms originating from switches. In particular, the TX pause watchdogs on the NICs will not work since the NIC only receives PFC storms from the switches. Furthermore, current switch hardware does not support the monitoring of pause frame transmissions. If a storm occurs on a ToR switch, even though the watchdogs on other switches are able to stop its spread, the ToR switch will continue to send pauses to end-hosts in the rack. The RDMA traffic via this ToR is consequently blocked.

Challenges. This new type of PFC storms invalidates Guo *et al.*'s solution, which focuses on insulating the PFC pause sources to prevent the storm from spreading. This methodology fails when the source is a ToR switch as all the hosts in the ToR are paused by the storm. Therefore, in order to achieve high availability, a general solution is required in Pangu to handle all PFC storm types, particularly those with unknown causes.

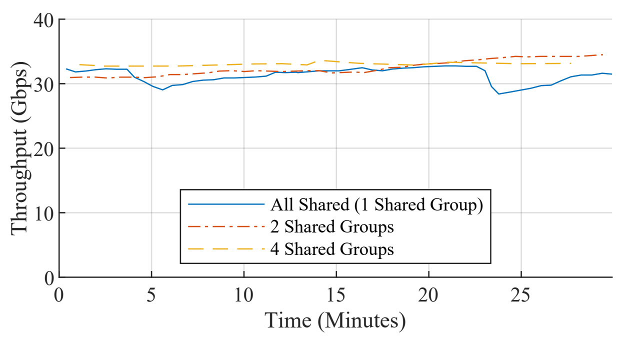
Ensuring the service quality of Pangu while simultaneously solving PFC storms is challenging. PFC storm detection must be timely and accurate to rapidly protect network traffic. In terms of availability, the overall convergence time of the PFC storm should be controlled to at most the minute level.

5.2 Design

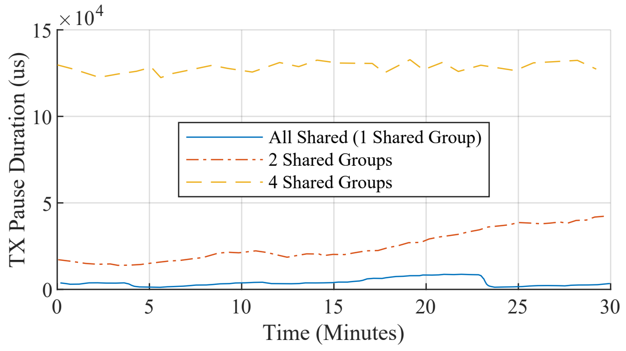
Our design principle of handling PFC storms is *escaping as fast as possible*. Despite new PFC storm solutions [11, 21, 26], we still resort to engineering-level work-arounds due to practical considerations (§2.3).

In Pangu, each NIC monitors the received PFC pause frames. For continuous pause frames, the NIC determines the presence of a PFC storm. Two work-around solutions are available for administrators in the case of a PFC storm.

Workaround 1: Shutdown. This solution, denoted as the "shutdown" solution, shuts down NIC ports affected by PFC storms for several seconds. The dual-home topology provides an emergency escape for PFC storms, whereby QPs will disconnect and connect again via another port. This method works together with the optimization to reduce the length of the QP timeout. This optimization is discussed further in §6.2. Although this solution is simple and effective, it is sub-optimal due to the loss of half of the bandwidth.

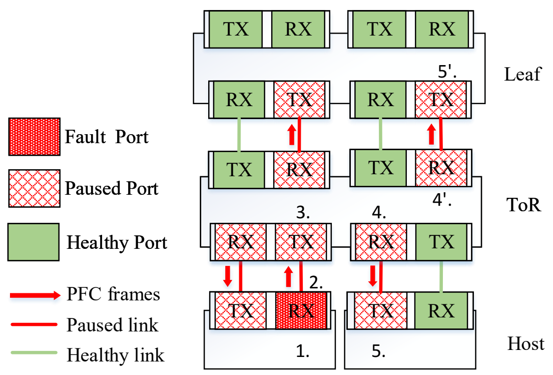


(a) Throughput in different thread groups.

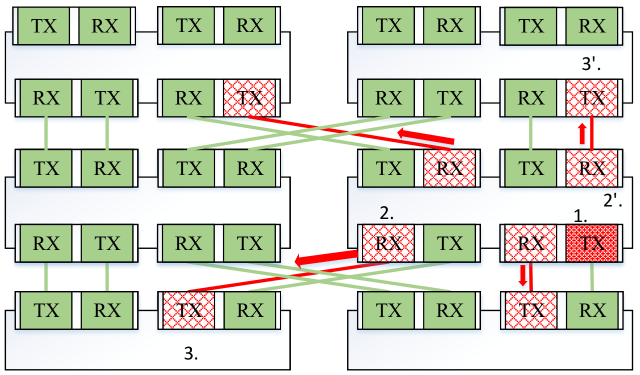


(b) Pause time in different thread groups.

Figure 8: Throughput and pause for different types of thread groups.



(a) The PFC storm originates in NICs.



(b) The PFC storm originates in switches.

Figure 9: Different types of PFC storms.

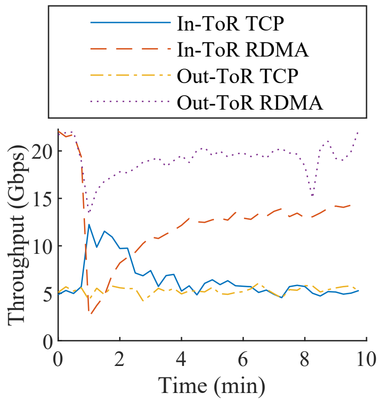
Workaround 2: RDMA/TCP Switching. In this solution, the affected RDMA links in a PFC storm are switched to TCP links. It compromises a more complex procedure compared to the shutdown solution, yet it is able to maintain the available bandwidth. We adopt a method similar to PingMesh [14] to detect the RDMA links affected in PFC storms. At each T ms, every worker thread picks a server and separately pings all its threads via the RDMA and TCP links. If the RDMA ping fails and the TCP ping succeeds for more than F times, the traffic on this RDMA link is switched to the TCP link. Once the RDMA ping has succeeded more than S times, the traffic on the switched TCP link is switched back to the RDMA link. For $T = 10$ ms and $F = 3$, bad RDMA links can be detected in approximately 10 seconds in a podset of 100 storage nodes. By switching the RDMA traffic to the TCP connections, the throughput can recover to more than 90% in less than 1 minute.

5.3 Evaluation

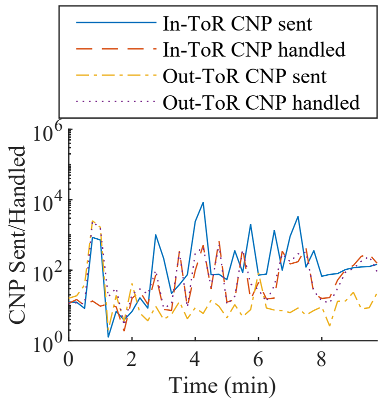
We simulate PFC storms by injecting the aforementioned bug into a switch for several cases, including the uplink/downlink ports on the ToR and Leaf switches. The RDMA/TCP switching solution exhibits strong performance for all cases. Fig. 10 displays the results for a PFC storm originating from a ToR switch downlink port. Note that the nodes inside the ToR behave differently from nodes outside the ToR. We choose two nodes (inside and outside the ToR) in order to demonstrate the difference. In such a case, the pause frames are transmitted to NICs and leaf switches directly connected to the given ToR switch.

The shutdown solution shuts down the NICs via the watchdogs in the occurrence of a fault due to excessive RX pauses. RDMA links subsequently reconnect through another NIC port, thus recovering traffic. Note that the counters of Congestion Notification Packet (CNP) and PFC frames gradually increase since the system load (at 0 minutes) is larger than the available bandwidth of a single port (25Gbps). The system then reaches a new balance in approximately 30 minutes. However, the shutdown solution has several limitations. For example, computing node requests may not respond within 1 minute (known as I/O hang sensed by applications). The downlink breakdown of a leaf or ToR switch can result in tens to hundreds of hang requests. Furthermore, the shutdown of ports is itself an aggressive action. Hundreds of ports may be shut down due to unexpected pauses. This risk may itself influence the availability of a large number of nodes.

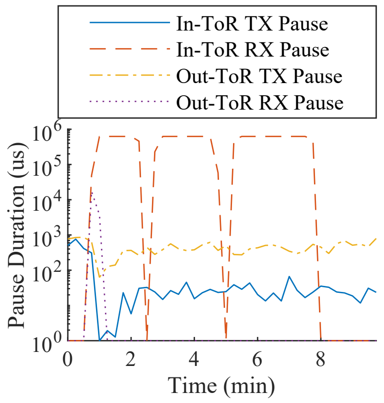
The RDMA/TCP switching solution switches the RDMA traffic that passes through the broken-down switch to TCP.



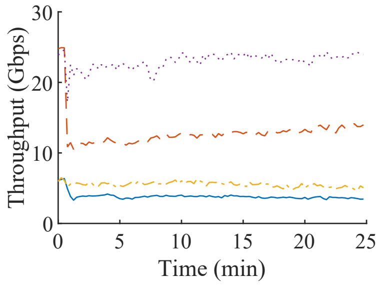
(a) Throughput of RDMA/TCP switching.



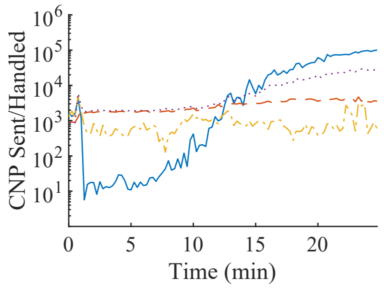
(b) CNP of RDMA/TCP switching.



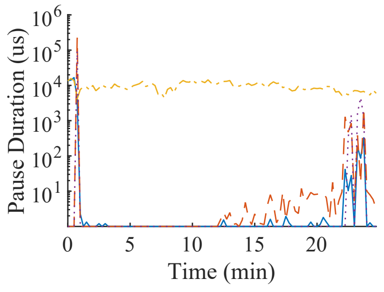
(c) PFC Pause of RDMA/TCP switching.



(d) Throughput of shutdown.



(e) CNP of shutdown.



(f) PFC Pause of shutdown.

Figure 10: Performance of two different solutions for PFC storms.

The RDMA links are then disconnected due to timeout. The QPs are separately distributed over the server's two NIC ports, thus the RDMA links may need several attempts to reconnect successfully. Note that although the pause storm in the ToR is not terminated, it will not spread further as the neighboring switch ports are transformed into the lossy mode via the RX pause watchdogs. The traffic throughput is not impacted during the migration to the TCP, and I/O hangs are not present.

6 SLA Maintenance

6.1 SLA in Pangu

It is commonly-known that network failures are hard to locate and repair. Network failure causes include mis-configuration, hardware/software bugs, and link errors. For example, the mis-configuration of the switch Access Control List (ACL) may only affect a specific flow while other flows behave normally [28, 45]. As a comparison, malfunctions occurring at storage devices or nodes can generally be easily located.

Sometimes network failures may not be explicit. Ideally, when a node breaks down, the heartbeat mechanism should ensure that the unavailability of service daemons (BlockServers and ChunkServers) on the node are informed to their masters (BlockMasters and PanguMasters). However, real situations can be more complicated, failing to be detected with just heartbeats. Connections may suffer intermittent packet loss or long latency rather than simple break downs. We also identified an interesting failure type involving a small number of links that flap between up and down states for a short period of time (*e.g.*, several seconds). This results in an extremely high tail latency for I/O requests, denoted as slow I/O (*e.g.*, over 1 second for storage clients). Hundreds of slow I/Os are observed daily for numerous reasons. Root causes of link flapping include optical ports covered with dust, loose physical interfaces, aging hardware, *etc.*

Previous Research on Network Failures. The majority of previous studies focus on determining the location of network failures (*e.g.*, Pingmesh [14] and 007 [1]). These solutions focus on the system network and can achieve the timely discovery of network errors. However, it may still take hours for engineers to manually check, fix, and replace the failed switches and links. Cloud storage calls for a methodology that integrates the storage and network function modules to ensure stable service quality in failed cases.

6.2 Design

Our SLA design principle aims to *exploit storage semantics whenever useful*. Distributed storage is designed with a redundancy mechanism and its performance is measured via the storage semantics. These semantics, such as data replicas and distributed placements, can be leveraged during a failure to improve system performance.

Network-Integrated Monitoring Service. Monitoring is a necessary component of distributed systems. A comprehensive monitoring system is key for the safe deployment of RDMA in production storage systems, as well as reliable performance.

Both configurations and counters must be monitored. NIC configurations include system environments, PFC/ECN/QoS configurations, and several link configurations. Pangu automatically checks, collects, and reports suspicious terms. Inspecting potential mis-configurations can reduce configuration and environment errors. For example, accidental reboot and software upgrades may reset the QoS, PFC, DCQCN [52] configurations and affect system performance. Monitoring can discover such cases and help fix them in advance.

Counters include storage system performance counters (*e.g.*, IOPS/latency) and network counters (*e.g.*, CNP sent/handled, TX/RX pauses, and RDMA errors on NICs). Congestion control counters that exceed thresholds can result in monitor daemons sending alarms to engineers for diagnosis and repair. Monitoring both storage and network indexes is crucial for the diagnosis of errors and for predictable performance. Storage indexes such as tail latency, slow I/O, and queuing time can directly reflect the status of a system. Moreover, monitoring system performance also help locate errors. For example, network features are unable to quickly reflect the flapping problem described in §6.1. However, this problem can be easily located by monitoring slow I/Os on the endpoints of storage applications.

Faster Timeout for Connection Failures. The basic solution to network failures is to reconnect through an alternative path. Since we use dual-home topology, each single point failure of the network can be bypassed using a different path. Thus, the timeout duration of the QPs is crucial in improving the system performance during a failure. In the NIC manual, the timeout duration of QPs is calculated as $4\mu s \times 2^{\text{timeout}} \times 2^{\text{retry\_cnt}}$, where $timeout$ and $retry\_cnt$ denote the retransmission timeout value and the retransmission retry times respectively. Initially, this value was a constant (approximately 16 seconds) configured in the hardware and cannot be changed. In a combined effort with the NIC providers, we were able to fix this bug. By using a smaller timeout value for QPs, the action time required for reconnecting during network failures was reduced by 4×.

An alternative work-around involves altering the connection path by modifying the source ports of the QPs (rather than a direct reconnection). This can accelerate the link recovery during a fail-over. However, effectively changing the QP source port requires a more recent NIC firmware (MLNX OFED 4.7 or newer) than what is currently deployed in Pangu. We leave this challenge to future work.

Blacklist. We adopt *blacklist* in Pangu to further improve the service quality in fail-over cases. BlockMasters collect information on I/O errors (including timeout) and slow I/Os from clients. If a BlockServer has a large number of slow/error I/Os from multiple clients, the masters adds it to the blacklist for a short period of time. The number of clients and slow/error I/Os that triggers the blacklist is configured according to the scale of the cluster. In order to ensure reliability, the maximum number of BlockServers in the blacklist is usually small (*e.g.*, 3 hosts). This blacklist mechanism temporarily isolates the BlockServers that provide a poor service. The system performance is not affected and engineers have sufficient time to fix the problems.

6.3 Daily Operation Scheme of Pangu

The daily operations of Pangu rely on these modules to work together. The monitoring system collects and reflects the status of Pangu. If abnormal network indicators or I/O metrics are observed, the monitoring system attempts to locate and report them to the administrators. For accidental failures such as link errors, the small QP timeout shortens the time required for failure recovery. The blacklist mechanism is able to determine and isolate nodes with poor service quality. By following these design and operator framework specifications, our RDMA-enabled Pangu system has not experienced any major faults in the last two years.

7 Experiences and Future Work

Monitoring NACK in Lossless RDMA. The operation of RDMA over a lossless fabric is difficult due to PFC risks. However, the lossless fabric increases the effectiveness of NACK events as indicators of the network error location since NACK is usually rare in a lossless fabric.

In order to detect and locate network problems, we build a subsystem based on packet loss in Pangu. In particular, Out-Of-Sequence (OOS) counters on RNICs and packet drop counters on switches are gathered. A packet loss is classified as either explicit or implicit based on whether it is recorded by switch counters. The former is easy to locate by checking the switch counters. However, determining the location of the latter is complex as RNIC counters do not distinguish between flows. By monitoring NACK in the networks, we can extract flows' five tuples and locate the root of a problem.

Building a System-Level Benchmark. To evaluate the system network performance and SLA, a representative benchmark must be constructed. Building the benchmark based on just the network metrics is simple. However, storage features such as replica completion time, slow I/O, and failure recovery should not be ignored. To measure the storage system performance and the SLA, we build a benchmark at the storage service level. The system evaluation indexes include FIO latency, IOPS, SLA with network errors, *etc.* Each upgrade in Pangu (for network and other components) is evaluated with the benchmark, allowing us to measure the overall system performance.

Congestion Control for Fail-Over Scenarios. In §3, we introduced the dual-home topology adopted in Pangu. Dual-home topology is also crucial to fail-over cases since it provides a backup path on NICs. However, we encounter a problem when deploying dual-home topology in practice.

According to our test, when one ToR switch is down, the RX pause duration can increase to 200ms per second. This is due to the transfer of the traffic from the broken ToR switch to the other switch. DCQCN handles the traffic burst poorly under this asymmetric topology. We adapt several DCQCN parameters as a temporary solution and leverage the fluid models [53] to analyze the available choices, including canceling the Fast Recovery stage and extending the rate increase duration. When removing the Fast Recovery stage in DCQCN, the pause can be eliminated yet the flow tail latency increases due to the slow recovery of the flow rate. In contrast, extending the duration of the rate-increase can result in a sharp reduction in the pause but only slightly increases the flow tail latency. In our experience, extending the rate-increase duration in DCQCN is effective for storage traffic patterns. The bandwidth drops slightly while the number of RX pauses is dramatically reduced.

This problem of DCQCN in fail-over scenarios (*e.g.*, asymmetric topology and traffic burst) indicates their important role when designing congestion control. We adopt parameter tuning to fix this problem at the price of a slight performance loss. The storage network still requires a flexible, robust and well-implemented congestion control mechanism that functions well in all scenarios. In 2019, Alibaba designed HPCC [30], a novel congestion control algorithm for the RDMA network. Adapting and integrating HPCC with the storage networks is left for future work.

Slow Processing of RDMA Read. The majority of large RPCs in Pangu are transferred via RDMA READ. We observed that when a NIC receives too much RDMA requests within a short period, it will send out many PFC frames. This is due to the slowed receiving process that results from cache misses. When a NIC is preparing for an RDMA READ, it accesses the QP context in its cache. Processing many RDMA READs consumes an excessive amount of cache resources. For slow RX rates, the NIC sends out PFC pause frames to prevent packet drops. We are currently working with the RNIC provider to solve this problem.

Lossy RDMA in Storage. Lossy RDMA is supported by Mellanox CX-5 and CX-6 RNICs. Note that CX-6 supports Selective Repeat (SR) retransmission. SR might be the ultimate step required to effectively eliminate PFC. The construction of lossy RDMA is a focal point for all RDMA-based systems. We tested lossy RDMA with Pangu over an extensive period and will deploy it for new clusters.

However, enabling the lossy feature with early generation RNICs (*e.g.*, CX-4) that have limited hardware resources and do not support SR is hard, and many production RDMA-based systems still host early generations RNICs.

NVMe-Over-Fabric. The ChunkServer data flow in Pangu is processed by CPUs. However, with NVMe-Over-Fabric, NICs can directly write the received data into NVMe SSDs. This CPU-bypass solution can save CPU costs and reduce latency. We are currently building our specialized storage protocol (and corresponding hardware) based on NVMe-over-Fabrics. A customized storage protocol for Pangu with hardware support can allow for more flexibility and control.

8 Related Work

PFC in RDMA PFC storm is the most well-known issue of RDMA. Several studies [11, 30, 52] focus on controlling network congestion to reduce the numbers of generated PFC pauses. DCQCN [52] is integrated in Mellanox RNICs. In Pangu, we tune several parameters in DCQCN to improve its performance in fail-over scenarios. However, PFC storms still occur due to hardware bugs [13]. In this paper, we present a different hardware bug that originates from switches. Existing solutions to remedy PFC storms include deadlock elimination [38] and performance optimization [46]. These solutions require switch modification. In Pangu, we combat PFC storms by switching affected links from RDMA to TCP without the need for any switch changes.

System & Network Co-Design. Recently, there have been increasing amount of work that adopts system and network co-design, including RPC systems [21, 47], distributed memory systems [39], key-value stores [22], distributed databases and transaction processing systems [6], and graph-processing systems [40]. We co-design our storage system and RDMA in Pangu. To our best knowledge, we are the first to share the experience of employing RDMA networks in large-scale distributed storage systems.

9 Conclusions

As a distributed storage system, Pangu has provided storage services to tenants both inside and outside of Alibaba for over a decade. In order to overcome the challenges of rising high-speed storage media and growing business requirements, we integrate RDMA into the storage network of Pangu, providing a common solution to different types of PFC storms. This allows for the safe deployment of RDMA. Pangu has successfully moved to a 100Gbps network by solving several new problems, such as the memory bandwidth bottleneck and QP number explosion. Furthermore, we improve the system performance of Pangu in fail-over cases.

Acknowledgment

We are really grateful to Yiying Zhang for shepherding our paper. We also thanks anonymous reviewers for their constructive suggestions. Yixiao Gao and Chen Tian are supported in part by the National Key R&D Program of China 2018YFB1003505, the National Natural Science Foundation of China under Grant Numbers 61772265, 61802172, and 62072228, the Fundamental Research Funds for the Central Universities under Grant Numbers 14380073, the Collaborative Innovation Center of Novel Software Technology and Industrialization, and the Jiangsu Innovation and Entrepreneurship (Shuangchuang) Program.

References

[1] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang (Harry) Liu, Jitu Padhye, Boon Thau Loo, and Geoff Outhred. 007: Democratically finding the cause of packet drops. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 419–435, Renton, WA, April 2018. USENIX Association.

[2] Amazon AWS. Summary of the amazon s3 service disruption in the northern virginia (us-east-1) region. s https://www.usatoday.com/story/tech/news/2017/02/28/amazons-cloud-service-goes-down-sites-scramble/98530914/, 2020.

[3] Amazon AWS. Summary of the amazon s3 service disruption in the northern virginia (us-east-1) region. s https://aws.amazon.com/cn/message/41926/, 2020.

[4] Microsoft Azure. Azure status history. s https://status.azure.com/status/history/, 2020.

[5] Buzzfeed. Buzzfeed website. s https://www.buzzfeed.com/, 2020.

[6] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. Fast and general distributed transactions using rdma and htm. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 26. ACM, 2016.

[7] Inho Cho, Keon Jang, and Dongsu Han. Credit-scheduled delay-bounded congestion control for datacenters. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 239–252. ACM, 2017.

[8] Alibaba Cloud. How does cloud empower double 11 shopping festival. s https://resource.alibabacloud.com/event/detail?id=1281, 2020.

[9] Google Cloud. Google cloud networking incident no.19005. s https://status.cloud.google.com/incident/cloud-networking/19005, 2020.

[10] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, pages 401–414, 2014.

[11] Yixiao Gao, Yuchen Yang, Tian Chen, Jiaqi Zheng, Bing Mao, and Guihai Chen. Dcqcn+: Taming large-scale incast congestion in rdma over ethernet networks. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, pages 110–120. IEEE, 2018.

[12] Alibaba Group. Alibaba group website. s https://www.alibabagroup.com/en/global/home, 1999-2020.

[13] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. Rdma over commodity ethernet at scale. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*, pages 202–215. ACM, 2016.

[14] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 139–152. ACM, 2015.

[15] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 29–42. ACM, 2017.

[16] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. TCP ≈ RDMA: Cpu-efficient remote storage access with i10. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 127–140, Santa Clara, CA, February 2020. USENIX Association.

[17] Alibaba Inc. Block storage performance. s https://www.alibabacloud.com/help/doc-detail/25382.html?spm=a2c5t.10695662.1996646101.searchclickresult.458e478fYtRYOO, 2018.

[18] Alibaba Inc. Pangu, the high performance distributed file system by alibaba cloud. s https://www.alibabacloud.com/blog/pangu-the-high-performance-distributed-file-system-by-alibaba-cloud_594059, 2018.

[19] Intel. Data plane development kit. s https://www.dpdk.org/, 2011.

[20] Intel. Intel memory latency checker. s https://software.intel.com/content/www/us/en/develop/articles/intelr-memory-latency-checker.html, 2020.

[21] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter rpcs can be general and fast. In *USENIX NSDI*, pages 1–16, 2019.

[22] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using rdma efficiently for key-value services. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 295–306. ACM, 2014.

[23] Anuj Kalia, Michael Kaminsky, and David G Andersen. Design guidelines for high performance rdma systems. In *2016 USENIX Annual Technical Conference*, page 437, 2016.

[24] Anuj Kalia, Michael Kaminsky, and David G Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (rdma) datagram rpcs. In *OSDI*, volume 16, pages 185–201, 2016.

[25] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Reflex: Remote flash = local flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 345–359, New York, NY, USA, 2017. Association for Computing Machinery.

[26] Yanfang Le, Brent Stephens, Arjun Singhvi, Aditya Akella, and Michael M Swift. Rogue: Rdma over generic unconverged ethernet. In *SoCC*, pages 225–236, 2018.

[27] David Lee, S Jamaloddin Golestani, and Mark John Karol. Prevention of deadlocks and livelocks in lossless, backpressured packet networks, February 22 2005. US Patent 6,859,435.

[28] Dan Li, Songtao Wang, Konglin Zhu, and Shutao Xia. A survey of network update in sdn. *Frontiers of Computer Science*, 11(1):4–12, 2017.

[29] Hao Li, Asim Kadav, Erik Kruus, and Cristian Ungureanu. Malt: distributed data-parallelism for existing ml applications. In *Proceedings of the Tenth European Conference on Computer Systems*, page 3. ACM, 2015.

[30] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. Hpcc: high precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 44–58. ACM, 2019.

[31] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 399–413, New York, NY, USA, 2019. Association for Computing Machinery.

[32] Mellanox. Mellanox rdma progamming manual. s https://www.mellanox.com/sites/default/files/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf, 2015.

[33] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *USENIX Annual Technical Conference*, pages 103–114, 2013.

[34] Netflix. Netflix website. s https://www.netflix.com/, 2020.

[35] J. Niu, J. Xu, and L. Xie. Hybrid storage systems: A survey of architectures and algorithms. *IEEE Access*, 6:13385–13406, 2018.

[36] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association.

[37] Pinterest. Pinterest website. s https://www.pinterest.com/, 2020.

[38] Kun Qian, Wenxue Cheng, Tong Zhang, and Fengyuan Ren. Gentle flow control: avoiding deadlock in lossless networks. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 75–89. ACM, 2019.

[39] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. Distributed shared persistent memory. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 323–337, 2017.

[40] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. Fast and concurrent rdf queries with rdma-based distributed graph exploration. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 317–332. USENIX Association, 2016.

[41] Galen M Shipman, Stephen Poole, Pavel Shamis, and Ishai Rabinovitz. X-srq-improving scalability and performance of multi-core infiniband clusters. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 33–42. Springer, 2008.

[42] SPDK. Storage performance development kit. s https://www.spdk.io, 2020.

[43] Spotify. Spotify website. s https://www.spotify.com/, 2020.

[44] Brent Stephens, Alan L Cox, Ankit Singla, John Carter, Colin Dixon, and Wesley Felter. Practical dcb for improved data center networks. In *IEEE INFOCOM 2014-IEEE Conference on Computer Communications*, pages 1824–1832. IEEE, 2014.

[45] Bingchuan Tian, Xinyi Zhang, Ennan Zhai, Hongqiang Harry Liu, Qiaobo Ye, Chunsheng Wang, Xin Wu, Zhiming Ji, Yihong Sang, Ming Zhang, Da Yu, Chen Tian, Haitao Zheng, and Ben Y. Zhao. Safely and automatically updating in-network acl configurations with intent language. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 214–226, New York, NY, USA, 2019. Association for Computing Machinery.

[46] C. Tian, B. Li, L. Qin, J. Zheng, J. Yang, W. Wang, G. Chen, and W. Dou. P-pfc: Reducing tail latency with predictive pfc in lossless data center networks. *IEEE Transactions on Parallel and Distributed Systems*, 31(6):1447–1459, 2020.

[47] Shin-Yeh Tsai and Yiying Zhang. Lite kernel rdma support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 306–324. ACM, 2017.

[48] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 87–104. ACM, 2015.

[49] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Manu Awasthi, Tameesh Suri, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. Performance characterization of hyperscale applicationson on nvme ssds. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS' 15, pages 473–474, New York, NY, USA, 2015. Association for Computing Machinery.

[50] Yiying Zhang and Steven Swanson. A study of application performance with non-volatile main memory. In *Symposium on Mass Storage Systems and Technologies*, 2015.

[51] Yang Zhao, Nai Xia, Chen Tian, Bo Li, Yizhou Tang, Yi Wang, Gong Zhang, Rui Li, and Alex X. Liu. Performance of container networking technologies. In *Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems*, HotConNet '17, page 1–6, New York, NY, USA, 2017. Association for Computing Machinery.

[52] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 523–536, New York, NY, USA, 2015. Association for Computing Machinery.

[53] Yibo Zhu, Monia Ghobadi, Vishal Misra, and Jitendra Padhye. Ecn or delay: Lessons learnt from analysis of dcqcn and timely. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pages 313–327. ACM, 2016.

Prism: Proxies without the Pain

Yutaro Hayakawa
LINE Corporation

Michio Honda
University of Edinburgh

Douglas Santry
Apple Inc.

Lars Eggert
NetApp

Abstract

Object storage systems, which store data in a flat name space over multiple storage nodes, are essential components for providing data-intensive services such as video streaming or cloud backup. Their bottleneck is usually either the compute or the network bandwidth of customer-facing frontend machines, despite much more such capacity being available at backend machines and in the network core. Prism addresses this problem by combining the flexibility and security of traditional frontend proxy architectures with the performance and resilience of modern key-value stores that optimize for small I/O patterns and typically use custom, UDP-based protocols inside a datacenter. Prism uses a novel connection hand-off protocol that takes the advantages of a modern Linux kernel feature and programmable switch, and supports both unencrypted TCP and TLS, and a corresponding API for easy integration into applications. Prism can improve throughput by a factor of up to 3.4 with TLS and by up to 3.7 with TCP, when compared to a traditional frontend proxy architecture.

1 Introduction

A scale-out architecture for object storage systems is essential not only for supporting large storage capacities but also to incorporate sufficient compute and network bandwidth so the system can offer a predictable high-throughput and low-latency service to clients. Non-volatile memories (NVM) now fill the performance gap between networking and storage [28] with measured throughputs of 39.4 Gb/s and access latencies of 305 ns [34], emphasizing the importance of minimizing storage stack overheads for use with NVM [33].

A common design pattern seen in object storage systems [2, 3, 22, 35, 41, 65] uses a set of frontend machines to mediate client requests and relay them to a set of backend machines, as illustrated in the left diagram in Figure 1. The frontend often acts as a cache and/or load balancer to sharded or replicated backends. In this architecture, handling small objects incurs severe network inefficiencies, and handling large objects is limited by both network and compute resource availability because of data movement and encryption [2]. However, such systems have been widely adopted because of practical tractability when encryption and filtering are required, and because the performance characteristics of traditional disk-based backends are so poor that the CPUs and network typically are under-utilized, i.e., are not bottlenecks [67].

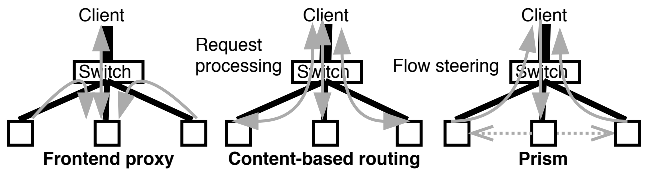


Figure 1: **Scale-out architecture variants.**

Recent approaches such as SwitchKV [45], NetCache [37] and Pegasus [44] use a content-based routing architecture, where servers interact with programmable switches, as illustrated in the central diagram in Figure 1. These approaches significantly improve throughput, latency and resilience to skew that is prevalent in realistic workloads, because the switches can redirect traffic that would otherwise be arriving at congested backend nodes.

While this overall concept is promising, these systems can only handle unencrypted single-packet-sized objects of up to 1.5 to 9 K B (with jumbogram fabrics), because the switch data path must understand the application logic. Also, clients need to use custom UDP-based protocols and implement loss recovery and congestion control functionality by themselves, which is non-trivial especially when serving clients over the Internet. Other content-routing approaches support large objects [13, 40, 71], but do not fully support TCP and hence cannot support industry-standard TLS security or popular application-level protocols (e.g., HTTP and Amazon S3).

This paper presents Prism, a framework that enables the new object storage architecture that combines the flexibility of frontend proxies with the efficiencies and resilience of content-routing approaches, as illustrated in the right diagram in Figure 1. It transparently scales out to many clients and supports arbitrarily-sized objects, TCP and TLS, allowing applications to secure industry-standard protocols such as S3 over HTTPS.

A key concept of Prism is *repeatable* TCP connection hand-off, which allows a TCP connection to be re-homed to different machines over its lifetime. This enables the frontend to examine (even encrypted) requests without requiring it to then also relay object payloads (i.e., bulk data), addressing one of the main drawbacks of traditional proxy architectures (Section 2). At the expense of a fixed per-request cost, repeatable connection hand-off smoothly distributes the I/O, compute and network bandwidth usage across the backends, avoiding bottlenecks in the data path.

The novelty and viability of Prism are based on two other recent innovations. Prism is novel because of its new TCP hand-off protocol that conforms to the TCP state serialization feature available in modern Linux, which is also under development in FreeBSD [66], and overcomes the limitation of this feature (Section 4). Prism is now viable because of the availability of scalable, fine-grained state management techniques for programmable switches, such as SilkLoad [51] and FlowBlaze [8, 58], which enable Prism to control a large number of concurrent flows with fast switch rule management.

Prism has been implemented on Linux hosts and eBPF-based software switches, and for this paper was instantiated as an S3-compatible object store in order to demonstrate much better throughput and latency when compared to a frontend proxy architecture.

This paper makes three main contributions:

- Characterization of network and CPU utilization in the current proxy-based architecture with TLS (Section 2.3).
- A robust TCP hand-off protocol for commodity programmable switches and network stack (Section 4.2).
- Improvement of resource utilization in the replicated or sharded backend architecture (Section 5).

In the remainder of this paper, Section 2 reviews how object storage systems work and characterizes their performance. Section 3 describes our high level approach and design challenges. Section 4 details the design and implementation of Prism, including its connection hand-off protocol and software stack. Section 5 evaluates Prism, and Section 6 discusses implications of our work. Section 7 describes related work. The paper concludes with Section 8.

2 Background and Problem

The primary focus of this paper is commercial object storage systems. This section describes the concepts behind such systems, then characterizes their performance.

2.1 Motivation: Object Storage Systems

Object storage systems serve huge amounts of data, both when instantiated as public cloud services, such as Amazon S3, Azure Storage, Dropbox and others, or as private installations, such as NetApp StorageGRID or Dell/EMC ECS. Cloud storage systems are also being deployed in edge clouds [4, 43], which are smaller but closer to clients, compared to hyperscalar public clouds, but can still generate a terabit of data per second [4]. Scale-out object storage systems are also used in many other scenarios. For example, OpenStack, a popular multi-tenant cloud platform, uses them as a primary data repository [65], and optionally supports bandwidth isolation and fine-grained filtering [22, 23]. IBM has deployed them to build a scale-out Docker registry that maintains Docker images and other data [3].

A common design pattern for such object storage systems uses a set of client-facing *frontend* machines that arbitrate access to a set of *backend* storage machines. The frontend typically does not persistently store any data, but may in some instantiations locally cache some objects. Clients that connect to the storage system, over the wider Internet or from within an enterprise or datacenter network, will be routed to one of the frontend machines via DNS round-robin or L4 load-balancers. The designated frontend machine then acts as a proxy for the set of storage backends. The role of the proxy includes application-level firewalling the internals of the storage system from the outside, possibly TCP and/or TLS termination, client and/or request authentication and authorization, in addition to relaying requests and responses.

Once the TCP connection is established, the network traffic consists mostly of bulk data transfer between client and the backend servers, relayed by a frontend machine. Consequently, a frontend machine spends the majority of their resources relaying traffic between clients and backends. The protocols used by clients are usually RESTful, reusing TCP connections for many individual storage transactions that can be served by the different backends. The frontends are responsible for concurrently handling many clients, which can lead to congestion at the client-facing links [2]. Modern storage devices, such as NVMe SSD and persistent memory, further stress frontend machines, because unlike slow spinning disks [6, 54] these devices do not constrain the networking throughput.

2.2 System Model and Components

In summary, the frontend machine in scale-out object storage systems:

1. terminates client TCP and/or TLS connections
2. receives requests that contain a target object identifier, e.g., a key or URL
3. redirects the request to a suitably chosen backend
4. forwards request data from the client to the backend
5. forwards response data from the backend to the client

Subsequent requests over an already-established connection can be directed to different backends. This re-homing of the connection incurs a cost that is made up of several components, which we will review here to better understand the measured end-to-end performance in Section 2.3.

Data movement: A proxy relays data between two TCP sockets issuing two system calls: `read()` on one socket and `write()` on another, each copying data to and from the kernel. The costs of these system calls increase with the number of bytes read or written. When the data is very small (a few hundreds of bytes or smaller), the fixed *per-call* costs, i.e., the context switch overhead (several tens of ns) dominates the total cost—the cost of moving the data is negligible. For large data sizes, the total cost is dominated by the *per-byte* cost of moving the data. Proposed optimizations include TCP Splice [48, 63] and `tproxy` in recent Linux kernels, which move data between two TCP sockets within the kernel by swapping socket buffer pointers, without actually copying any data. However, these approaches struggle to support encryption and other more complex application logic.

Data encryption: For confidentiality and authentication, TLS has become the standard in today's Internet and datacenters. Therefore, offloading techniques for TLS have attracted considerable attention. Some TLS libraries can take advantage of hardware acceleration for various ciphers (e.g., AES-NI CPU instructions). In-kernel TLS support was recently added to Linux and FreeBSD to use cryptographic engines available on some NICs. Nevertheless, encrypting traffic comes at significant *per-byte* processing costs, similar to the data movement costs.

Application logic: Proxies typically perform application-level processing when relaying data. For example, they look up a key or URL embedded in the request data to select a suitable backend for the target value or object, or they may scan data to filter out particular requests. Since such information is typically included in the application-level protocol headers, the costs of such application logic usually does not increase with the size of transferred object. (Although it can if this information is contained in the request payload.)

Network stack: Modern kernel TCP/IP stacks can send and receive bulk data at the rate of tens of Gb/s by utilizing NIC offload features (in particular, checksum, TCP segmentation and large receive offloads). TCP and especially TLS connection establishment is an expensive operation due to needing multiple network round-trips, and the required updates to shared resources in the kernel impair multi-core parallelism [57]. However, modern application protocols are usually already designed to maximize connection reuse to amortize these costs. Hence, this paper does not concern itself with application protocols that have high connection-open rates; various improvements that are complementary to our work have already been proposed for high-rate connection openings [46, 57] and small data transfers [24, 36].

Network topology: In rack-scale storage in large datacenters [44, 53] or edge clouds such as used for content delivery networks (CDNs) [4], operators wire servers uniformly so that they can assign the role of servers (e.g., frontend and backend) flexibly depending on the node failure or service demands, as it is very expensive to rewrite cables [73] or the physical space is at a premium in the edge clouds [4]. Each rack consists of one or more top-of-rack (ToR) switches equipped with high bandwidth uplinks. These uplinks are a Clos [16] network fabric in datacenters and Internet exchanges in CDNs, respectively. In such a deployment, individual machines cannot be "scaled up" to create additional proxying capacity.

2.3 End-to-End Performance

To characterize performance of the proxy-based systems, we imitate the aforementioned network topology using a six logical node cluster, each of which connects to a cluster switch over a 10 Gb/s link, and a client machine that is connected to the cluster switch over a 40 Gb/s link that serves as a high bandwidth uplink (see Section 5.1 for hardware details). We install `nginx`, a high-performance, popular web server, to every server cluster node; one acts as a frontend (also generally called reverse proxy) and the other five act as backends. The client node runs `wrk` HTTP benchmark tool to generate requests to retrieve objects from the server cluster.

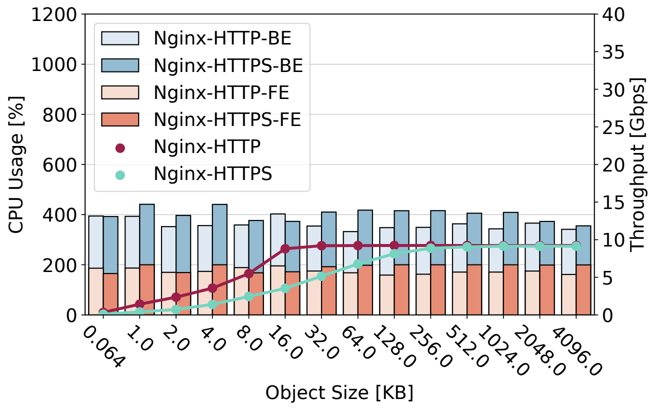


Figure 2: **Throughput and CPU usage of an `nginx` cluster.** One node acts as a proxy and the other five act as backends. Every node connects to a switch with a 10 Gb/s link. The switch connects to a client machine with a 40 Gb/s link (See Figure 8 for illustration). CPU usage can be at most 1200 % due to six dual-core servers.

Figure 2 plots throughput measured by the client and CPU utilization monitored at the servers. The requested object sizes vary, from 64 B to 4 MB. Although the real object storage systems handle a wider range of object sizes (e.g., hundreds of KB to a few MB for photos, and tens or hundreds of MB for videos, deep learning models and VM images), we select the range that characterizes the performance of the system.

With unencrypted HTTP, the frontend ("Nginx-HTTP") can serve up to 9.2 Gb/s, which is close to the 10 Gb/s line rate, taking into account protocol header and framing overheads. At the same time, the frontend CPU resources ("Nginx-HTTP-FE") are also fully utilized. When we allocate one more CPU core to the frontend (not plotted), the network bandwidth becomes the bottleneck (i.e., it results in idle CPU cycles).

When using HTTPS, the frontend ("Nginx-HTTPS") is able to serve only up to 9.1 Gb/s ("Nginx-HTTPS-FE"). Further, it requires larger objects to achieve that throughput than HTTP cases. This performance reduction is due to performing TLS cryptographic operations at the frontend that acts as a proxy, which fully utilizes its CPU resources.

These experiments confirm that either the fabric attachment bandwidth or the CPU resources of the frontend proxy become the bottleneck for this workload, depending on the hardware setup and the use of TLS. Since the Internet-facing capacity of the switch and backend CPU resources are left underutilized in these experiments, ideally the backends would circumvent the proxy and the switch would forward data directly between clients and backends. The Prism architecture enables this design.

3 Approach and Challenges

These observations confirm the need for reducing CPU usage at the frontend and increasing network utilization at the backends and switch uplink. Is it possible to exclude the frontend from the end-to-end data path for the majority of a transaction, while allowing it to perform its necessary tasks? We will show that it indeed is possible, but that doing so requires a different request-redirection approach than traditionally used. In this section we describe the high-level approach and highlight the main challenges, then describe our resulting design of Prism in Section 4.

3.1 Request-Granularity Redirection

The fundamental problem with a proxy architecture is that all traffic is mediated by the frontend; it relays all traffic between clients and backends. If the ToR switch itself could be instructed to relay data between clients and backends, that forwarding would happen at the faster core network speeds, and—more importantly—eliminate the traditional frontend participation in bulk data relaying. In other words, the frontend could focus on the control-plane aspects of relaying, and the fabric would focus on the data-plane aspects, which optimizes their relative strengths. A frontend confined to the control plane would have a great deal more network and CPU bandwidth available to support more clients per machine, and would thus reduce the overall number of frontend machines needed to support a given client population, making the service more cost effective.

Specialized examples if such a general architecture have been realized in the narrow domain of small-object key-value stores, such as SwitchKV [45], NetCache [37] and Pegasus [44]. However, these systems handle only single-packet-sized transactions and require clients to use custom, unencrypted protocols. It should be noted that for large objects, it is not trivial to implement congestion control and loss recovery that are able to cope with various Internet conditions, such as tail loss [9], incast [1] and phase effects [17].

General, commercially viable object storage systems of course need to support objects larger than a single packet. They also need to be able to secure their client communications with industry standard protocols such as TLS. One example is Amazon S3, the de facto industry standard for object storage, which runs over HTTPS, i.e., TCP and TLS. For such a protocol, *after* the frontend has processed application-level connection setup (e.g., user authentication), it is hence necessary to migrate the entire TCP and TLS connection state to one of the storage backends and have the fabric switches redirect traffic to it—based on *flow-level* information rather than application-level information.

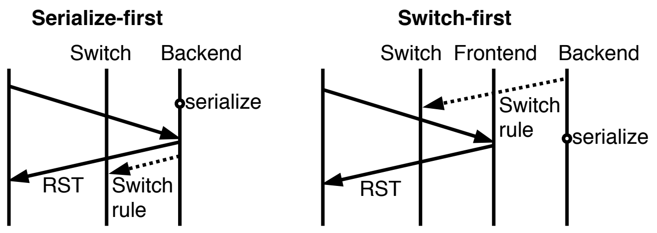


Figure 3: **Breakages with naive TCP hand-off designs.** Leaked packets trigger a connection reset (Section 3.2).

3.2 TCP Hand-Off

Basic TCP hand-off provides a starting point for the request-granularity TCP redirection of Prism. Although TCP hand-off was already explored over twenty years ago [5, 55] based on a custom TCP stack, it has not seen much real-world deployment or open source availability. However, TCP connection serialization, one of the essential features to enable TCP hand-off, was added to the Linux kernel in 2012 [12]. The identical feature is also under development in FreeBSD [66]. Therefore, designing a new TCP hand-off protocol based on this feature could ease the deployment of Prism approach.

When TCP connection state is serialized and then migrated to another machine, it is essential to carefully coordinate the updates to the necessary fabric switch rules, so that no packets "leak" to machines that do not hold the required state—such leakage would generate TCP RST (reset) messages, impacting client operation. Figure 3 depicts two example scenarios where such a connection reset occurs, because the hand-off protocol is incorrectly designed. On the left, an already-migrated connection receives a packet at the original machine, because the switch does not yet to redirect packets to the new target another server. On the right, the switch begins forwarding packets to the new target before the connection has been migrated there. These scenarios can happen with the hand-off protocol designed in the past [5]. The Prism migration protocol avoids any such problems, using the two-phase hand-off protocol described in Section 4.2.

Also, a fabric-based connection hand-off raises concerns about latency and scalability, because it requires manipulating fabric switch rules on a per-request basis (or at least every time the connection migrates). Previous studies imply that this might pose a significant hurdle: In 2014, Liu et al. [47] report that configuring a hardware switch can take hundreds of µs, meaning that for short backend transactions, the hand-off cost could significantly increase the overall request processing cost. In 2011, Yu et al. [69] show maintaining fine-grained flow state in hardware switches to be infeasible because of limited on-chip memory.

However, more recent work has addressed some of these constraints. SilkLoad [51] and FlowBlaze [8, 58] store state for millions of flows in hardware switches; the latter inserts hardware switch rules in a few µs. The Prism hand-off protocol incorporates these more recent observations.

4 Design

Based on the challenges above, the goals of Prism are to (1) design an efficient connection hand-off protocol that works for both TCP and TLS without breaking client sessions, and (2) to build a software stack that implements the protocol and provides a suitably abstract API to applications. We start with describing what an end-to-end data TCP transfer looks like, using the example of a read request that is received by a frontend and served by a backend. We then detail our connection hand-off protocol, which performs a two-phase switch configuration, and our software stack, which ensures correct kernel- and application-level operations.

4.1 Prism in Action

Figure 4 illustrates the Prism hand-off protocol in a packet sequence diagram. Solid arrows indicate packets sent on the TCP connection; dashed lines indicate Prism control messages between the frontend, switch and backends.

Establish connection: As illustrated in Figure 4, a client opens a TCP connection with a Prism frontend server, optionally followed by a TLS handshake.

Parse request: The client begins a transaction by sending a request, which the frontend receives and parses. When the frontend determines that it has received the entire request, it consults the metadata it maintains about the backend servers to select one to handle the request.

Hand-off request to backend: The frontend serializes the TCP connection and TLS session state. TCP state includes ports, sequence and ACK numbers and the TCP options negotiated for both directions of the connection; the TLS state includes the exchanged shared secrets. The frontend then contacts the chosen backend and passes the serialized states, the client IP address, and the client request, so that the backend can take over the connection and serve the request.

The backend instructs the Prism switch to rewrite the destination IP address of packets sent from the client to that of the chosen backend server, and to rewrite the source IP address of packets sent from that backend server to the client to that of the frontend. The switch also rewrites the destination or source MAC address, if the client resides in the same broadcast domain. The consequence of this is that any subsequent packets on this connection will be exchanged directly between the client and the backend, with the switch performing the required rewriting. Since the inserted rules only affect a single TCP connection, other connections, either

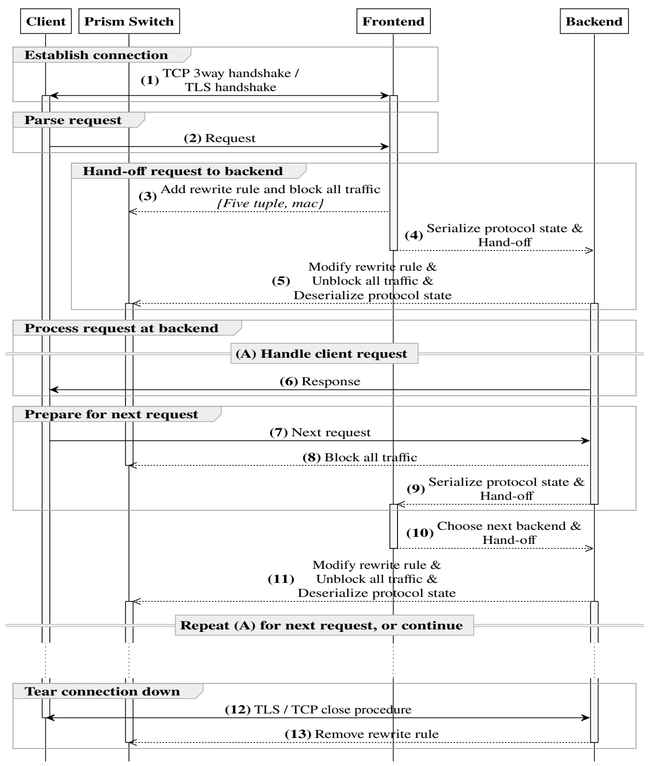

Client
Prism Switch
Frontend
Backend
Establish connection
(1) TCP 3way handshake / TLS handshake
Parse request
(2) Request
Hand-off request to backend
(3) Add rewrite rule and block all traffic {Five tuple, mac}
(4) Serialize protocol state & Hand-off
(5) Modify rewrite rule & Unblock all traffic & Deserialize protocol state
Process request at backend
(A) Handle client request
(6) Response
Prepare for next request
(7) Next request
(8) Block all traffic
(9) Serialize protocol state & Hand-off
(10) Choose next backend & Hand-off
(11) Modify rewrite rule & Unblock all traffic & Deserialize protocol state
Repeat (A) for next request, or continue
Tear connection down
(12) TLS / TCP close procedure
(13) Remove rewrite rule

Figure 4: **End-to-end Prism operation.** Solid arrows indicate TCP packets, dashed ones control messages. Step (8)–(11) indicates our two-phase hand-off protocol described in Section 4.2.

to the same frontend or other destinations, remain unaffected. We describe detailed procedures later in this section.

The backend then de-serializes the TCP and TLS state by instantiating a TCP socket based on the information in the serialized connection state and its local IP address (i.e., not frontend's). Because of the active switch rules, the client sees the traffic coming from this backend as if it was coming from the original frontend.

Process request at backend: The backend serves the client, sending back the response over the migrated connection.

Prepare for next request: After a transaction has completed, the backend may return the connection to the frontend and remove the corresponding switch rules, if it wishes subsequent requests on the same connection to be handled by the original frontend. The backend may parse the next request by itself and hand off the request to another backend.

Tear connection down: When the client or the backend itself closes the connection, the backend withdraws the corresponding switch rules.

4.2 Two-Phase Hand-Off Protocol

As shown in Figure 3, a deficient TCP hand-off protocol design breaks client connections. We therefore develop a

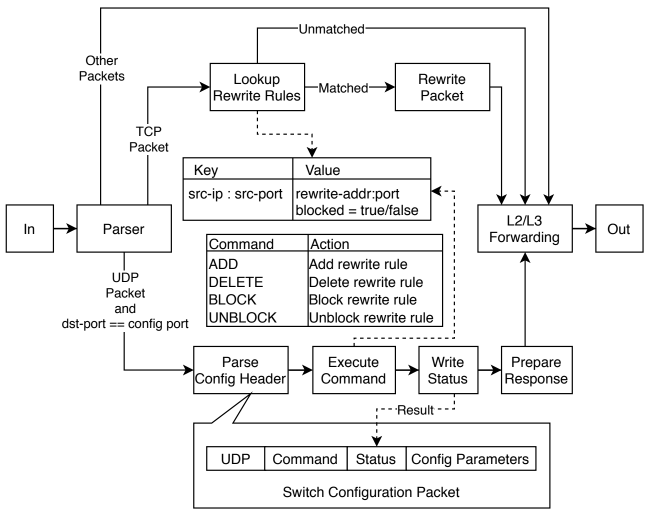


Figure 5: **Prism switch logic.** Custom UDP packets update switch rules.

new connection hand-off protocol that works with the TCP serialization feature available in Linux.

One "hack" would be to drop reset segments sent on a connection under migration with host firewall rules. However, this design requires maintaining flow steering state across the switch fabric *and* the servers, which complicates failure handling. We thus reject this design option.

Our solution instead employs a two-phase switch configuration. First, the host instructs the fabric switch to *drop* all traffic that belongs to the connection being migrated. This prevents this connection from receiving any further packets that might then lead to RSTs. It should be noted that this does not affect performance, because what may be dropped are only unusual packets, such as spurious retransmissions.

Then, a machine serializes a TCP connection and its TLS session and hands this serialized state off to another machine. The target machine then restores the TCP and TLS state. Finally, the target machine sends three commands to the switch atomically. The first one inserts a new rule that rewrites the source IP address of outgoing packets sent from the target machine. The second updates the existing switch rule to redirect any inbound packets to the target machine (instead of the original one). The third removes the drop rule. This two-phase hand-off procedure is depicted over step (8)–(11) in Figure 4.

Prism inserts or withdraws switch rules over a simple, stateless UDP control protocol that triggers in-switch rule manipulation without control-plane involvement. The switch logic that enables the two-phase connection hand-off is illustrated in Figure 5. This protocol implements a simple timeout-based retransmission mechanism, because we assume the communication over the shared links with client data traffic, which can be congested.

4.3 Stack and API

The hand-off protocol described above dictates that many individual commands and application I/O, which runs asynchronously, be executed in coordination with each other. Thus, we need a *software stack* that ensures the correct system state transitions, rather than just API extensions. This stack adds a loadable kernel module that allows applications to detect completed connection removal, an event-based execution engine that drives both the hand-off protocol and application I/O, and high-level APIs for applications to read and write data, and open, close and redirect connections. Figure 6 illustrates the Prism stack; the rest of this section details key components.

TCP state tracking. Before the withdrawal of a switch rule that rewrites the source IP address, the in-kernel connection state must have been freed *completely* to ensure the connection does not transmit any further packets. Unfortunately, this happens silently, long after an application closes a socket. Since the kernel does not notify applications of such events, we implemented a new kernel module to do so, using the Linux `eventfd` framework and socket destructor ("Conn. dtor" in Figure 6). This approach is suitable because the event loop component in the stack, described later, can monitor connection removal events together with any other events, such as new data read from the kernel and requests issued by the application. A similar method is possible also in FreeBSD.

TCP and TLS state serialization. Prism relies on the Linux `TCP_REPAIR` feature [12] to serialize TCP connection state. Based on the option parameter, `getsockopt()` serializes send and receive buffer data, sequence and ACK numbers and negotiated TCP options, which are restored using `setsockopt()` with the same option name. Prism uses the `tlse` library for TLS handshake and serialization, but for the data path, it uses the in-kernel TLS stack of recent Linux kernels, in order to benefit from future hardware offload support. We implemented a new `getsockopt()` option to retrieve the in-kernel TLS state, and upstreamed it to the mainline Linux kernel.

Switch communication. The stack is in charge of communication with the switch using the custom UDP-based packets (Section 4.2). It schedules and sends switch rule update commands to the switch when the application requests connection hand-off or restoration, and waits for the response that includes a status code.

Event loop. Because of the need to perform migration operations for application sockets, Prism discourages applications from using their sockets directly. The stack needs to coordinate and execute these operations for multiple file descriptors in parallel. Therefore, Prism augments `libuv`, a popular event-based I/O library that hides low-level system calls, like `epoll`, `kqueue`, `read` and `write`, from applications. An application associates its own callbacks with events, such as new connection establishment or network or file descriptor readiness for I/O. Prism extends `libuv` to allow applications

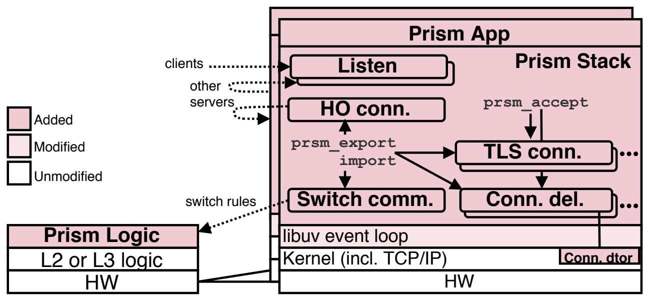


Figure 6: **Prism stack in the host (right) and switch (left).** Rounded rectangles represent file descriptors, and monospaced font corresponds to the calls used in Figure 7.

to associate their callbacks on top of TLS connections, and to coordinate the application requests and its hand-off protocol, including TCP state tracking and switch communication described above.

API. The programming model of Prism is based on `libuv` with two new methods to export or import migrating connections, and a TLS connection abstraction. Figure 7 implements the frontend and backend roles of the example object store, which statically partitions data across the servers by key, and covers the vast majority of Prism APIs. The `main()` function initiates the event loop with two TCP ports to monitor: one for client requests and the other for hand-offs from other servers. `on_accept()` runs upon establishment of a client TCP connection and initiates the TLS handshake via `uv_tls_init()`. When new client data arrives, it is decrypted and `on_read()` is executed. `who_has_val()` identifies whether the current server hosts the requested content, based on its key. The server either redirects the request to another server using `prsm_export()`, or returns the content using `uv_write()`, which schedules transmission in the event loop.

Although the use of these APIs ensures the correct hand-off state transitions, we do not prevent applications from the use of regular socket APIs. A regular application based on an `epoll` event loop would monitor three additional types of file discriptors: one for the terminated connection notification, another for switch communication and the other for connections to other servers. The application then needs to coordinate the events of these descriptors, for example, to serialize and hand-off a connection only after configuring the switch.

Switch. Many conventional frameworks, including eBPF and P4, are suitable to implement the Prism switching logic described in Figure 5. Although OpenFlow can also be used, this option incurs higher connection hand-off latency due to the access to remote control plane.

The Prism packet transformation logic can be implemented *on top of* an existing L2 switching or L3 forwarding logic without having to disrupt the existing network addressing, as shown in Figure 6. This also allows the Prism switch to be deployed alongside a non-programmable switch as a software switch, as we will show in our experiments.

```
1  on_read(client, buf) { // buf contains a decrypted request
2    const uint64_t key = get_key(buf);
3    who = who_has_val(key);
4    if (who != me)
5      prsm_export(client, who); // prsm_* are Prism methods
6    else {
7      (char *obj, int objlen) = get_objp(key);
8      req = new uv_buf_t(.base = obj, .len = objlen);
9      uv_write(req, client, on_write);
10   }
11 }
13 on_write(req) {
14   free(req->base);
15 }
17 on_accept(server) {
18   client = new uv_tls_t;
19   loop = server->loop;
20   uv_tls_init(loop, client); // uv_* are libuv
       objects/subclasses
21   prsm_accept(server, client);
22   uv_read_start(client, on_read);
23 }
25 main() {
26   uv_tls_t server, internal; // extend uv_tcp_t
27   loop = uv_default_loop();
28   uv_tls_init(loop, &server);
29   uv_tcp_bind(&server, "0.0.0.0:50000");
30   uv_tcp_bind(&internal, "0.0.0.0:60000");
31   uv_listen(&server, on_accept);
32   uv_listen(&internal, prsm_import);
33   uv_run(loop);
34 }
```

Figure 7: **Prism application pseudo code.** Prism extends `libuv`. Server returns requested object (line 9) if present, otherwise hands off request to actual custodian (line 5). The code thus serve the role of both frontend and backend.

4.4 Limitations

Concurrent requests. When a frontend or backend receives parallel requests being handled by different backends in the same TCP connection, it must serialize these requests using one of the following options. The first option is for a backend to simply block any arriving subsequent requests. If these requests need to be processed by different backends, the server hands off the connection after processing its current request. The second option is for the backend to send a TCP "zero window" advertisement to the client. This method turns out to be rather complex, because the backend must do so *before* TCP acknowledges received data, which may already have contained a subsequent request.

To maximize the performance, it is ideally the responsibility of the application-level protocol to prevent the client from issuing another request that might need to be handled by a different backend before an ongoing transaction on the same connection has completed. Traditional proxies can process parallel requests faster than Prism, if the frontend has enough attachment network bandwidth and spare CPU cycles to aggregate the responses from multiple backends; we leave this analysis as future work.

Small transfers. For a small-message transactional workload, i.e., where requests and responses fit into a few TCP packets, Prism may not be a good solution. In such cases, the overheads—switch configuration and connection hand-off—

cannot be sufficiently amortized. We analyze this trade-off in Section 5, which shows that 8 K B (i.e., 6 packets) is sufficient to amortize these overheads.

4.5 Implementation

The Prism server stack consists of 2193 Lines-of-Code (LoC): 612 LoC for TCP and TLS state serialization, 255 LoC for the switch configuration protocol, 167 LoC for active connection tracking, 130 LoC for the loadable kernel module to detect TCP connection removal and 1029 LoC for integration of these components. We modify a single line of `libuv`, and port `tlse` for our TLS abstraction.

We also implement a high performance software switch that makes up the Prism logic in Figure 5. To run the same code in hardware in the future, and to prevent the system from unexpected crash caused by software bug that affects many servers, we implement an eBPF execution environment as a switching logic module of mSwitch [29], a scalable, modular software switch that runs in the kernel. eBPF is popular these days and known that some switch vendors will support hardware offloading of eBPF processing. This software switch never becomes a bottleneck throughout the experiments in the next section.

Our source code is publicly available at `https://github.com/YutaroHayakawa/Prism-HTTP`.

5 Evaluation

This section evaluates Prism and reports the following main results:

- Prism improves throughput by a factor of up to 3.7 (with HTTP) and 3.4 (with HTTPS), utilizing the switch uplink and backend CPU resources efficiently.
- Prism's throughput increases with the number of backends due to the very light remaining load at the frontend, in terms of both network bandwidth and CPU usage.
- Prism improves object retrieval latency by up to 74 % and 96 % in the 50th and 90th percentile, respectively.
- Prism's connection hand-off latency is 232 µs, which is a win when transferring at least 2 K B with HTTP or 16 K B with HTTPS.
- Prism can be used to build object storage systems with partitioned or replicated backends.

5.1 Experiment Setup

Hardware and OS: Figure 8 depicts the testbed setup used for the experiments. Each machine has a quad-core Xeon E-1231v3 CPU clocked at 3.4 GHz, 16 GB of RAM and a dual-port Intel X540-T2 10 Gb/s NIC, running Linux kernel 4.18. We partition it into two logical servers, dedicating one 10 Gb/s port and two CPU cores to each. The switch has a ten-core Xeon E5-2690v4 CPU clocked at 2.6 GHz, 64 GB

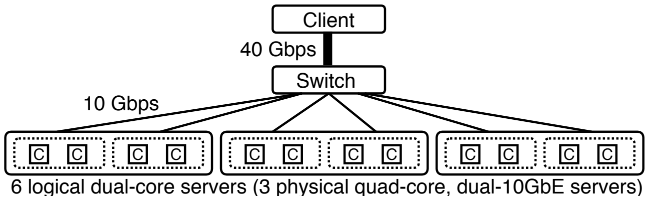

Client
40 Gbps
Switch
10 Gbps
C
6 logical dual-core servers (3 physical quad-core, dual-10GbE servers)

Figure 8: **Experimental topology.**

of RAM, three dual-port Intel 10 Gb/s NICs where each port connects to a logical server, and one Intel XL710 40 Gb/s NIC that connects to the client. We confirmed that this switch never becomes a bottleneck during the experiments. The client has two Xeon E5-2640v4 CPUs, 64 GB of RAM and the same 40 Gb/s NIC as the switch. Our network does not use jumbo frames. The connection hand-off traffic shares the same links with the data traffic.

Software: In the baseline experiment, all of the logical servers run a single `nginx` process. One server acts as a reverse proxy, the others as backends. In the Prism experiment, each logical server runs our custom application implemented on top of the Prism stack described in Section 4.3. Unlike the pseudo code in Figure 7, we use a static frontend setup where the same frontend always establishes incoming TCP connections and examines every request. Therefore, a backend that receives the next request always returns the connection to the frontend.

The communication protocol is always S3, either over HTTP or HTTPS (i.e., with or without TLS). The client continuously generates read or write requests using the `wrk` HTTP benchmark tool [21], instrumented to issue S3 requests (using the Lua scripting support) over 100 parallel persistent TCP connections per backend.

5.2 Link and CPU Utilization

Figure 9 plots the throughput and CPU utilization of Prism and the `nginx` baseline. Both systems redirect requests to the backends in round-robin fashion and each backend serves static, in-memory content.

For HTTP, Prism saturates the 40 Gb/s switch uplink (with the protocol headers and framing overheads) for object transfers of 256 K B and above, whereas `nginx` throughput is constrained by the 10 Gb/s attachment link capacity. Furthermore, Prism reduces the CPU utilization of the frontend by 23 to 53 % in comparison to `nginx`, which utilizes nearly the entire CPU to relay data, and uses the same amount of backend CPU resources as the Prism backends.

For HTTPS, Prism achieves a throughput of 31.4 Gb/s whereas the `nginx` baseline achieves only 9.2 Gb/s. Prism consequently leads to a much higher utilization of the backend CPUs, whereas `nginx` leaves around 75 % of backend CPU resources idle. Since the Prism frontend offloads and load-balances data encryption to the backends and avoids relaying data, its CPU usage reaches at most 70 %, which is spent on

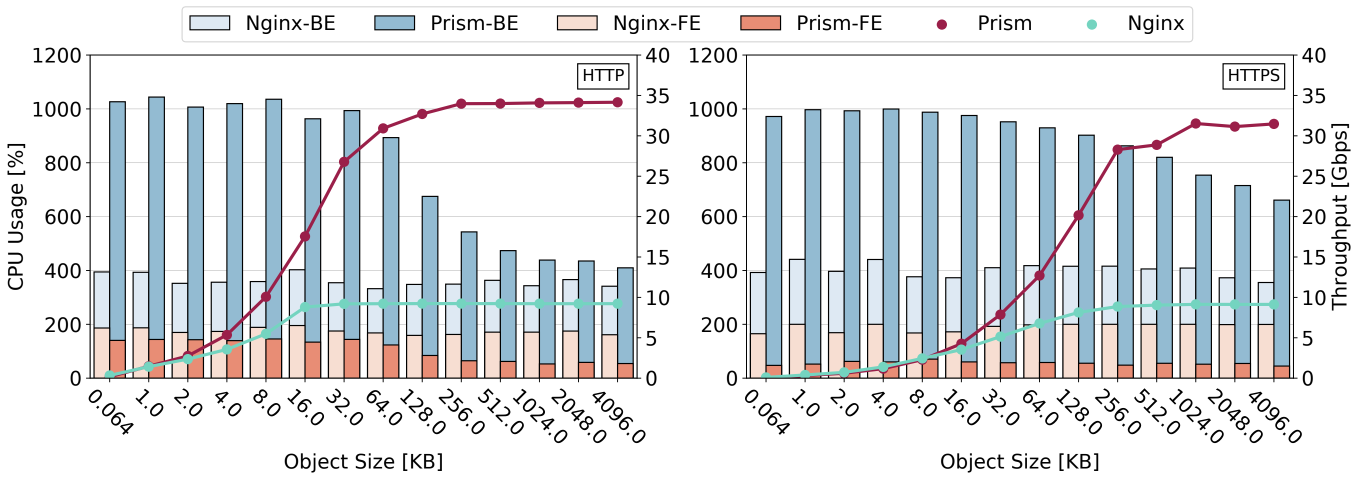


Figure 9: **Throughput and CPU usage with Prism and `nginx`.** FE and BE stand for frontend and backend, respectively. Prism utilizes backend CPU and network resources while keeping the frontend load low.

request redirection. These results imply that Prism would allow operators to provision a fewer number of frontend machines than traditional proxy architectures.

It may seem odd that overheads of the Prism frontend appear low even for 1 K B object sizes, but there is an obvious explanation. With `nginx`, all requests and responses over all 500 parallel connections are relayed by the frontend. Prism's frontend hands off connections to the backends instead of relaying requests, but it does *not* relay responses from the backends. Our results indicate that the advantage of not relying (nor encrypting) the responses at the frontend outweighs the connection hand-off costs.

Figure 10 plots the throughput and CPU utilization of Prism and baseline on a fewer number of backends for a subset of the object sizes, which are 16 , 256 and 512 K B. It confirms that throughputs of Prism have increased almost proportionally to the number of backends until they reach the limit of client processing capacity, which is lower with HTTPS (34 Gb/s and 28 Gb/s with HTTP and HTTPS, respectively) because of decryption and framing overheads.

Overall, Prism improves throughput by a factor of up to 3.7 without TLS, and 3.4 with TLS. It improves CPU utilization by factors of 2.4 and 2.6, respectively.

5.3 End-to-End Latency

Figure 11 plots the 50th and 90th percentile transaction latencies to retrieve 16, 256 or 512 K B objects (same experiments as Figure 10). These latencies decrease when adding more backends, except for the 90th percentile latencies with five backends and 256 or 512 K B objects, where the throughputs reach the maximum and queues start building up (see Figure 10). With `nginx`, those latencies do not change much with the number of backends, and are always higher than Prism, except for those 90th percentile latencies with 256 or 512 K B object sizes and five backends.

5.4 Connection Hand-Off Latency

Operation	Latency [µs]	Std. dev. [µs]
Block all traffic	22	13.1
Serialize TCP	7	1.1
Serialize TLS	5	1.1
Serialize HTTP	2	0.6
Close TCP socket	9	5.0
Hand-off (to frontend)	21	14.4
Hand-off (to backend)	21	14.4
Deserialize HTTP	3	1.5
Deserialize TLS	109	64.3
Deserialize TCP	11	1.0
Modify rewrite rule	22	9.4
Total	**232**	-

Table 1: **Connection hand-off latency breakdown.** The sequence starts from a backend returning the current connection to the frontend.

The largest concern about Prism is the connection hand-off overheads. As described in Section 4.2, the hand-off protocol takes two network round trips to the switch, in addition to the connection state transfer between frontend and backends.

Table 1 reports a breakdown of the latencies of a single connection hand-off cycle in which a backend returns a connection to a frontend, which then hands off the connection to another backend. The total is 232 µs. Each operation takes 21 to 22 µs if the network is involved, otherwise 2 to 11 µs, except for TLS deserialization that takes 109 µs and whose improvement is left as future work. The hand-off latency could be improved if the backend examined the next request after serving the current one, as in described in Figure 7, bypassing the frontend and saving 21 µs. Nevertheless, since Prism out-

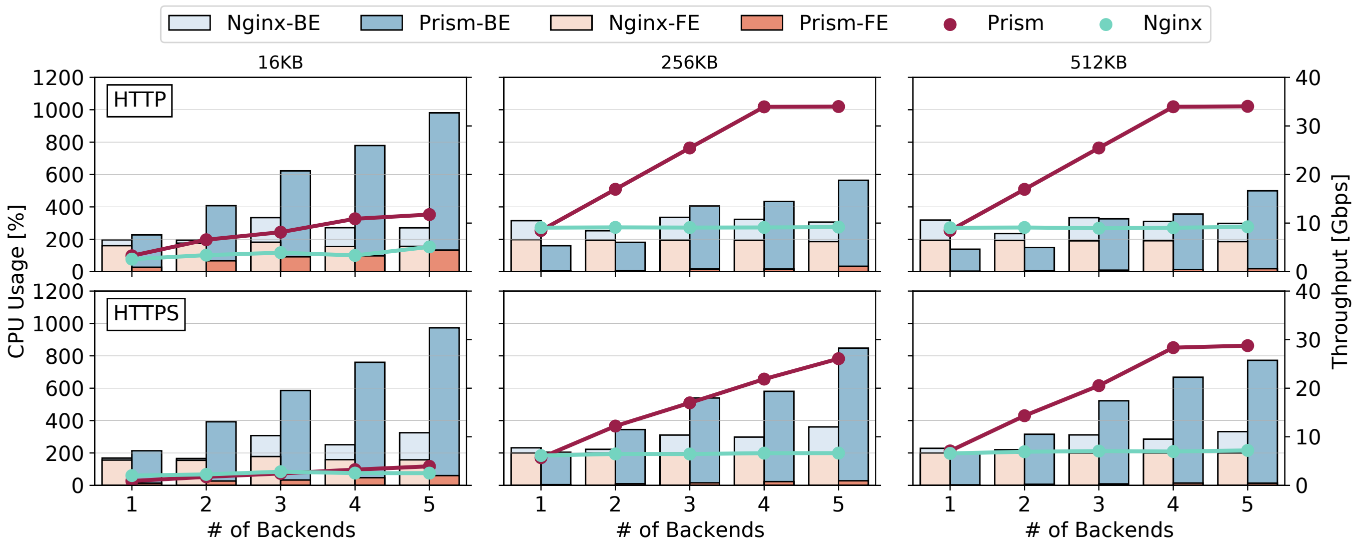


Figure 10: **Prism throughput with different numbers of backends.** The top and bottom row indicates HTTP and HTTPS, respectively. Prism linearly increases the throughput until the limit of client processing capacity.

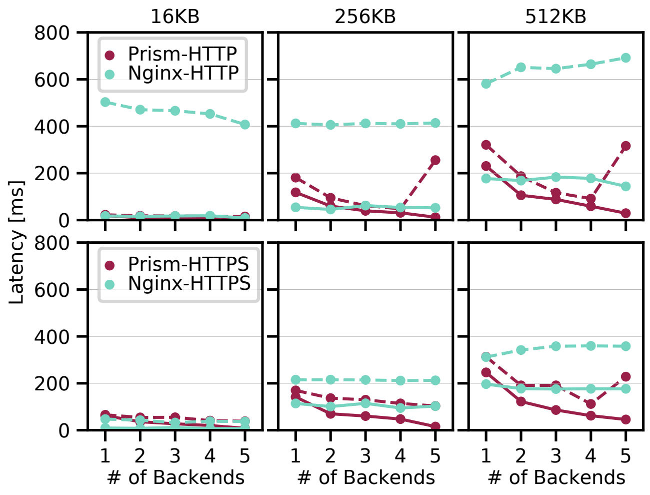


Figure 11: **Prism latency.** Solid and dashed lines indicate 50th and 90th percentile latencies, respectively.

performs the throughput of the traditional proxy architecture with 2 K B (HTTP) or 16 K B (HTTPS) of objects (Figure 9), we conclude that Prism outweighs the costs of the hand-off latency in various workloads.

5.5 Use Case

We implement two variants of an object storage system. The first *partitions* content across the backends, and the second *replicates* content at *all* backends. Thus, the partition variant writes a write request to one of the backends based on the key, but the replication variant does it to all the backends, before responding to the client. In contrast to the previous microbenchmarks in which a backend always serves the same in-memory content, these object storage variants use LevelDB where keys identify objects that are organized into a log structured merge tree. Thus, they include realistic storage stack overheads. We use a RAM disk for the storage medium, assuming faster-than-network NVM-style storage medium. The client protocol is again S3.

5.5.1 Partitioned Object Storage Backends

Figure 12 shows throughputs over two YCSB workloads, read-only and read-mostly that contains 5 % writes. We vary object sizes between 16 to 512 K B and request key skewness, which is 0.9 to 1.2 of Zipfian parameters, uniform distribution (least skewed) and requests that always ask for the same key (most "skewed").

We observe throughputs stay almost the same up to Zipfian 1.0 of skewness, and a slight, up to by 17 %, drop at Zipfian 1.2 that can be considered extremely skewed. We observe that even for an extremely skewed workload (Zipfian 1.2), throughput decays only by up to 17 % in comparison to the uniform distribution, demonstrating Prism's robustness against skewed workloads.

5.5.2 Replicated Object Storage Backends

Figure 13 shows throughputs with the same workloads but for the replicated backends, assuming fault tolerance and load balancing. We observe the same throughput regardless of skewness, as the frontend redirects requests in a round-robin fashion. Since every write is replicated to all the backends, throughput decreases with higher write rates, more so in comparison to the partitioned-backend cases. The lower overall rates compared to those cases can be attributed to larger active data that stress OS buffer caches.

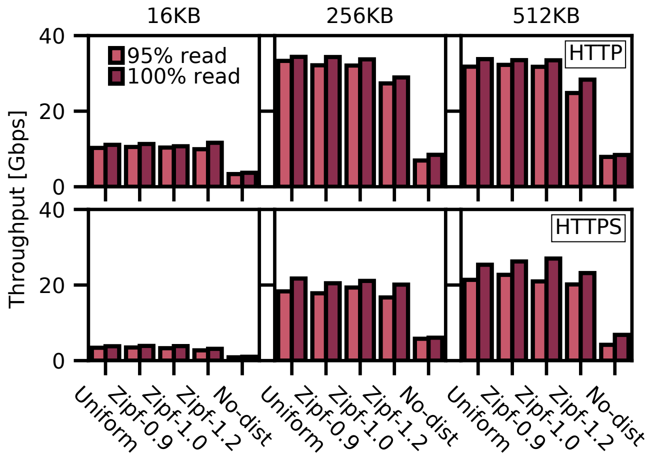


Figure 12: **Prism throughputs with partitioned backends.** Top and bottom rows plot results with HTTP and HTTPS, respectively. Prism preserves high throughputs even under skewed workloads.

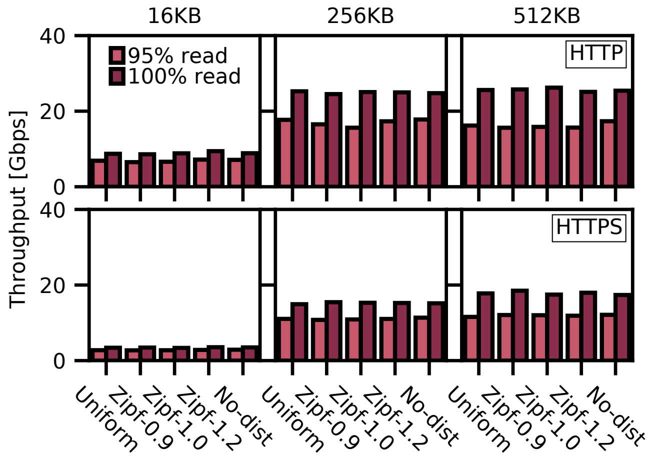


Figure 13: **Prism throughputs with replicated backends.** Throughputs are unaffected by key skewness due to evenly distributed requests.

Prism's ability to handle a partitioned key space and to balance loads across replicas indicates its feasibility for implementing sophisticated data layouts and replication algorithms, respectively.

6 Lessons Learned

Although TCP hand-off had been proposed multiple times at least since 1998, it has never been widely used. We initially attributed this to the lack of scalable flow-level programmable switches, but increasingly realized many other reasons. The mechanism to snapshot or instantiate TCP connections in any state, an essential feature, has been enabled in Linux in a rather inconvenient form, that is, packet transformations must be performed elsewhere, such as with a host firewall or at a switch. Moreover, the TCP stack needs to emit RST packets during the hand-off process to conform to the invariants that apply throughout the TCP implementation.

Alternative approaches that do not require programmable switches have been proposed by Snoeren et al. in 2000 [61, 62]. However, deploying new TCP options has been increasingly difficult over the last decade due to slow network stack evolution [30] and middlebox interferences [32].

These constraints resulted in difficulties for TCP hand-off overall, because it requires that many operations be performed atomically. This requirement prevents the use of a host firewall for source address rewriting, because otherwise manage flow state needs to be managed at both hosts and switches, requiring a complex coordination mechanism.

Building the end-system stack was also a great burden. However, since the TCP connection serialization has become available in the mainline kernel, systems like Prism can now be realized without modifications to the kernel; in fact, we still needed kernel modification, but the Linux community accepted the necessary changes, which are to implement a new API to access the in-kernel TLS state [26], to be included in the mainline kernel. We were able to implement the other kernel extension, which is the connection removal notification (Section 4.3), as a loadable kernel module. Moreover, fewer applications today call "low-level" socket APIs directly; many use higher-level networking libraries such as `libuv`, which we extend to enable the Prism stack (Section 4.3). These phenomena support the deployability of Prism.

Another problem is frequent switch rule updates. This has been problematic for older switches, because the rules must be updated via the switch control plane that runs slow CPUs. In fact, our initial prototype communicated with an RPC server running in the control plane, and it consequently suffered from high latencies of up to 828 µs, which requires 256 K B of transmitted data to amortize these hand-off costs. This led us to the use of a custom switch manipulation mechanism that updates the rules directly within the data plane. This becomes possible with newer hardware and software switches, such as P4 and eBPF, that have advanced programmability. Further, recent improvements of flow scalability in hardware switches [8, 51, 58] also supports the feasibility of this approach.

Further, Prism does not complicate layer 4 firewalls in the network or host, because Prism hosts never rely on host firewalls, nor "spoof" packet addresses, as the packet transformation happens in the switch. This means that host firewall rules can be configured based on not the address of the frontend but that of the individual host. Local firewall policies apply to the restored TCP connections, because Linux `netfilter` creates connection states when it sees any egress packets [11].

Last but not least, we believe our approach is feasible even in sharing switches, which has been a major concern in the use of programmable switches in many existing systems [50].

This is because we turn the feature of the content-based routing approaches into flow-level operations that can be easily isolated between tenants and machines; for example, the operator can limit address-modification rules inserted to the switch within the address range allocated to the tenant.

7 Related Work

Our previous short paper [27] introduced Prism's overall approach with a minimalistic proof-of-concept implementation. This paper significantly extends Prism by incorporating the ability to handle TLS-encrypted communication, a robust hand-off protocol, a much-improved software stack, a more mature implementation and an extensive evaluation.

TCP connection migration. The closest related work is a proposal by Aron et al. in 2000 [5] that proposes content-aware request distribution to the backend cluster, similar to Prism. However, their hand-off protocol can break client connections (Section 3.1) and does not support TLS (or SSL). TCP Migrate Options [61, 62] achieve TCP connection migration using TCP options instead of programmable switches, but have deployment problems (Section 6).

Proxy enhancements and L7 load balancers. TCP Splice (Section 2.2) has been improved by software [10, 59] or hardware-assisted [52, 72] approaches. Yoda [19] improves the fault tolerance of the proxy architecture. Squid [64], HAProxy [25] and Proxygen [60] are open source proxy implementations. Unlike Prism, these approaches do not eliminate the need of a frontend proxy to remain involved in bulk data relaying.

Content based routing. SwitchKV [45], Pegasus [44] and NetCache [37] as discussed in Section 3.1 eliminate frontends that mediate traffic between the client and the backend, by having programmable switches play the role of the frontend. They only support data that fits into a single packet over unencrypted custom UDP-based protocol. NICE [40] supports large data objects, but it relies on unencrypted UDP-based requests and TCP connections initiated by the server for reply. Therefore, it supports neither TLS nor industry-standard protocols such as S3. Strata [13] is a scale-out storage system with NFSv3. It breaks client connections and relies on reconnection to resume the NFS session after change of the storage backend. NetKV [70] is an application-level load balancer for memcached, but does not support TCP.

L4 load balancers. Maglev [14] and Ananta [56] are software load-balancers implemented in commodity servers. Duet [20], Rubik [18] and Faild [4] are similar, but partially leverage standard hardware switches for improved performance. Unlike Prism, none of these approaches support request-granularity redirection. L4 load balancers are used to distribute traffic between multiple frontends of the Prism or proxy architecture.

Flexible packet processing. The Prism frontend might resemble a packet forwarding system implemented as middleboxes, but it differs in that it does not forward packets but hands off established TCP connections. Our software implementation on a programmable soft-switch uses mSwitch [29] for performance and flexibility, and eBPF for protection, but it can be other flexible packet processing frameworks. Recent scalable hardware packet processing systems, such as SilkRoad [51] and FlowBlaze [8, 58], enable Prism to scale to a large number of flows and reduce the latency of switch rule updates. Some vendors are implementing eBPF hardware offloading [39], which accelerates our switch logic (Figure 5).

Caching, sharding and replication algorithms. Many object placement algorithms for storage systems have been proposed [7, 15, 40, 44]. Prism is a framework to implement object storage systems with these algorithms without worrying about communication with clients. In Section 5, we experimentally demonstrated that Prism can be used to implement replicated backends for load balancing and sharded ones for capacity scaling. More sophisticated algorithms, such as selective replication [44], will further improve the storage utilization and performance.

High performance host storage stack with TCP/IP. This class of work, such as Diskmap [49], ReFlex [42] and i10 [33] for NVMe, and Decibel [53] and PASTE [31] for persistent memory, could enhance Prism's backends, and benefit from Prism, because they encrypt or push data at higher rates than traditional host storage stack.

8 Conclusion

As faster storage devices push data to CPUs and networks at higher rates, it is important to scale-out these resources. We built Prism, which combines the performance and resilience of content-based routing approaches with the generality and flexibility of a conventional proxy architecture. We demonstrated that Prism can be used to build object storage systems for the industry-standard S3 protocol over TCP and TLS, and to implement partitioned and replicated backends for capacity scaling and load balancing. Prism is based on a connection hand-off technique that has been proposed in the past, but we redesigned it to address practical problems with the previous systems, taking into account modern technology and requirements across the network switches and today's OS kernels.

Future work will develop object storage systems with advanced data layout or load balancing algorithms, and further reduce the overheads of the TCP/TLS connection migration. Modern low-latency networking techniques based on efficient stacks [68] or RPC designs [38] are a perfect fit in this space.

Acknowledgments

We are grateful to anonymous NSDI reviewers and Z. Morley Mao, our shepherd, for their insightful comments.

References

[1] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. "Data Center TCP (DCTCP)". *Proc. ACM SIGCOMM*. 2010.

[2] A. Anwar, Y. Cheng, A. Gupta, and A. R. Butt. "Mos: Workload-aware elasticity for cloud object stores". *Proc. ACM HPDC*. 2016.

[3] A. Anwar, M. Mohamed, V. Tarasov, M. Littley, L. Rupprecht, Y. Cheng, N. Zhao, D. Skourtis, A. S. Warke, H. Ludwig, D. Hildebrand, and A. R. Butt. "Improving Docker Registry Design Based on Production Workload Analysis". *Proc. USENIX FAST*. 2018.

[4] J. T. Araujo, L. Saino, L. Buytenhek, and R. Landa. "Balancing on the Edge: Transport Affinity without Network State". *Proc. USENIX NSDI*. 2018.

[5] M. Aron, D. Sanders, P. Druschel, and W. Zwaenepoel. "Scalable content-aware request distribution in cluster-based network servers". *Proc. USENIX ATC*. 2000.

[6] D. Beaver, S. Kumar, H. C. Li, J. Sobel, P. Vajgel, et al. "Finding a Needle in Haystack: Facebook's Photo Storage." *Proc. USENIX OSDI*. 2010.

[7] N. Beckmann, H. Chen, and A. Cidon. "LHD: Improving Cache Hit Rate by Maximizing Hit Density". *Proc. USENIX NSDI*. Apr. 2018.

[8] G. Bianchi, M. Bonola, S. Pontarelli, D. Sanvito, A. Capone, and C. Cascone. "Open Packet Processor: a programmable architecture for wire speed platform-independent stateful in-network processing". *arXiv preprint arXiv:1605.01977*, 2016.

[9] P. Cheng, F. Ren, R. Shu, and C. Lin. "Catch the whole lot in an action: Rapid precise packet loss notification in data center". *Proc. USENIX NSDI*. 2014.

[10] A. Cohen, S. Rangarajan, and H. Slye. "On the Performance of TCP Splicing for URL-aware Redirection". *Proc. USENIX USITS*. Oct. 1999.

[11] *Conntrack tales - one thousand and one flows*. https://blog.cloudflare.com/conntrack-tales-one-thousand-and-one-flows/.

[12] J. Corbet. "TCP Connection Repair". *LWN.net*, May 2012.

[13] B. Cully, J. Wires, D. Meyer, K. Jamieson, K. Fraser, T. Deegan, D. Stodden, G. Lefebvre, D. Ferstay, and A. Warfield. "Strata: High-Performance Scalable Storage on Virtualized Non-volatile Memory". *Proc. USENIX FAST*. 2014.

[14] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein. "Maglev: A Fast and Reliable Software Network Load Balancer". *Proc. USENIX NSDI*. Mar. 2016.

[15] B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky. "Small Cache, Big Effect: Provable Load Balancing for Randomly Partitioned Cluster Services". *Proc. ACM SoCC*. 2011.

[16] M. Al-Fares, A. Loukissas, and A. Vahdat. "A Scalable, Commodity Data Center Network Architecture". *Proc. ACM SIGCOMM*. 2008.

[17] S. Floyd and V. Jacobson. "Traffic phase effects in packet-switched gateways". *SIGCOMM CCR*, 1991.

[18] R. Gandhi, Y. C. Hu, C.-k. Koh, H. Liu, and M. Zhang. "Rubik: Unlocking the Power of Locality and End-point Flexibility in Cloud Scale Load Balancing". *Proc. USENIX ATC*. Jul. 2015.

[19] R. Gandhi, Y. C. Hu, and M. Zhang. "Yoda: A Highly Available Layer-7 Load Balancer". *Proc. ACM EuroSys*. Apr. 2016.

[20] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang. "Duet: Cloud Scale Load Balancing with Hardware and Software". *Proc. ACM SIGCOMM*. Aug. 2014.

[21] W. Glozer. *Modern HTTP benchmarking tool*. https://github.com/wg/wrk.

[22] R. Gracia-Tinedo, J. é, E. Zamora, M. Sánchez-Artigas, P. Garcıa-López, Y. Moatti, and E. Rom. "Crystal: Software-Defined Storage for Multi-Tenant Object Stores". *Proc. USENIX FAST*. 2017.

[23] R. Gracia-Tinedo, J. é Sampé, and G. Parıs. *Crystal: Software-Defined Storage for OpenStack Swift*. GitHub. 2017.

[24] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy. "MegaPipe: A New Programming Interface for Scalable Network I/O." *Proc. USENIX OSDI*. 2012.

[25] HAProxy. *HAProxy - The Reliable, High Performance TCP/HTTP Load Balancer*. http://www.haproxy.org/.

[26] Y. Hayakawa. *Re: [PATCH net-next] net/tls: Implement getsockopt SOL_TLS TLS_RX*. https://www.mail-archive.com/netdev@vger.kernel.org/msg350523.html.

[27] Y. Hayakawa, L. Eggert, M. Honda, and D. Santry. "Prism: A Proxy Architecture for Datacenter Networks". *Proc. ACM SoCC*. 2017.

[28] T. Hoefler, R. B. Ross, and T. Roscoe. "Distributing the data plane for remote storage access". *Proc. ACM HotOS*. 2015.

[29] M. Honda, F. Huici, G. Lettieri, and L. Rizzo. “mSwitch: A Highly-Scalable, Modular Software Switch”. *Proc. ACM SOSR*. Jun. 2015.

[30] M. Honda, F. Huici, C. Raiciu, J. Araujo, and L. Rizzo. “Rekindling Network Protocol Innovation with User-level Stacks”. *ACM SIGCOMM CCR*, Apr. 2014.

[31] M. Honda, G. Lettieri, L. Eggert, and D. Santry. “PASTE: A Network Programming Interface for Non-Volatile Main Memory”. *Proc. USENIX NSDI*. 2018.

[32] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. “Is It Still Possible to Extend TCP?”: *Proc. ACM IMC*. Nov. 2011.

[33] J. Hwang, Q. Cai, A. Tang, and R. Agarwal. “TCP≈RDMA: CPU-efficient Remote Storage Access with i10”. *Proc. USENIX NSDI*. 2020.

[34] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson. “Basic Performance Measurements of the Intel Optane DC Persistent Memory Module”. *CoRR*, 2019. arXiv: 1903.05714.

[35] Y. Japan. *Dragon: A Distributed Object Storage at Yahoo! JAPAN*. WebDB Forum 2017. 2017.

[36] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. “mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems”. *Proc. USENIX NSDI*. Apr. 2014.

[37] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. “NetCache: Balancing Key-Value Stores with Fast In-Network Caching”. *Proc. ACM SOSP*. 2017.

[38] A. Kalia, M. Kaminsky, and D. G. Andersen. “Datacenter RPCs can be General and Fast”. *Proc. USENIX NSDI*. 2019.

[39] J. Kicinski and N. Viljoen. “Making Linux TCP Fast”. *The Technical Conference on Linux Networking (NETDEV 1.2)*. Nov. 2016.

[40] S. Al-Kiswany, S. Yang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. “Nice: Network-integrated cluster-efficient storage”. *Proc. ACM HPDC*. 2017.

[41] A. Klimovic, C. Kozyrakis, E. Thereska, B. John, and S. Kumar. “Flash Storage Disaggregation”. *Proc. ACM EuroSys*. 2016.

[42] A. Klimovic, H. Litz, and C. Kozyrakis. “ReFlex: Remote Flash≈Local Flash”. *Proc. ACM ASPLOS*. 2017.

[43] M. Körner, T. M. Runge, A. Panda, S. Ratnasamy, and S. Shenker. “Open carrier interface: An open source edge computing framework”. *Proc. ACM NEAT*. 2018.

[44] J. Li, J. Nelson, E. Michael, X. Jin, and D. R. Ports. “Pegasus: Tolerating Skewed Workloads in Distributed Storage with In-Network Coherence Directories”. *Proc. USENIX OSDI*. Nov. 2020.

[45] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman. “Be Fast, Cheap and in Control with SwitchKV”. *Proc. USENIX NSDI*. Mar. 2016.

[46] X. Lin, Y. Chen, X. Li, J. Mao, J. He, W. Xu, and Y. Shi. “Scalable Kernel TCP Design and Implementation for Short-Lived Connections”. *Proc. ACM ASPLOS*. 2016.

[47] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz. “zUpdate: Updating data center networks with zero loss”. *ACM SIGCOMM*. 2013.

[48] D. A. Maltz and P. Bhagwat. “TCP Splicing for Application Layer Proxy Performance”. *J. High Speed Netw.* Jan. 2000.

[49] I. Marinos, R. N. Watson, M. Handley, and R. R. Stewart. “Disk| Crypt| Net: rethinking the stack for high-performance video streaming”. *Proc. ACM SIGCOMM*. 2017.

[50] J. McCauley, A. Panda, A. Krishnamurthy, and S. Shenker. “Thoughts on load distribution and the role of programmable switches”. *SIGCOMM CCR*, 2019.

[51] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. “SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs”. *Proc. ACM SIGCOMM*. 2017.

[52] Y. Moon, S. Lee, M. A. Jamshed, and K. Park. “AccelTCP: Accelerating Network Applications with Stateful TCP Offloading”. *Proc. USENIX NSDI*. Feb. 2020.

[53] M. Nanavati, J. Wires, and A. Warfield. “Decibel: Isolation and Sharing in Disaggregated Rack-Scale Storage”. *Proc. USENIX NSDI*. 2017.

[54] E. B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue. “Flat Datacenter Storage”. *Proc. USENIX OSDI*. Oct. 2012.

[55] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. “Locality-aware Request Distribution in Cluster-based Network Servers”. *Proc. ACM ASPLOS*. 1998.

[56] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri. “Ananta: Cloud Scale Load Balancing”. *Proc. ACM SIGCOMM*. Aug. 2013.

[57] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris. “Improving network connection locality on multicore systems”. *Proc. ACM EuroSys*. 2012.

[58] S. Pontarelli, R. Bifulco, M. Bonola, C. Cascone, M. Spaziani, V. Bruschi, D. Sanvito, G. Siracusano, A. Capone, M. Honda, F. Huici, and G. Bianchi. “FlowBlaze: Stateful Packet Processing in Hardware”. *Proc. USENIX NSDI*. 2019.

[59] M. Rosu and D. Rosu. “Kernel Support for Faster Web Proxies”. *Proc. USENIX ATC*. Jun. 2003.

[60] B. Schlinker, I. Cunha, Y.-C. Chiu, S. Sundaresan, and E. Katz-Bassett. “Internet Performance from Facebook’s Edge”. *Proc. ACM IMC*. 2019.

[61] A. C. Snoeren, D. G. Andersen, and H. Balakrishnan. “Fine-grained Failover Using Connection Migration”. *Proc. USENIX USITS*. Mar. 2001.

[62] A. C. Snoeren and H. Balakrishnan. “An End-to-end Approach to Host Mobility”. *Proc. ACM MobiCom*. Aug. 2000.

[63] O. Spatscheck, J. S. Hansen, J. H. Hartman, and L. L. Peterson. “Optimizing TCP Forwarder Performance”. *IEEE/ACM Trans. Netw.* Apr. 2000.

[64] Squid. *Squid: Optimising Web Delivery*. http://www.squid-cache.org/.

[65] O. Stack. *Swift Object Store*. Open Stack Software. 2016.

[66] Y. Takagawa and K. Matsubara. “Yet Another Container Migration on FreeBSD”. *AsiaBSDCon*. 2019.

[67] P. Vajgel. *Needle in a haystack: efficient storage of billions of photos*. 2009.

[68] K. Yasukata, M. Honda, D. Santry, and L. Eggert. “StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs”. *Proc. USENIX ATC*. Jun. 2016.

[69] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. “Scalable Flow-Based Networking with DIFANE”. *Proc. ACM SIGCOMM*. 2010.

[70] W. Zhang, T. Wood, and J. Hwang. “NetKV: Scalable, Self-Managing, Load Balancing as a Network Function”. *Proc. IEEE ICAC*. Jul. 2016.

[71] W. Zhang, T. Wood, K. Ramakrishnan, and J. Hwang. “Smartswitch: Blurring the line between network infrastructure & cloud applications”. *Proc. USENIX HotCloud*. 2014.

[72] L. Zhao, Y. Luo, L. Bhuyan, and R. Iyer. “SpliceNP: A TCP Splicer Using a Network Processor”. *Proc. ACM/IEEE ANCS*. Oct. 2005.

[73] S. Zhao, R. Wang, J. Zhou, J. Ong, J. C. Mogul, and A. Vahdat. “Minimal rewiring: Efficient live expansion for clos data center networks”. *Proc. USENIX NSDI*. 2019.

Programming Network Stack for Middleboxes with Rubik

Hao Li[1], Changhao Wu[1,2], Guangda Sun[1], Peng Zhang[1], Danfeng Shan[1], Tian Pan[3], Chengchen Hu[4]

[1]*Xi'an Jiaotong University* [2]*Brown University*
[3]*Beijing University of Posts and Telecommunications* [4]*Xilinx Labs Asia Pacific*

Abstract

Middleboxes are becoming indispensable in modern networks. However, programming the network stack of middleboxes to support emerging transport protocols and flexible stack hierarchy is still a daunting task. To this end, we propose Rubik, a language that greatly facilitates the task of middlebox stack programming. Different from existing hand-written approaches, Rubik offers various high-level constructs for relieving the operators from dealing with massive native code, so that they can focus on specifying their processing intents. We show that using Rubik one can program the middlebox stack with minor effort, *e.g.*, 250 lines of code for a complete TCP/IP stack, which is a reduction of 2 orders of magnitude compared to the hand-written versions. To maintain a high performance, we conduct extensive optimizations at the middle- and back-end of the compiler. Experiments show that the stacks generated by Rubik outperform the mature hand-written stacks by at least 30% in throughput.

1 Introduction

Middleboxes are pervasively deployed in modern networks. In the middlebox, a low-level *network stack* (*e.g.*, TCP/IP) is responsible for parsing raw packets, and a set of high-level hooks (*e.g.*, HTTP dissector) process the parsed data for various purposes. There is a constant need for programming the network stack of middleboxes in order to accommodate different networks (*e.g.*, IEEE 802.11 [9] in WLAN), support new protocols (*e.g.*, QUIC [50]), realize customized functions (*e.g.*, P4 INT [5]), and capture new events (*e.g.*, the IP fragmentation [11]), *etc*.

By programming a middlebox stack, an operator Alice is mainly concerned with the following tasks. (1) *Writing new parsers.* It is common that a network needs to support a new protocol. Then, Alice needs to write a new parser to parse such traffic. (2) *Customizing stack hierarchy.* Another common need is to change the protocol layering, say to support encapsulation methods like IP-in-IP [1]. Then, Alice needs to re-organize the parsers. (3) *Adding new functions.* Finally, new functions may be requested from the network stack to meet diverse needs. For example, Alice may need to know when IP fragmentation happens, and modify an existing network stack to capture this event. In the following, we will show the difficulty of the above programming tasks.

The difficulty of writing new parsers. Currently, protocol parsers are written in low-level native code to ensure the high efficiency, which leads to a large number of lines of code (LOC) even for a single protocol, *e.g.*, ∼7K C LOC for TCP protocol parser in mOS [14]. Someone may argue that the TCP/IP is the de facto narrow waist for middlebox processing, so a general-purpose TCP/IP substrate is sufficient for extended programmability. However, many networks have their own customized transport layers (*e.g.*, QUIC [50]), and in those cases, Alice still needs to manually write new parsers.

The difficulty of building stack hierarchy. The data structures in current middlebox stacks are monolithic and closely coupled with standard stacks, making it difficult to reuse the existing protocol parsers for upgrading the stacks. For example, `libnids` [11] can parse Ethernet, IP, UDP and TCP protocols, but due to its TCP-specific data structure, it can hardly support the IP-in-IP stack, *i.e.*, ETH→IP→GRE→IP→TCP, although there is only one thin GRE parser need to be added. As a result, we have to modify 1022 LOC of `libnids` in our preliminary work to support the IP-in-IP stack, where most effort (815 LOC) is devoted to stack refactoring.

The difficulty of adding new functions. Since the network stack is closely coupled, adding a new function often needs a deep understanding of a huge code base. For example, if one wants to capture a low-level IP fragmentation event that is not supported in Zeek [24], she has to first read all the native code related to the IP protocol and the event callback module, which involves ∼2K LOC. Another scenario would be the feature pruning: *e.g.*, for writing a stateful firewall without the need to buffer the TCP segments, the operator has to go through considerable code to ensure the code deletion in a full-functional TCP stack will not produce other side effects.

Table 1: Existing approaches to program middlebox stack and their support of the three programming tasks.

Approaches	Protocol Parser	Stack Hierarchy	Stack Functions
Packet Parser (*e.g.*, P4 [18], VPP [23])	◐	◐	○
TCP-Specific Stacks (*e.g.*, mOS [44])	○	○	◐
NFV Frameworks (*e.g.*, NetBricks [58])	◐	●	◐

● : *Can be fully programmed with high-level abstractions, i.e., minor LOC*
◐ : *Partially supported or can be programmed with moderate LOC*
○ : *Not supported or can only be programmed with large amount of LOC*

Apart from the mature and fixed TCP stack libraries like mOS and `libnids`, many attempts have been made to fulfill the above three tasks in order to make the middlebox stack programmable. However, none of them can fully facilitate those onerous tasks, as shown in Table 1: the packet parsers like P4 [18] cannot efficiently buffer the packets; the NFV frameworks like NetBricks [58] and ClickNF [37] often rely on TCP-specific modules that are pre-implemented with many native LOC. We discuss these related work in detail in §2.2.

In fact, there exists a dilemma between the abstraction level and code performance when enabling programmability on the performance-demanding middlebox stack. On the one hand, many exceptions like out-of-order packets can arise in L2-L4, so higher-level abstractions are desired to relieve the developers from handling those corner cases. On the other hand, optimizing a stack at wire speed also relies on tuning underlying processing details, which becomes much more challenging if those details are transparent to the developers. As a result, previous works tend to trade off the programmability for the performance, offering limited programmability over specific stacks, *e.g.*, TCP (see §2.3).

In this paper, we propose Rubik, a domain-specific language (DSL) for addressing the above dilemma, which can *fully* program the middlebox stack while assuring *wire-speed* processing capability. For facilitating the stack writing, Rubik offers a set of handy abstractions at the language level, *e.g.*, packet sequence and virtual ordered packets, which handle the exceptions in an elegant fashion. Using these declarative abstractions, operators can compose a more robust middlebox stack with much fewer lines of code, and retain the possibility of flexible extension for future customization needs. For maintaining high performance, Rubik translates its program into an intermediate representation (IR), and uses domain-specific knowledge to automatically optimize its control flow, *i.e.*, eliminating the redundant operations. The optimized IR is then translated into native C code as the performant runtime.

In sum, we make the following contributions in this paper.

- We propose Rubik, a Python-based DSL to program the network stack with minor coding effort, *e.g.*, 250 LOC for a complete TCP/IP stack (§3 and §4).
- We design and implement a compiler for Rubik, where a set of domain-specific optimizations are applied at the IR layer, so that all stacks written in Rubik can benefit from those common wisdom, without caring about how to integrate them into the large code base (§5).
- We prototype Rubik, and build various real cases on it, including 12 reusable protocol parsers, 5 network stacks, and 2 open-source middleboxes (§6). Experiments show that Rubik is at least 30% faster than state of the art (§7).

2 Motivation and Challenges

In this section, we demonstrate that programming middlebox stack is a necessity in modern networks (§2.1), while no existing tool can really enable such programmability (§2.2). We pose the challenges of designing a DSL for middlebox stack, and summarize how our approach addresses them (§2.3).

2.1 Programming Middlebox Stack Matters

As presented in §1, programming a middlebox stack requires huge human effort. However, some argue that it might not be a problem: most middleboxes work with standard TCP/IP protocols, thus a well-written TCP/IP stack should be sufficient. In contrast, we believe there are plenty of scenarios where a deeply customized middlebox stack is desired.

First, the middlebox stacks need to be customized for serving diverse networks with different protocols [13,32,34,42,43, 63,70]. Apart from the existing ones, we note that the emerging programmable data plane may cause an upsurge of new protocols, each of which requires an upgrade of the middlebox stack, or its traffic cannot traverse the network [16, 55].

Second, even for a fixed stack, the operators may still manipulate the packets in arbitrary ways, and the implementation of middlebox stacks varies to satisfy those user-specified strategies. For example, if a TCP packet is lost in the mirrored traffic [22], `libnids` will view this as a broken flow and directly drop it for higher performance [11], while mOS will keep the flow and offer an interface to access the fragmented sequence for maximumly collecting the data [44].

Third, middleboxes are constantly evolving for providing value-added functions, *e.g.*, adding a new layer [12, 35], measuring performance [20], inspecting encrypted data [40,49,67] and migrating/accelerating NFV [38, 47, 57, 68, 72]. These extensions heavily rely on a highly customized stack.

The above facts prove that programming middlebox stack is a necessity in modern networks, which however demands massive human effort. In practice, such overhead has begun to hinder the birth of new protocols: the middlebox vendors tend to be negative to support new protocols, as the huge code modification can cost large human labor and introduce bugs or security vulnerabilities. For example, some middlebox vendors suggest blocking the standard port of QUIC (UDP 443) to force it falling back to TCP, so that their products can analyze such connections [10]. This heavily impacts the user experience [4, 19], and will finally result in the ossification of underlying networks [60].

2.2 Related Work

Plenty of attempts have been made to facilitate the middlebox development, as shown in Table 1. In the following, we show why they are not sufficient to program the middlebox stack.

Programmable packet parsers like P4 [18] and VPP [23] can dissect arbitrary-defined protocols in an amiable way. However, since they target at implementing a switch/router, they cannot efficiently buffer and/or reassemble the packets.

There are also DSLs, *e.g.*, Binpac [59], Ultrapac [52], FlowSifter [56] and COPY [51], that can automatically generate L7 protocol parsers. The parsers they generate focus on the L7 protocols like HTTP, hence can only work on the already reassembled segments. In other words, they can facilitate the development of high-level functions of a middlebox, *e.g.*, HTTP proxy, deep packet inspection, but have to cooperate with a low-level stack, instead of serving as one.

TCP stack libraries include those for end-host stacks and those for middlebox stacks. The end-host stack libraries, *e.g.*, mTCP [45], Modnet [61], Seastar [27], F-Stack [25], only maintain the unidirectional protocol state for a certain end host, while middlebox stacks must track the bidirectional behaviors of both sides. As a result, the middlebox developers cannot build their applications on end-host stack libraries.

On the other hand, the major feature a middlebox stack library provides is the bidirectional TCP flow management. Previously, such libraries are closely embedded in IDS frameworks like Snort [21] and Zeek [24], therefore cannot be reused when developing new applications. `libnids` [11] decouples the TCP middlebox stack from the high-level functions, making the stack reusable. Recent works like mOS [44] and Microboxes [53] implement a more comprehensive TCP stack with fast packet I/O, and more importantly, provide the flexible user-defined event (UDE) programming schemes, *e.g.*, dynamic UDE registration, parallel UDE execution. Besides, they provide limited programmability over TCP stack, *e.g.*, unidirectional buffer management. However, all above approaches are hard-coded, hence cannot support non-TCP stacks without massive native code understanding and writing.

NFV frameworks offer a packaged programming solution from L2 to L7. However, none of them provide complete middlebox stack programmability. MiddleClick [30] and ClickNF [37] can manipulate the stack hierarchy using Click model [48], but they rely on pre-implemented elements, *e.g.*, ClickNF implements the TCP-related elements with 156 source files (12K LOC in C++) [7]. NetBricks [58] supports the customization of the header parser, the scheduled events, *etc*, which is sufficient for programming a connectionless protocol. However, the abstractions for programming a connection-oriented protocol, *e.g.*, transmission window, connection handshake, are still TCP-specific. OpenBox [33] and Metron [46] abstract and optimize a set of L2-L7 elements for middlebox applications. However, the flow management element still has to be pre-implemented using native code.

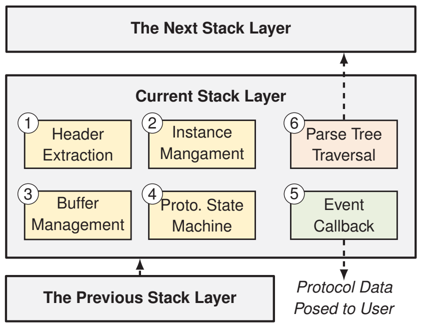


Figure 1: The three key modules in a middlebox stack layer: protocol parser (yellow boxes), event callback (green box), and parse tree traversal (red box).

2.3 Challenges and Our Approach

A DSL that fully captures the L2-L4 abstractions can be a cure to above problems. In the next, we revisit the high-level pipeline of middlebox stack, and pose the challenges of designing and implementing a DSL corresponding to it.

Middlebox stacks follow a layer-based processing pipeline, which is largely the same with the end-host stacks, as shown in Figure 1. Specifically, each stack layer starts by *parsing the protocol* (①–④), which extracts the header, manages the instance, buffers the segments, updates the protocol state machine (PSM), *etc*. Next, the *event callback* module (⑤) will raise the events with the protocol data fed to the users, *e.g.*, the reassembled or retransmitted data. Finally, the *parse tree* (⑥) will decide the next protocol to be parsed. However, even the above pipeline is seemingly natural and generalized, designing and implementing a DSL corresponding to it can still be a challenging task. The reason is two-fold.

First, working at L2-L4, the middlebox stacks run more complex logic than it appears in the pipeline. For example, the out-of-order packets can mess around the PSM, *e.g.*, an early-arrived FIN packet may mislead the stack to tear down the TCP connection. Each of these exceptions is handled with native code in fixed stacks, and it is extremely difficult to provide a neat DSL that covers all such cases. *Rubik addresses this challenge by offering a set of high-level constructs to hide such exceptions from programmers, e.g., "packet sequence" that hides the retransmission exception and "virtual ordered packet" that hides the out-of-order exception, which can maximumly correspond to the intuitive pipeline (§4).*

Second, the middlebox stacks must realize wire-speed processing to serve high-level functions. However, due to the complexities presented above, the optimizations on the stack can only be achieved by carefully tuning the native code. That is, the program written in DSL that hides the processing details will likely produce low-performance native code. *Rubik addresses this challenge by employing an IR and a set of domain-specific optimizations on it before producing the native code, which automatically optimize the DSL program to avoid potential performance traps (§5).*

3 Rubik Overview

In this section, we use a walk-through example to overview how Rubik can facilitate the middlebox stack writing. Our example is an ETH→IP/ARP stack, where we raise a typical event, IP fragmentation, for each IP fragment.

Figure 2 shows the real (and almost complete) Rubik code of realizing our example. We start from declaring the IP layer (Line 2), which initializes internal structures of a connectionless protocol, including the header parser, the packet sequence, PSM, *etc*. These components are specialized as follows.

Parsing the header fields (Line 5–18). One has to first define the header format before she references the headers. In Rubik, a header format is a Python class that inherits `layout`, and each header field is a member of this class, which specifies its length measured by `Bit()`. The order of the members indicates the layout of the fields. Line 5–15 show the IP header structure, `ip_hdr`. We can then use this structure to compose the header parser with one LOC in Line 18. After that, Rubik can reference the fields by their names, *e.g.*, `ihl` can be referenced by `ip.header.ihl`.

Managing the instance table (Line 20). Having the headers, the stack layer then finds the instance that the packet correlates to, *e.g.*, the TCP flow, and processes it by the previous state and data of the same instance. The instances are stored in an instance table, *e.g.*, TCP flow table.

To achieve this, Rubik forms a key to index the instance table, which consists of bi-directional protocol contexts. For IP protocol, the instance key is a list that contains the source and destination IP addresses (Line 20). Note that for connection-oriented protocols, the instance key should contain two lists, each of which indexes the packets of one direction.

Preprocessing the instance (Line 23–27). Before getting into buffer and PSM processing, operators can update some permanent contexts for each individual instance (`perm`), or use some temporary variables for facilitating the programming (`temp`). This part of logic will be executed each time after the instance is found/created. Line 23–27 define a temporary data structure that stores the fragmentation offset for each IP packet, which can then be referenced as `ip.temp.offset`.

Managing the packet buffers (Line 30–31). Many protocols buffer the packets to ensure the correct order of incoming packets. Rubik offers a *packet sequence* abstraction to handle this task. In our example, the IP protocol has to buffer the fragmented packets according to their fragmentation offset. Line 30–31 define a sequence block filled with the IP payload and indexed by the fragmentation offset. This block will be inserted into the packet sequence associated with the instance, which is automatically sorted in ascending order by the meta.

A connection-oriented instance will maintain two sequences for two sides, respectively. The packets payload will be automatically inserted into the corresponding sequence according to the direction indicated by the instance key.

```
1  # Declare IP layer
2  ip = Connectionless()
3
4  # Define the header layout
5  class ip_hdr(layout):
6      version = Bit(4)
7      ihl = Bit(4)
8      ...
9      dont_frag = Bit(1)
10     more_frag = Bit(1)
11     f1 = Bit(5)
12     f2 = Bit(8)
13     ...
14     saddr = Bit(32)
15     daddr = Bit(32)
16
17 # Build header parser
18 ip.header = ip_hdr
19 # Specify instance key
20 ip.selector = [ip.header.src_addr, ip.header.dst_addr]
21
22 # Preprocess the instance using 'temp'
23 class ip_temp(layout):
24     offset = Bit(16)
25 ip.temp = ip_temp
26 ip.prep = Assign(ip.temp.offset,
27                  ((ip.header.f1<<8)+ip.header.f2)<<3)
28
29 # Manage the packet sequence
30 ip.seq = Sequence(meta=ip.temp.offset,
31                   data=ip.payload[:ip.payload_len])
32 # Define the PSM transitions shown in Figure 3
33 ip.psm.last = (FRAG >> DUMP) + Pred(~ip.header.more_frag)
34 ip.psm.frag = ...
35
36 # Buffering event
37 ip.event.asm = If(ip.psm.last | ip.psm.dump) >> Assemble()
38 # Callback each IP fragment using 'ipc'
39 class ipc(layout):
40     sip = Bit(32)
41     dip = Bit(32)
42 ip.event.ip_frag = If(~ip.psm.dump) >> \
43                    Assign(ipc.sip, ip.header.saddr) + \
44                    Assign(ipc.dip, ip.header.daddr) + \
45                    Callback(ipc)
```

Figure 2: IP layer and fragmentation event written in Rubik.

Updating the PSM (Line 33–34). PSM tracks the protocol states, which are useful in most connection-oriented protocols (*e.g.*, TCP handshake), and also in some connectionless protocols that buffer the packets (*e.g.*, IP fragmentation). Consider the IP PSM shown in Figure 3. If an IP packet unsets the `dont_frag` flag, the parser will take a transition from the `DUMP` state to the `FRAG` state that waits for more fragments. The instance will be destroyed if the PSM jumps into an accept state, *e.g.*, `DUMP` in IP PSM. Line 33 defines the `last` transition, *i.e.*, `FRAG`→`DUMP`.

Assembling data and hooking IP fragments (Line 37–45). Unlike the UDEs that are raised by the high-level functions, *e.g.*, HTTP request event, the built-in events (BIEs) reveal the inherent behaviors in the stack, *e.g.*, buffer assembling, connection setup. Previous works only pose fixed and TCP-specific BIEs [11, 44], while Rubik can program two types of BIEs for arbitrary stacks.

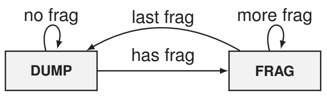


Figure 3: Simplified PSM for IP fragmentation.

The first is for the packet sequence operations, *i.e.*, buffer assembling. Rubik uses `If()` to specify the conditions of raising the events and `Assemble()` to assemble the continuous sequence blocks. This function will form a service data unit (SDU) for the next layer parsing. Line 37 defines the events for assembling the fragments in IP layer.

The second type of action is for posing the user-required data, which is achieved by a `Callback()` function that indicates what content should be posed. Line 39–45 define an event on the condition that fragmented packets arrive. The back-end compiler will declare an empty function in the native C code, *i.e.*, `ip_frag(struct ipc*)`, and invoke it each time the condition is satisfied.

Parse tree for ETH→IP/ARP. Each time after processing a layer, the network stack decides the next layer to be proceeded, until it reaches the end of the stack. All such parsing sequences form a parse tree [39].

The parse tree of our example consists of two layers, *i.e.*, Ethernet, and IP/ARP. The stack executes from the root node, which triggers the Ethernet protocol parser. This parser will extract the headers of Ethernet, *e.g.*, `dmac`, `type`. Next, the parse tree checks the predicates carried by the two transitions, and decides which one could be further parsed. In this case, the `type` field is used to distinguish the IP and ARP protocol. Rubik offers a simple syntax similar to PSM transition to define the parse tree, as shown below.

```
st = Stack()
st.eth, st.ip, st.arp = ethernet, ip, arp
st += (st.eth>>st.ip)  + Pred(st.eth.header.type==0x0800)
st += (st.eth>>st.arp) + Pred(st.eth.header.type==0x0806)
```

where `ethernet`, `ip` and `arp` are protocol parsers. We note that the parsers can be reused in the stack. For example, we can define another IP layer in this stack with `st.other_ip = ip`. This will largely facilitate the customization of encapsulation stacks (see Appendix C.3 for a GTP example).

Summary. We omit the implementation of Ethernet layer and ARP layer, which are quite simple compared to IP layer. In sum, we use ∼50 LOC to define the IP protocol parser (see Appendix C.1 for the complete code), 7 LOC to hook the expected event, and 4 LOC to build the parse tree. As a comparison, `libnids` consumes ∼1000 C LOC to implement the similar stack [11].

4 Rubik Programming Abstractions

§3 shows the potential of reducing coding effort with Rubik. However, as discussed in §2.3, there exists lots of complex programming needs that call for more sophisticated programming abstractions. In this section, we dive into the language internals to present how Rubik conquers those complexities.

4.1 Context-Aware Header Parsing

We consider the following two context-aware header parsing needs in middlebox stack, and address them using Rubik.

Conditional layout. The L2-L4 protocols can have conditional header layout. For example, QUIC uses its first bit to indicate the following format, *i.e.*, long header or short header. To this end, we can first parse the fixed layout, using which to determine the next layout to be parsed, as shown below.

```
quic.header =  quic_type
quic.header += If(quic.header.type == 0) >> long_header
                Else() >> short_header
```

Type-length-value (TLV) parsing. Rubik extends its header parsing component in two ways to express the TLV fields: (1) the value of a field can be assigned before parsing, which can be used to define a `type` field, *e.g.*, `type = Bit(32,const=128)` defines a 32-bit field that must be 128; (2) the length of a field can refer to a pre-defined field with arithmetic expressions, which can be used to define the length of `value` field, *e.g.*, `value = Bit(length << 3)`.

Besides, TLV headers are often used in a sequence with non-deterministic order, *e.g.*, TCP options. Rubik offers a syntax sugar for parsing those headers, as shown below.

```
tcp.header += AnyUntil((opt1, opt2, opt3), cond)
```

where `opt1`–`opt3` are TLV header layouts, and `AnyUntil()` will continuously parse the packet according to their first fields, *i.e.*, `type`, until `cond` turns to be false.

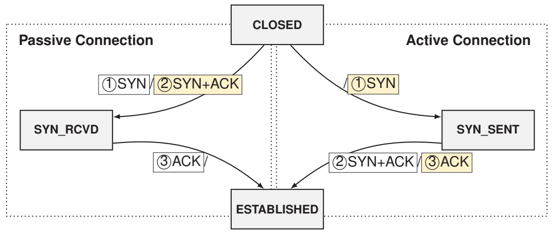
4.2 Flexible Buffer Management

The transport protocols can buffer the packets in flexible ways other than simply concatenating them in order. Specifically, we consider the following three exceptions in buffer management, *i.e.*, retransmission, conditional buffering and out-of-window packets, and use the sequence abstraction in Rubik to address them.

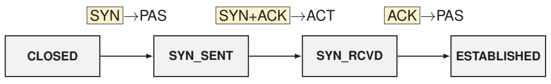
Each time a sequence block is inserted, `Sequence()` in Rubik will compare its meta (*e.g.*, sequence number in TCP) and length with existing buffered blocks, in order to identify the fully and partial retransmission. Operators can decide whether the retransmitted parts should be overwritten by passing a `overwrite_rexmit` flag to `Sequence()`. Once the retransmission is detected, Rubik will automatically raise an event, which can be referenced by `event.rexmit`.

In some cases operators would disable the sequence buffering. For example, a TCP stateful firewall relies on the meta of sequence block to track the TCP states, but does not need the content in the block. Operators can disable the content buffering by simply writing `tcp.buffer_data=False`.

Transport protocols use window to control the transmitting rate. Operators can pass a `window=(wnd,wnd_size)` parameter to `Sequence()`, which specifies the valid range of meta. The out-of-window packets will not be inserted into the sequence, but raise an `out_of_wnd` event.



(a) The end host stack PSM for TCP handshake.



(b) The sender-side PSM for TCP handshake.

Figure 4: The middlebox stack PSM (b) only models the sending behaviors of the end host stack PSM (a).

4.3 Virtual Ordered Packets

The sequence abstraction will sort the out-of-order packets, which, however, still mess around the stack processing. Consider the IP PSM shown in Figure 3. In the possible out-of-order cases, the "last frag" packet can arrive earlier than a "more frag" packet. With the transitions defined in Figure 2, such packet will trigger `ip.psm.last` and form an incomplete SDU for the next layer. To avoid such mistake, the transition has to track two states (instead of the fragmentation flag only), *i.e.*, whether the "last frag" packet *has* arrived and whether the sequence *is* continuous. This counter-intuitive expression makes `Pred` in PSM transitions quite complex.

Rubik addresses such problem by offering an abstraction of *virtual ordered packets*, which gives an illusion to the operators that they are accessing the ordered packets. For example, to handle the early-arrived "last frag" exception, the transition `ip.psm.last` can be rewritten as follows.

```
ip.psm.last = (FRAG >> DUMP) + Pred(~ip.v.header.more_frag)
```

where `ip.v` indicates the virtual ordered packet. The compiler of Rubik will take care of tracking the real arriving order and ensuring the sequence continuity (see §5.4).

Note that the virtual ordered packets are for facilitating the inconsistent condition checking, while no real packet will be buffered and re-accessed. In other words, operators can only use this abstraction in the conditions of `If()` or `Pred()`.

4.4 Sender-Side PSM

Directly emulating the PSM of the end host stack in the middlebox is not a trivial task. Consider a simplified PSM for TCP handshake, whose end host version is shown in Figure 4a. Each transition in the PSM is triggered by two packets: the received packet in the white frame and the sent packet in the gray frame. For example, the passive host (*i.e.*, server) can jump into `SYN_RCVD` state only after it received the SYN packet *and* sent the SYN+ACK packet. This transition is natural for the end hosts, since the receiving and sending behaviors are synchronized.

However, the middlebox cannot capture those two behaviors at the same time. Instead, it has to use two states to respectively capture them. For example, for the passive side, the PSM of a middlebox will jump to a new state, say `SYN_HALF_RCVD`, when processing an SYN packet sent from the client, and will further jump to `SYN_RCVD` only after it sees an SYN+ACK packet sent reversely. That is, the middlebox stack has to maintain two PSMs for two sides, each of which introduces many more states and transitions.

Rubik proposes a new PSM abstraction to reduce the number of states and transitions, *i.e.*, the sender-side PSM, which combines the two-side behaviors and is triggered by a single packet. Figure 4b shows the sender-side PSM of TCP handshake, which consists of only three transitions. The key of this PSM is that it proceeds only by the sent packets (yellow ones), but ignores whether they have been received (white ones). The following defines the first transition in Figure 4b.

```
tcp.psm.syn = (CLOSED >> SYN_SENT) +
                  Pred(tcp.v.header.syn & tcp.to_passive)
```

where `to_passive` indicates the packet is being sent to the passive side in a connection-oriented session.

Note that the sender-side PSM is not a unidirectional PSM. Instead, it tracks *all* the bi-directional packets, but removes the redundancy in the end-host PSMs. For example, in Figure 4a, $\boxed{\text{SYN}}$ is the same packet with $\boxed{\text{SYN}}$. In fact, the sender-side PSM assumes the sent packets must be received. This is reasonable, because the stack cannot detect the packets lost downstream the middlebox. In practice, the middlebox stack will eventually be in the correct state after seeing the retransmitted packets, and before that, a retransmission event will alert that the current state may be inconsistent.

4.5 Event Ordering

By default, all the events will be raised after proceeding PSM and before parsing the next layer (see §5.2). However, operators have to further clarify two kinds of relationships between the events to avoid the potential ambiguity.

First, operators may need to define the "happen-before" relationships of two events, if they have the same or overlapped raising conditions. Consider the aforementioned two events in IP layer, `ip.event.asm` and `ip.event.ip_frag`, both of which will be raised when `ip.psm.last` is triggered. As a result, `ip_frag` might lose the last fragment if `asm` happens first, since the reassemble operation will clear the sequence.

Second, operators may want to raise an event if the other is happening, *i.e.*, the "happen-with" relationship. For example in TCP, an event `rdata` that poses the retransmitted data should be raised only when the retransmission event occurs.

Rubik offers an event relationship abstraction to address the above requirements. The following code indicates that `ip_frag` should happen before `asm`, and `rdata` will be checked and raised each time `rexmit` is happening.

```
ip.event_relation  += ip.event.ip_frag, ip.event.asm
tcp.event_relation += tcp.event.rexmit >> tcp.event.rdata
```

5 Compiling Rubik Programs

In this section, we introduce the compiler of Rubik, which translates the Rubik program into native C code. We first reveal the difficulties of handling the performance issues in the middlebox stack (§5.1). To this end, we translate the Rubik program into an intermediate representation (IR) to reveal the factual control flow of the stack (§5.2). Then the middle-end of the compiler performs domain-specific optimizations on the IR to avoid the performance traps (§5.3), and finally the back-end translates the IR into performant C code (§5.4).

5.1 Avoiding Performance Issues is Hard

As mentioned in §2.3, the generalized execution model can cause severe performance issues. For example, the simple pipeline will insert every IP packet into the sequence, while this is redundant for the non-fragmented IP packets, as their blocks will be assembled right after being inserted. And since the normal IP packets dominate the traffic, this redundant copy will heavily degrade the stack performance.

Previously in hand-written stacks, developers handle each of those performance issues using native code. However, due to the function diversity in the stack, identifying and avoiding *all* such traps for *all* stacks is too harsh for the developers. Moreover, even the developers are aware of those traps, sometimes they have to trade off performance for the code modularity or generality, since the proper handling of those issues will heavily increase the size of the codebase, making the program more bug-prone.

As a DSL, Rubik has better chance to address those performance issues, if it can capture and handle them through its automatic compilation process. This task, however, is still challenging. First, Rubik is designed as a declarative language, which means although the developers can easily write a “correct” program without caring about the inner logic, they also can do little for providing more hints for a “better” program. Second, it is also an impossible mission for the native code compiler due to the lack of domain-specific knowledge, *e.g.*, the fact that the aligned IP packets can be directly passed cannot be obtained from the view of the native compiler.

5.2 Intermediate Representation in Rubik

Rubik addresses above challenges by introducing an IR into the compilation, which brings the following merits. First, the IR code is much smaller, making it possible to do effective optimizations that are unaffordable in native code (see §5.3). Second, the IR code still holds the high-level intent to perform the domain-specific optimizations. Third, the IR layer is a common ground for all Rubik programs, which means the optimizations applied on IR work for *all* stacks.

Specifically, we adopt the Control-Flow Graph (CFG) as the IR, which can clearly reveal the control flow of the stack.

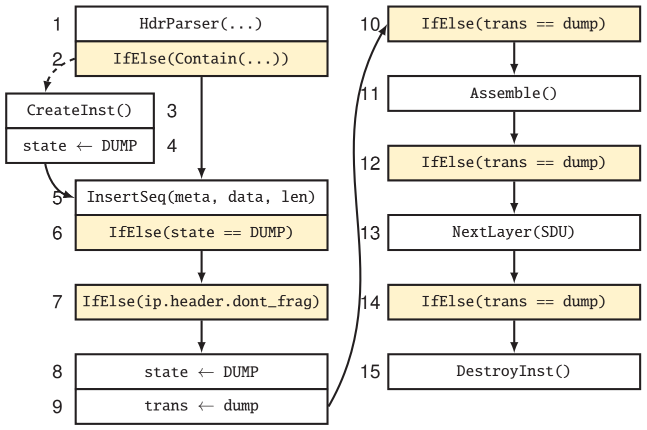


Figure 5: Partial CFG of the IP program. The yellow boxes are the branching blocks. Only one PSM transition (`ip.psm.dump`) is shown. Most of the `false` branches (dashed edge) are also omitted.

Note that each protocol layer works independently in the stack, so we view a protocol parser along with the events defined in its layer as an individual Rubik program.

Composing the control flow. The compiler composes the real control flow of the stack with the next four parts.

First, the compiler constructs the CFG for the protocol parser following the pipeline depicted in Figure 1, *i.e.*, header parsing, instance table, packet sequence and PSM transition, and elaborates it with more information that is relevant to the optimization, *e.g.*, the operations on the instance table and the sequence. The conditional statements, *e.g.*, PSM transitions and event callback, will be translated into the branching blocks with their `Pred`/`If` as the branching conditions.

Second, the compiler decides when to raise the events. It is possible to raise an event just after all its conditions have been met, *i.e.*, the triggering conditions are satisfied and the data in the callback structures are ready. As such, an event that only requires header information can be raised just after the header parser. However, the high-level functions hooking this event may modify the packets, which could impact the correct execution of the PSM. Hence, the compiler puts all events after proceeding the PSM, and decides their order by the explicitly defined happen-before and happen-with relationship in `event_relation`. The only exceptions are the built-in sequence events, *i.e.*, `rexmit` and `out_of_wnd`, which will be triggered by sequence operations before proceeding the PSM, as well as the events happening with them.

In the third and fourth parts, the compiler checks the conditions for parsing the next layer and destroys the instance if it jumps into the accepted PSM states.

Figure 5 shows a partial CFG for the IP program presented in §3. The white boxes are the basic blocks, and the yellow ones are branching blocks, where the solid/dashed edges indicate the `true`/`false` branches. In these blocks, IR uses instructions to reveal the operations on the real data structure. For example, we use `CreateInst()` to create and insert an

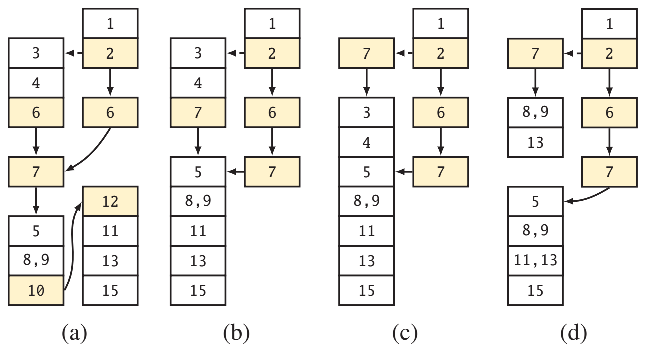


Figure 6: (a) First-round branch lifting, where $\boxed{6}$ is bounded by $\boxed{2}$ and $\boxed{4}$. (b) Constant analysis, where $\boxed{6}$, $\boxed{10}$ and $\boxed{12}$ are eliminated because they are always `true` guarded by $\boxed{4}$ and $\boxed{9}$. (c) Second-round branch lifting, where $\boxed{7}$ is further lifted above $\boxed{3}$, due to the absence of $\boxed{6}$. (d) Peephole optimization, where $\boxed{3}$-$\boxed{5}$, $\boxed{11}$ and $\boxed{15}$ are eliminated.

instance into the instance table, and `InsertSeq()` to insert the payload into its sequence. Blocks $\boxed{1}$–$\boxed{9}$ are for the IP protocol parser (only one PSM transition, *i.e.*, `ip.psm.dump`, is shown); blocks $\boxed{10}$ and $\boxed{11}$ are for the sequence assemble event; blocks $\boxed{12}$ and $\boxed{13}$ are for the next layer parsing, and blocks $\boxed{14}$ and $\boxed{15}$ handle the accept PSM state.

Revealing the dependency. Given the real control flow with CFG, the compiler can then reveal the dependency relationships between each instruction. This is achieved by checking the read/write operations in the instructions. For example, the instructions below a branching block can only be executed after reading the objects in that branching conditions. Hence, these instructions all depend on those objects.

We note that some read/write relationships are not explicit in the CFG. For example, `InsertSeq()` writes the sequence in current instance, and `Assemble()` reads the same sequence. Hence, `Assemble()` depends on `InsertSeq()`. We pre-define the implicit dependencies for all instructions.

5.3 Middle-End: Optimizing Control Flow

The middle-end first transforms the CFG to expose the complete processing logic on a same set of packets. Then, it applies domain-specific optimizations targeting on the "heavy" instructions to produce an optimal CFG. Specifically, the middle-end iteratively takes the following three steps until the CFG converges to a stable form.

Step 1: Lifting the branches. We lift all the branching blocks to the top of the CFG, as long as they do not depend on an upper block. Figure 6a shows the CFG with branching blocks lifted. We take $\boxed{6}$ as an example: it is lifted to top of the `true` branch of $\boxed{2}$, because in this branch, it only depends on the conditions in $\boxed{2}$; in contrast, in the `false` branch it can only be lifted below block $\boxed{4}$, because it reads the variable `state`, which is written by block $\boxed{4}$. Through this process, $\boxed{6}$ is duplicated, as both branches should traverse it. Note that some `false` branches are omitted in Figure 6a, *e.g.*, the `false` branches of $\boxed{6}$ and $\boxed{7}$ that contain the duplicated $\boxed{5}$.

The branch lifting process merges the basic blocks, which helps to expose a complete processing logic on an individual set of packets. This process maps to the "code sinking" transformation in conventional compilers, which however usually is not performed, since the codes would explode due to the duplication. In contrast, Rubik's IR code is small and with a neat pipeline, making this expensive transformation affordable.

Step 2: Constant analysis. This step replaces or removes the instructions if they are evaluated to be a constant. For example, consider $\boxed{6}$ in the `false` branch in Figure 6a. We can easily assert that its condition (`state==DUMP`) is always `true`, because $\boxed{4}$ has just assigned `state` with `DUMP`. Similar analysis takes place in block $\boxed{9}$, $\boxed{10}$, $\boxed{12}$, where `trans==dump` in the latter two must be `true`. Those always-`true` branch blocks can be removed, as shown in Figure 6b. Note that this elimination may create new opportunities to iteratively lift the branch, *i.e.*, Step 1. For example, $\boxed{7}$ can be further lifted above $\boxed{3}$, due to the absence of $\boxed{6}$, as shown in Figure 6c.

Step 3: Peephole optimizations. After the first two steps, the complete processing logic on each packet set is revealed. For example, $\boxed{3}$–$\boxed{15}$ in Figure 6c illustrates the processing logic on the first packet of an IP instance which is with `dont_frag` flag. In this step, we engage a series of peephole optimizations [36] to identify and eliminate performance traps.

Considering $\boxed{3}$–$\boxed{15}$, we have the following easy optimizations. (1) For $\boxed{3}$, $\boxed{5}$, $\boxed{11}$, $\boxed{13}$, it is obvious that the first and only inserted block is directly assembled. Hence, $\boxed{5}$ and $\boxed{11}$ can be eliminated, and $\boxed{13}$ can be rewritten into `NextLayer(Payload)`. (2) $\boxed{3}$ and $\boxed{15}$ is another pair of redundant operations, where the inserted instance is directly removed from the instance table. These two blocks can also be eliminated. Figure 6d shows the CFG that removes all redundant operations, most of which are very expensive, *e.g.*, instance creation and sequence insertion. We can therefore expect a much higher performance with this optimized CFG.

We emphasize that the patterns of the optimizations are not newly designed, but the common wisdoms borrowed from the mature stack implementations. The key is that manually realizing those optimizations for each stack would mess around the processing pipeline, and significantly increase the complexity of the code. In contrast, Rubik's middle-end are stack- and implementation-oblivious, *i.e.*, operators can focus on the logic of the optimizations without caring about how to integrate them with the stack logic. That is, the new patterns can be easily extended in the future, and the developers can obtain a fully optimized pipeline for all stacks. Appendix D shows the peephole optimizations that are currently employed.

5.4 Back-End: Producing Efficient Code

The back-end of compiler translates and assembles the optimized CFGs into native C code, and ensures its efficiency in two ways: (1) maximize the code efficiency without considering the code readability, *e.g.*, composing a single large

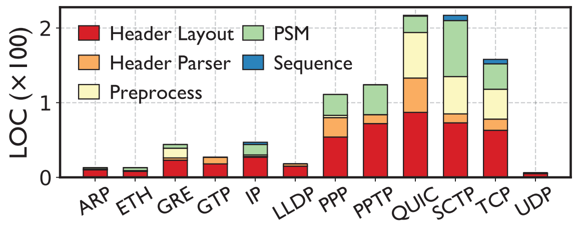


Figure 7: LOC breakdown of protocol parsers.

function for the whole stack to force the optimization in the C compiler; (2) borrow the best practice from existing middlebox stacks, *e.g.*, a fast hashing library. Except for the above general methods, we highlight two designs in the back-end.

Handling the header. The back-end translates the header fields and their references using the following principles.

- Each `layout` will be translated into a C `struct`, and the header parser is a `struct` pointer to the starting address of the header, so that each field can be directly accessed as a `struct` member. For the composite headers, *e.g.*, `ip.header=ip_hdr+ip_opt`, the back-end will generate multiple pointers pointing to different locations.
- Conditions and predicates of virtual ordered packets, *e.g.*, `If(ip.v.header.more_frag)`, will be implemented as tracking two states, *i.e.*, `if(seen_frag && no_hole)`, where `seen_frag` will be set if "more_frag" packet has arrived, and `no_hole` is assigned by checking the sequence each time a sequence block is inserted.

Threading model. We adopt the shared-nothing model with the run-to-complete workflow when generating native code. That is, each core runs an independent stack, which eliminates the inter-core communication [46]. Specifically, the back-end leverages the symmetric receive-side scaling (S-RSS) technique [71], so bi-directional packets from the same connection can be correlated to the same thread. Since modern NICs support hardware-based S-RSS, this usually linearly boosts the stack performance with the number of cores (see §7.1).

The back-end also takes care of other cases that require considerable human effort, *e.g.*, buffer outrun and timeout.

6 Rubik in Action

Rubik builds upon Python while offering domain-specific syntaxes and functions. In total, our prototype amounts to 3K Python LOC for Rubik internals, and 2K C LOC for hashing, packet I/O and sequence operations. The source code of Rubik is available at https://github.com/ants-xjtu/rubik.

In this section, we demonstrate the practicality of Rubik by implementing numbers of mainstream L2-L4 protocols and stacks (§6.1), and developing typical high-level middlebox functions (§6.2).

6.1 Collected Protocols and Stacks

We collect and implement 12 L2-L4 protocol parsers using Rubik. Here we focus on how many LOC used for the implementation, which reflects the complexity and robustness of the program. From the LOC breakdown shown in Figure 7, we have the following observations.

Table 2: Rubik and generated LOC for composing stacks.

Stack	Parse Tree	Addi.	Total	Gen.
TCP/IP	ETH→IP→UDP/TCP	14	245	11061
GTP	ETH→IP→UDP→GTP→IP→TCP	18	304	11384
PPTP	ETH→IP→TCP→PPTP ETH→IP→GRE→PPP→IP→TCP	37	586	46546
QUIC	loopback→IP→UDP→QUIC	23	361	14007
SCTP	ETH→IP→SCTP	9	233	23863

Addi.: *additional Rubik LOC apart from the individual protocol parsers*
Total: *total Rubik LOC* **Gen.:** *generated native LOC*

First, Rubik can express the mainstream L2-L4 protocols with minor LOC. Most connectionless protocols only take tens of LOC. The connection-oriented ones take more, but within hundreds of LOC. Second, most LOC are for defining the header layout (46% in average), since one field takes one LOC in Rubik. This task is quite straightforward if given the protocol specification, so the factual effort of writing a protocol parser is even less than it appears in the figure.

Reducing the effort of implementing above parsers is very valuable. For example, the stream control transmission protocol (SCTP) [2] provides many useful transmission features like message boundary preservation and multi-homing. However, this requires a significant change for middleboxes, *e.g.*, 4400 C LOC in Wireshark [3], making SCTP much less deployed [41]. Using Rubik, it only takes 210 LOC for implementing the SCTP layer. Another example is QUIC [6]: although its multiplexing feature improves the transmission efficiency, existing middleboxes cannot support it without a fundamental upgrade. As a reference, Wireshark takes ∼3100 C LOC to realize the QUIC protocol parser [26]. Using Rubik, merely 216 LOC is enough for prototyping a QUIC parser (without the decryption feature, see §8).

We finally implement 5 typical stacks, as shown in Table 2. We highlight that with the reusable protocol parsers, composing a middlebox stack requires minor additional Rubik LOC, although the native LOC generated is massive.

6.2 Developing Applications with Rubik

`Callback()` in defining BIEs will generate empty callback functions in the native code, which will be invoked each time the BIEs are triggered. The programmers can then develop their applications by implementing those functions.

The most typical example can be a DPI application that inspects the L7 data. In this case, the developer can pose an event happening with the assemble event (`tcp.event.asm`).

```
tcp.event.sdu = Assign(sdu_layout.sdu, tcp.sdu) + \
                Callback(sdu_layout)
tcp.event_relation += tcp.event.asm >> tcp.event.sdu
```

Other examples include the detection of SYN-flood and fake-reset, which can be implemented through the BIEs triggered by the first SYN packet and the reset transitions, respectively.

We present how we port Snort [21] as a more comprehensive example. Snort may scan the traffic multiple times against the rules, *e.g.*, on the fragmented IP packets or on the reassembled L7 data. With Rubik, the programmers can implement these scanning behaviors through the corresponding BIEs posed in the stack. Specifically, we implement 25 rule options, *e.g.*, `content`, `pcre`, `http_header`, and translate the rules into event-based callback functions. We replace Snort's `stream` and `http-inspect` modules with Rubik-generated stacks and events, and reuse the high-level matching modules like Aho-Corasick algorithm for string matching and HyperScan [69] for regular expression matching.

Note that, unlike mOS and Microboxes, Rubik currently does not support programming UDEs. As a result, for HTTP-related rules, we need to manually parse the L7 protocols in the callback function (instead of using a set of inherited UDEs), then the L7 rules can be matched against those parsed HTTP headers. §8 discusses the UDE programming in detail.

7 Evaluation

In this section, we evaluate the performance of Rubik. Specifically, our experiments aim to answer the following questions:

(1) Do Rubik-generated stacks provide comparable or even better performance than the hand-written stacks? (§7.1)

(2) Do Rubik-ported applications work correctly and efficiently on various stacks? (§7.2)

(3) Do the middle-end optimizations help improve the performance of Rubik? (§7.3)

7.1 Microbenchmarks

To measure Rubik's performance under certain traffic load, we build real end-host applications and set a bump-in-the-wire testbed as the middlebox stack. Due to the lack of high-performance non-TCP applications (*e.g.*, QUIC, SCTP), the microbenchmarks are mostly about the TCP/IP stack.

Experimental settings. We build the testbed on an x86 machine (20×Intel Xeon 2.2Ghz, 192GB memory) with three dual-port 40G NICs (Intel XL710). We use another six machines (8×Intel Xeon 2.2Ghz, 16GB memory) to build three server/client pairs. Each server/client has a single-port 40G NIC, and is connected through one NIC in the testbed server.

The clients and servers generate 96K concurrent connections in total (32K from each pair). Each connection fetches a file from the server (1KB by default), and will immediately restart when it terminates. Note that the three pairs cannot drain the 120Gbps link, so we indicate the upper bounds of the throughput for each setting in the experiments, *i.e.*, the throughputs when directly wiring the clients up to the servers.

We synthesize three high-level functions to simulate different workloads of the middlebox: (1) a flow tracker (FT) that tracks the L4 states but ignores the payload, (2) a data assembler (DA) that dumps bidirectional L7 data to `/dev/null`, and

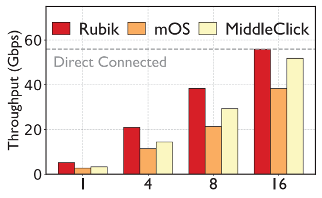


Figure 8: The multi-core scalability (DA, 1KB file).

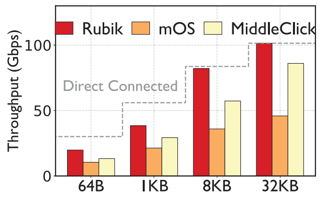


Figure 9: The file size scalability (DA, 8 cores).

(3) a string finder (SF) that matches 50 regular expressions against L7 data. We run DA as the default function in the experiments, as it reflects the intrinsic performance of a complete middlebox stack, *i.e.*, with bidirectional data reassembly and without heavy operations on that data. When running FT, we disable the data buffering for all involved approaches.

TCP stack. From the existing approaches shown in Table 1, we choose to compare Rubik with the TCP-specific stack and the NFV framework, since the packet parser cannot implement a full-functional stack. Specifically, we involve mOS [14] and MiddleClick [28] in our experiments. The former is the state-of-the-art TCP middlebox stack, and the latter with Click model is reported to be more efficient. All approaches have the same packet I/O capability with DPDK [8] and S-RSS. The clients and servers are implemented using mTCP [45].

Figure 8 shows the multi-core scalability of the involved approaches. Thanks to S-RSS, the performance of all approaches can almost linearly scale with the number of CPU cores. Note that Rubik's TCP stack achieves 5.2Gbps, 20.9Gbps, 38.4Gbps when using 1, 4, 8 cores, respectively, and can reach the upper bound (55.9Gbps) with 16 cores. Such throughput outperforms other approaches by 30%–90%.

Figure 9 shows the scalability with different file sizes. With 8 cores, Rubik's TCP stack can reach the upper bound with the file larger than 8KB (82.1Gbps in 8KB, 101.4Gbps in 32KB). We also report that Rubik can reach the upper bound for all file sizes if using 16 cores. Note that given the flow size (which can be inferred from the file size) and the throughput, we can estimate the connection arrival rates, *i.e.*, how many new connections can be handled per second. We report that with 8 cores, the connection arrival rates of Rubik's TCP stack are 4.5M/s and 1.1M/s for 64B and 8KB files, respectively.

Figure 10 shows the throughput with different functions. Rubik's stack can realize 44.1Gbps and 25.5Gbps for FT and SF with 8 cores, which maintain the lead to other approaches (34% and 90% faster than MiddleClick and mOS). We also report that Rubik's stack adds reasonable transferring latency (not shown in the figure), *e.g.*, running DA for a single flow adds 29μs and 97μs to the flow completion time when transferring 64B and 8KB files (62μs and 119μs without middlebox).

Why Rubik outperforms other approaches. First, the peephole optimizations applied in the middle-end let Rubik handles each packet class in the most efficient way, which guarantees a comparable performance to the mature hand-written ones. Second, the hand-written stacks have to trade off perfor-

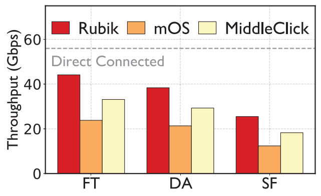


Figure 10: Throughput vs. functions (1KB file, 8 cores).

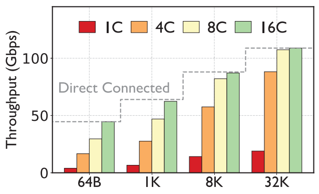


Figure 11: Performance of the GTP stack (DA).

mance for the code maintainability. For example, for maintaining the 8K C LOC of TCP stack, mOS spans more than 100 non-inline functions, dozens of which would be invoked for processing each packet. In contrast, Rubik puts 11K C LOC in a single function for composing the same stack, which incurs much fewer function calls, hence the higher throughput. Third, the one-big-function also forces the optimizations of native C compiler. Specifically, we re-compile MiddleClick and Rubik's generated code with `-O0` instead of `-O3`, and observe that MiddleClick's performance downgrades by 40%, while Rubik suffers 50% degradation. This partially confirms that deeper optimizations can be applied in the one-big-function.

We emphasize that both mOS and MiddleClick are with high-quality code. Even though, the performance trade-offs for maintainability are still inevitable when handling so many LOC with human oracle. Such risk would only be higher for more complex stacks. In contrast, Rubik avoids the performance traps for *all* stacks by automatically applying the domain-specific optimizations in its middle-end, while ensuring the maintainability with its neat syntaxes in the front-end.

GTP stack. Besides the TCP stack, we also run the GTP stack in end hosts for evaluation. Specifically, we modify mTCP to encapsulate/strip IP, UDP, and GTP layers for each TCP packet, where the new IP and UDP layers have the same IP addresses and ports with the original TCP packet. Note that this operation adds ~50 bytes to each packet, which will lift the throughput upper bound when transferring small files.

Figure 11 shows the performance of Rubik's GTP stack with different CPU cores and file sizes. With only one core, Rubik's GTP stack can realize 4.0Gbps and 14.2Gbps for 64B and 8KB file, respectively, and can reach their performance upper bounds (44.6Gbps and 88.0Gbps) with 16 cores. We highlight that even the GTP stack has much more layers than the TCP stack, Rubik can still catch up with the throughput, as its back-end introduces minor overhead between each layer.

7.2 Performance on Various Stacks

We collect real and synthetic traces to evaluate the performance of Rubik on various stacks with real applications.

Traces. We prepare traces for five stacks, as shown in Table 3. The TCP trace is captured in a campus network. The GTP trace is captured in an ISP's base station. For PPTP stack, we set a PPTP server with MPPE and PPP compression disabled, and capture the trace by accessing random websites. For QUIC stack, we set a pair of client and server

Table 3: The collected traces.

Trace	#Pkts.	#Flows	Avg. Flow Size	Avg. Pkt. Length	L7 Data Size	Total Size
TCP	25.9M	558K	32KB	652.13B	8.4GB	16.8GB
GTP	18M	630K	14KB	484.03B	0.6GB	8.7GB
PPTP	6.7M	9K	665KB	892.98B	4.2GB	6.0GB
QUIC	12.7M	3K	637KB	643.25B	5.4GB	8.1GB
SCTP	7.6M	600K	5KB	374.35B	2.3GB	2.9GB

Table 4: The throughput (Gbps) on the traces (16 cores).

	Snort	Rubik+Snort	nDPI	Rubik+nDPI	DA
TCP	20.41	26.86	25.94	25.26	117.76
GTP	15.36	22.79	18.87	18.37	113.42
PPTP	13.91	20.01	18.79	18.22	118.41
QUIC	-	-	-	-	116.29
SCTP	-	-	-	-	101.27

using `ngtcp2` [17], and capture the trace by querying random resources. Rubik currently does not support the online decryption, so we replace the encrypted data in the trace with a deciphered one using the local SSL key (see §8). For SCTP stack, we set the server and client using `usrsctp` [29], and capture the trace by fetching random files. We filter out the incomplete connections (flows without handshake or teardown) for all the traces, so we can properly replay them on a loop.

Applications. We port two well-known middlebox applications, Snort [21] and nDPIReader [15], to Rubik. For Snort, we port it as presented in §6.2, and load 2800 TCP- or HTTP-related rules from its community rule set. We note that nDPIReader does not implement a complete flow reassembly feature. To this end, we pose the TCP and UDP assemble events, and invoke the core detecting functions provided by nDPI in the callback functions. The Rubik-ported version can then detect protocols on reassembled data.

We equip the original Snort and nDPI with DPDK/S-RSS and involve them into the comparison. Note that neither of original and Rubik-ported versions can inspect QUIC or SCTP trace, because the rules and detecting applications are TCP-specific, *i.e.*, they assume the transport layer must be TCP/UDP. However, we argue that with the help of Rubik, it would be quite simple to port such rules to new stacks, *e.g.*, inspecting the HTTP content carried by an SCTP connection.

Performance. We split the traces into three pieces by their IP addresses and use three machines to inject them into the testbed (120Gbps line rate). The testbed runs on 16 cores.

Table 4 shows the throughput with different applications, from which we have two observations. First, the Rubik-ported Snort is faster than the original and the boost is more significant on the stacks with more layers (+31.6% for TCP vs. +48.3% for GTP), because "heavy" stacks would amplify the efficiency of Rubik's generated one-big-function. Second, the Rubik-ported nDPI is slightly slower than the original (−2.8% in average), because the latter does not reassemble the data at all. Specifically for TCP stack, the Rubik-ported versions perform better than the mOS-ported versions (+31.6% vs.

Table 5: Throughput boost (Gbps) and compilation slow down (seconds) from the middle-end optimizations (1 core, DA). Shadowed cells show the number with optimizations.

	TCP		GTP		PPTP		QUIC		SCTP	
Throughput	8.83	22.4	5.47	14.4	8.16	18.9	7.03	13.7	4.38	6.61
Rubik→C	0.06	0.28	0.07	0.31	0.10	1.19	0.07	0.42	0.06	1.22
C→Binary	3.32	3.47	3.47	5.46	3.73	13.1	3.47	4.02	3.27	6.86

+16.8% for Snort, −2.6% vs.−3.7% for nDPI) [44]. We finally highlight that when running DA, all Rubik's stacks can achieve more than 100Gbps throughput for their traces.

Correctness. We respectively select 100 flows from all traces. By manually verifying the results of protocol parsers and event callbacks, we confirm the correctness of Rubik.

7.3 Middle-End Optimizations

We inject the same traces used in §7.2 into the testbed and measure the performance and overhead of corresponding stacks, by enabling/disabling the middle-end optimizations.

Performance. The top part of Table 5 shows that all stacks can significantly benefit from the optimizations, with a boost rate of 51%–163%. The effect of the optimizations depends on two factors. First, since each layer is independently optimized, more layers lead to more improvements. Second, the major optimization is the elimination of the sequence operations, so more boosts can be gained when handling large flows. For example, PPTP and QUIC traces have similar flow size, but the PPTP stack with more layers gains more from the optimizations; TCP and SCTP stacks have the same number of layers, but the SCTP stack does not boost as much as TCP due to the smaller flow size of the trace (5KB vs. 32KB).

Overhead. The branch lifting in the middle-end leads to much larger CFG, which increases the time of compilation, *i.e.*, from Rubik program to C code, and from C code to binary. The bottom part of Table 5 shows that such overhead is minor in practice, *i.e.*, all stacks are compiled within 15 seconds.

8 Limitations and Discussion

Semantics completeness. There are generally two types of middleboxes: the flow-monitoring ones that parse the protocols, check the reassembled data, and forward/drop the original packets (*e.g.*, IDS), and the flow-modifying ones that intercept the connection and modify L7 content (*e.g.*, HTTP proxy). To the best of our knowledge, Rubik can well support programming the former type. For the latter, Rubik should extend its sequence abstraction for inline-reordering, and event abstraction for modifying the packet content. These extensions are realizable and will be explored in our future work.

Encrypted layers. Rubik can cooperate with the encrypted layers in the following two ways: (1) the stacks can directly work on the raw packets, if the middleboxes are placed inside the secure district, where the encrypted content has already been resolved by the gateway; (2) the stacks can inspect the encrypted content, given the proper decipher keys. We simulate the first scenario with QUIC protocol in our evaluation. For the second, we can offer an extra decryption function to modify SDU. We leave this feature to our future work.

UDE programming. Prior to Rubik, literatures focus on how to facilitate the middlebox development by offering friendly UDE interfaces. mOS [44] unifies the TCP stack and provides a BSD-style socket interface, so the developers can dynamically register/deregister their UDEs. Microboxes [53] further optimizes the UDE model by parallelizing their executions, making the performance scalable with the number of network functions. Since the UDE programming is oblivious from the stack programming, we believe providing such feature should be an easy task for Rubik's back-end, by borrowing the best practice from mOS and Microboxes.

Boosting S-RSS. We discuss two possible techniques that can further boost the packet dispatching. First, unlike S-RSS that dispatches correlated packets to fixed CPU cores, RSS++ [31] and eRSS [64] can collect the flow statistics and dynamically dispatch packets to different cores, which can further balance the CPU load. Second, the hardware-based S-RSS can only classify fixed protocols. For example, for PPTP stack, it only dispatches the packets by the lower-layer IP addresses, which are almost fixed as a tunnel. The dispatching can be much more balanced if higher-layer IP and TCP protocols can be considered. A smart NIC [54] or a programmable switch associated with the middlebox [46] that can dispatch the packets by arbitrary headers could be a cure to this problem.

Middlebox deployments. While Rubik facilitates the development of a single middlebox, there are literatures that deploy middleboxes in a distributed way [62, 65] or as a cloud service [66]. These works can be complementary to Rubik.

Ethics statement. The TCP and GTP traces used in the experiments are anonymized before given to us. We claim that our work does not raise any ethics issue.

9 Conclusion

This paper proposed Rubik, a language for programming the middlebox stack, which offers a set of high-level constructs as efficient building blocks, and an optimizing compiler to produce high-performance native code. We demonstrated the minor effort for implementing 12 protocol parsers and 5 popular stacks using Rubik. We evaluated Rubik with real applications and traces, and showed that the generated stacks outperform existing approaches by at least 30% in throughput.

Acknowledgments. We thank the anonymous NSDI reviewers and our shepherd Dejan Kostić for their valuable feedback. We thank Patrick P. C. Lee and Aaron Gember-Jacobson for their comments on the early version of this paper, and Tom Barbette for his help on the codebase of MiddleClick. This work is supported by NSFC (61702407, 61772412, 61902307 and 61702049), and Fundamental Research Funds for the Central Universities. Peng Zhang is the corresponding author.

References

[1] Ip in ip tunneling. https://tools.ietf.org/html/rfc1853, 1995.

[2] Stream control transmission protocol. https://tools.ietf.org/html/rfc4960, 2007.

[3] packet-sctp.c. https://github.com/boundary/wireshark/blob/master/epan/dissectors/packet-sctp.c, 2013.

[4] Do you guys block udp port 443 for your proxy/web filtering? https://bit.ly/2OBM511, 2015.

[5] In-band network telemetry (int). https://p4.org/assets/INT-current-spec.pdf, 2016.

[6] Quic: A udp-based secure and reliable transport for http/2. https://tools.ietf.org/html/draft-ietf-quic-transport-11, 2017.

[7] Clicknf. https://github.com/nokia/ClickNF, 2018.

[8] Dpdk. http://www.dpdk.org/, 2018.

[9] IEEE 802.11 wireless local area networks. http://grouper.ieee.org/groups/802/11/, 2018.

[10] The impact on network security through encrypted protocols - quic. https://bit.ly/2xw7z8y, 2018.

[11] Libnids. http://libnids.sourceforge.net/, 2018.

[12] Middlebox cooperation protocol specification and analysis. https://mami-project.eu/wp-content/uploads/2015/10/d32.pdf, 2018.

[13] Middleboxes: Taxonomy and issues. https://tools.ietf.org/html/rfc3234, 2018.

[14] mos-networking-stack. https://github.com/ndsl-kaist/mos-networking-stack, 2018.

[15] ndpi. https://www.ntop.org/products/deep-packet-inspection/ndpi/, 2018.

[16] The new waist of the hourglass. http://tools.ietf.org/html/draft-tschofenig-hourglass-00, 2018.

[17] ngtcp2. https://github.com/ngtcp2/ngtcp2, 2018.

[18] P4 language. https://p4.org/, 2018.

[19] Quic protocol | cisco community. https://community.cisco.com/t5/switching/quic-protocol/td-p/3402269, 2018.

[20] Report from the iab workshop on stack evolution in a middlebox internet (semi). https://www.rfc-editor.org/rfc/rfc7663.txt, 2018.

[21] Snort - network intrusion detection and prevention system. https://www.snort.org/, 2018.

[22] Snort: Re: Is there a snort/libnids alternative. http://seclists.org/snort/2012/q4/396., 2018.

[23] Vector packet processing. https://fd.io/, 2018.

[24] The zeek network security monitor. https://www.zeek.org/, 2018.

[25] F-Stack. http://www.f-stack.org/, 2019.

[26] packet-quic.c. https://bit.ly/32ED3YK, 2019.

[27] Seastar. http://seastar.io/, 2019.

[28] Middleclick. https://github.com/tbarbette/fastclick/tree/middleclick, 2020.

[29] usrsctp: A portable sctp userland stack. https://github.com/sctplab/usrsctp, 2020.

[30] T. Barbette, C. Soldani, R. Gaillard, and L. Mathy. Building a chain of high-speed vnfs in no time: Invited paper. In *IEEE HPSR*, 2018.

[31] Tom Barbette, Georgios P. Katsikas, Gerald Q. Maguire, and Dejan Kostić. Rss++: Load and state-aware receive side scaling. In *ACM CoNEXT*, 2019.

[32] Apurv Bhartia, Bo Chen, Feng Wang, Derrick Pallas, Raluca Musaloiu-E, Ted Tsung-Te Lai, and Hao Ma. Measurement-based, practical techniques to improve 802.11ac performance. In *ACM IMC*, 2017.

[33] Anat Bremler-Barr, Yotam Harchol, and David Hay. Openbox: A software-defined framework for developing, deploying, and managing network functions. In *ACM SIGCOMM*, 2016.

[34] Ryan Craven, Robert Beverly, and Mark Allman. A middlebox-cooperative tcp for a non end-to-end internet. In *ACM SIGCOMM*, 2014.

[35] K. Edeline and B. Donnet. Towards a middlebox policy taxonomy: Path impairments. In *IEEE INFOCOM Workshops*, 2015.

[36] Charles N. Fischer, Ron K. Cytron, and Richard J. LeBlanc Jr. *Crafting A Compiler*. Pearson, 2009.

[37] Massimo Gallo and Rafael Laufer. Clicknf: a modular stack for custom network functions. In *USENIX ATC*, 2018.

[38] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. Opennf: Enabling innovation in network function control. In *ACM SIGCOMM*, 2014.

[39] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown. Design principles for packet parsers. In *IEEE/ACM ANCS*, 2013.

[40] Juhyeng Han, Seongmin Kim, Jaehyeong Ha, and Dongsu Han. Sgx-box: Enabling visibility on encrypted traffic using a secure middlebox module. In *ACM APNet*, 2017.

[41] David A. Hayes, Jason But, and Grenville Armitage. Issues with network address translation for sctp. *ACM SIGCOMM Comput. Commun. Rev.*, 39(1):23–33, 2009.

[42] Benjamin Hesmans, Fabien Duchene, Christoph Paasch, Gregory Detal, and Olivier Bonaventure. Are tcp extensions middlebox-proof? In *ACM HotMiddlebox*, 2013.

[43] Michio Honda, Yoshifumi Nishida, Costin Raiciu, Adam Greenhalgh, Mark Handley, and Hideyuki Tokuda. Is it still possible to extend tcp? In *ACM IMC*, 2011.

[44] Muhammad Asim Jamshed, YoungGyoun Moon, Donghwi Kim, Dongsu Han, and KyoungSoo Park. mos: A reusable networking stack for flow monitoring middleboxes. In *USENIX NSDI*, 2017.

[45] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mtcp: a highly scalable user-level TCP stack for multicore systems. In *USENIX NSDI*, 2014.

[46] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. Metron: NFV service chains at the true speed of the underlying hardware. In *USENIX NSDI*, 2018.

[47] Junaid Khalid, Aaron Gember-Jacobson, Roney Michael, Anubhavnidhi Abhashkumar, and Aditya Akella. Paving the way for NFV: Simplifying middlebox modifications using statealyzr. In *USENIX NSDI*, 2016.

[48] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, 2000.

[49] Chang Lan, Justine Sherry, Raluca Ada Popa, Sylvia Ratnasamy, and Zhi Liu. Embark: Securely outsourcing middleboxes to the cloud. In *USENIX NSDI*, 2016.

[50] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. The quic transport protocol: Design and internet-scale deployment. In *ACM SIGCOMM*, 2017.

[51] Hao Li, Chengchen Hu, Junkai Hong, Xiyu Chen, and Yuming Jiang. Parsing application layer protocol with commodity hardware for sdn. In *IEEE/ACM ANCS*, 2015.

[52] Zhichun Li, Gao Xia, Hongyu Gao, Yi Tang, Yan Chen, Bin Liu, Junchen Jiang, and Yuezhou Lv. Netshield: massive semantics-based vulnerability signature matching for high-speed networks. In *ACM SIGCOMM*, 2010.

[53] Guyue Liu, Yuxin Ren, Mykola Yurchenko, K.K Ramakrishnan, and Timothy Wood. Microboxes: High performance nfv with customizable, asynchronous tcp stacks and dynamic subscriptions. In *ACM SIGCOMM*, 2018.

[54] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto smartnics using ipipe. In *ACM SIGCOMM*, 2019.

[55] Alberto Medina, Mark Allman, and Sally Floyd. Measuring the evolution of transport protocols in the internet. *ACM SIGCOMM Computer Communication Review*, 35(2):37–52, 2005.

[56] C Meiners, E Norige, A. X Liu, and E Torng. Flowsifter: A counting automata approach to layer 7 field extraction for deep flow inspection. In *IEEE INFOCOM*, 2012.

[57] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: A framework for nfv applications. In *ACM SOSP*, 2015.

[58] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. Netbricks: Taking the v out of NFV. In *USENIX OSDI*, 2016.

[59] Ruoming Pang, Vern Paxson, Robin Sommer, and Larry Peterson. binpac: A yacc for writing application protocol parsers. In *ACM IMC*, 2006.

[60] G. Papastergiou, G. Fairhurst, D. Ros, A. Brunstrom, K. Grinnemo, P. Hurtig, N. Khademi, M. Tüxen, M. Welzl, D. Damjanovic, and S. Mangiante. De-ossifying the internet transport layer: A survey and future perspectives. *IEEE Communications Surveys Tutorials*, 19(1):619–639, 2017.

[61] Sharvanath Pathak and Vivek S. Pai. Modnet: A modular approach to network stack extension. In *USENIX NSDI*, 2015.

[62] Zafar Ayyub Qazi, Cheng-Chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. Simple-fying middlebox policy enforcement using sdn. *ACM SIGCOMM*, 2013.

[63] Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. How hard can it be? designing and implementing a deployable multipath TCP. In *USENIX NSDI*, 2012.

[64] Alexander Rucker, Muhammad Shahbaz, Tushar Swamy, and Kunle Olukotun. Elastic rss: Co-scheduling packets and cores using programmable nics. In *APNet*, 2019.

[65] Vyas Sekar, Sylvia Ratnasamy, Michael K. Reiter, Norbert Egi, and Guangyu Shi. The middlebox manifesto: Enabling innovation in middlebox deployment. In *ACM HotNets*, 2011.

[66] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In *ACM SIGCOMM*, 2012.

[67] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Blindbox: Deep packet inspection over encrypted traffic. In *ACM SIGCOMM*, 2015.

[68] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina Argyraki, Sylvia Ratnasamy, and Scott Shenker. Resq: Enabling slos in network function virtualization. In *USENIX NSDI*, 2018.

[69] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. Hyperscan: A fast multi-pattern regex matcher for modern cpus. In *USENIX NSDI*, 2019.

[70] Zhaoguang Wang, Zhiyun Qian, Qiang Xu, Zhuoqing Mao, and Ming Zhang. An untold story of middleboxes in cellular networks. In *ACM SIGCOMM*, 2011.

[71] Shinae Woo, Eunyoung Jeong, Shinjo Park, Jongmin Lee, Sunghwan Ihm, and KyoungSoo Park. Comparison of caching strategies in modern cellular backhaul networks. In *ACM MobiSys*, 2013.

[72] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. Elastic scaling of stateful network functions. In *USENIX NSDI*, 2018.

Appendix A Rubik Built-in Abstractions

	Abstractions	Initialized Syntax and Parameters	Functionality and Semantics
Packet Data	`p.header`	`layout+layout` `AnyUntil((layout), cond)`	generate a header layout
	`p.temp, p.perm`	`layout`	the permanent/temporary data bound to the instance
	`p.seq`	`Sequence(meta,data,data_len)`	the sequence indexed by *meta* and filled with *data*
	`p.payload`	initialized by `p.header`	the payload of layer *p*
	`p.sdu`	initialized by `Assemble()`	the SDU passing to the upper layer
Inst. Table	`p.selector`	`[inst_key]` `([active_key],[passive_key])`	maintain the instance table by the key
Proto. State Machine	*s*	`PSMState(start, accept)`	define a PSM state *s*
	`p.psm`	`PSM(state set)`	build the PSM with the *state set*
	`p.psm.t`	`(`s_1 `>>` s_2`) + Pred(cond)`	define a transition *t* from s_1 to s_2 with condition *cond*
	`p.to_active/passive`	initialized by `p`	sending to active/passive side (connection-oriented)
Event Callback	`p.event.e`	`Assemble()`, `Callback(lay)`	an event *e* that assembles the sequence or poses *lay*
	`p.event_relation`	`(p.event.a, p.event.b)` `p.event.c >> p.event.d`	*a* happens before *b*, while *d* happens when *c* happens
Parse Tree	`st`	`Stack()`	define a stack *st*
	`st.`*lay*	*p*	define layer *lay* with parser *p*
	`st`	`(st.`*lay*1 `>> st.`*lay*2`) + Pred(`*pred*`)`	define *lay*2 as the next layer of *lay*1 with condition *pred*
Global Parameters	`p.cursor`	initialized by `p`	the current position (in byte) through the header parsing
	`p.cur_state`	initialized by `p`	the current PSM state of the instance
	`p.timeout`	a float number	the global keep-live threshold for the instance of `p`
	`p.psm.t.timeout`	a float number	the keep-live threshold of the instance in transition *t*

Appendix B Instructions and Expressions in Rubik's CFG

Instructions	Read	Write	Description
Assign(*reg*, *expr*)	*expr*	*reg*	modify *reg* with the result of *expr*
AssignSDU(*expr*)	*expr*	`SEQUENCE`	modify sequence with the result of *expr*
CreateInst()	-	`INSTTABLE`	create and insert the instance into instance table
DestroyInst()	-	`INSTTABLE`	remove the instance from the instance table
InsertSeq(*m*,*d*,*l*)	-	`SEQUENCE`	insert a sequence block (`meta=`*m*`,data=`*d*`,len=`*l*) into the sequence
Assemble()	-	`SEQUENCE`	assemble the continuous blocks in the sequence and pop it
Call(*expr*)	*expr*	-	invoke a callback function with the data *expr*
NextLayer()	-	-	jump into the next layer according to the parse tree
Expressions	**Read**	**Write**	**Description**
HeaderContain(*f*)	`HEADER`	-	check whether the parsed header contains field *f*
Payload()	`HEADER`	-	the content besides the parsed header from the packet
SDU()	`SEQUENCE`	-	the SDU assembled from the sequence
Contain(*key*)	`INSTTABLE`	-	check whether the instance with *key* is in the instance table

Appendix C Example Rubik Programs

C.1 IP Protocol Parser

The following code shows a complete IP protocol parser.

```python
class ip_hdr(layout):
    version = Bit(4)
    ihl = Bit(4)
    tos = Bit(8)
    tot_len = UInt(16)
    id = Bit(16)
    blank = Bit(1)
    dont_frag = Bit(1)
    more_frag = Bit(1)
    f1 = Bit(5)
    f2 = Bit(8)
    ttl = Bit(8)
    protocol = Bit(8)
    check = Bit(16)
    saddr = Bit(32)
    daddr = Bit(32)


class ip_temp(layout):
    offset = Bit(16)
    length = Bit(16)


def ip_parser():
    ip = Connectionless()

    ip.header = ip_hdr
    ip.selector = [ip.header.saddr, ip.header.daddr]

    ip.temp = ip_temp
    ip.prep = Assign(
        ip.temp.offset, ((ip.header.f1 << 8) + ip.header.f2) << 3
    ) + Assign(ip.temp.length, ip.header.tot_len - (ip.header.ihl << 2))

    ip.seq = Sequence(meta=ip.temp.offset, data=ip.payload[: ip.temp.length])

    DUMP = PSMState(start=True, accept=True)
    FRAG = PSMState()
    ip.psm = PSM(DUMP, FRAG)
    ip.psm.dump = (DUMP >> DUMP) + Pred(
        ((ip.header.dont_frag == 1) & (ip.temp.offset == 0))
        | ((ip.header.more_frag == 0) & (ip.temp.offset == 0))
    )
    ip.psm.frag = (DUMP >> FRAG) + Pred(
        (ip.header.more_frag == 1) | (ip.temp.offset != 0)
    )
    ip.psm.more = (FRAG >> FRAG) + Pred(ip.header.more_frag == 1)
    ip.psm.last = (FRAG >> DUMP) + Pred(ip.v.header.more_frag == 0)

    ip.event.asm = If(ip.psm.dump | ip.psm.last) >> Assemble()

    return ip
```

C.2 TCP Protocol Parser

The following shows the complete TCP protocol parser written in Rubik, including the header option parsing, the bi-directional buffering, the out-of-window exception handling, *etc*. We note that the code is formatted according to PEP8, which largely increases the number of LOC. Moreover, the definitions of header layout and auxiliary structures take about 40% of the LOC, which means the factual effort of writing this parser is even less than the number of LOC shows.

```
class tcp_hdr(layout):
    sport = UInt(16)
    dport = UInt(16)
    seq_num = UInt(32)
    ack_num = UInt(32)
    hdr_len = Bit(4)
    blank = Bit(4)
    cwr = Bit(1)
    ece = Bit(1)
    urg = Bit(1)
    ack = Bit(1)
    psh = Bit(1)
    rst = Bit(1)
    syn = Bit(1)
    fin = Bit(1)
    window_size = UInt(16)
    checksum = Bit(16)
    urgent_pointer = Bit(16)


class tcp_nop(layout):
    nop_type = Bit(8, const=1)

class tcp_mss(layout):
    mss_type = Bit(8, const=2)
    mss_len = Bit(8)
    mss_value = Bit(16)


class tcp_ws(layout):
    ws_type = Bit(8, const=3)
    ws_len = Bit(8)
    ws_value = Bit(8)


class tcp_SACK_permitted(layout):
    SCAK_permitted_type = Bit(8, const=4)
    SCAK_permitted_len = Bit(8)


class tcp_SACK(layout):
    SACK_type = Bit(8, const=5)
    SACK_len = Bit(8)
    SACK_value = Bit((SACK_len - 2) << 3)


class tcp_TS(layout):
    TS_type = Bit(8, const=8)
    TS_len = Bit(8)
    TS_value = Bit(32)
    TS_echo_reply = Bit(32)


class tcp_cc_new(layout):
    cc_new_type = Bit(8, const=12)
    cc_new_len = Bit(8)
    cc_new_value = Bit(32)


class tcp_eol(layout):
    eol_type = Bit(8, const=0)


class tcp_blank(layout):
    blank_type = Bit(8)
    blank_len = Bit(8)
    blank_value = Bit((blank_len - 2) << 3)
```

```


class tcp_data(layout):
    active_lwnd = Bit(32, init=0)
    passive_lwnd = Bit(32, init=0)
    active_wscale = Bit(32, init=0)
    passive_wscale = Bit(32, init=0)
    active_wsize = Bit(32, init=(1 << 32) - 1)
    passive_wsize = Bit(32, init=(1 << 32) - 1)
    fin_seq1 = Bit(32, init=0)
    fin_seq2 = Bit(32, init=0)


class tcp_temp(layout):
    wnd = Bit(32)
    wnd_size = Bit(32)
    data_len = Bit(32)


def tcp_parser(ip):
    tcp = ConnectionOriented()

    tcp.header = tcp_hdr
    tcp.header += If(tcp.cursor < tcp.header.hdr_len << 2) >> AnyUntil(
        [
            tcp_eol,
            tcp_nop,
            tcp_mss,
            tcp_ws,
            tcp_SACK_permitted,
            tcp_SACK,
            tcp_TS,
            tcp_cc_new,
            tcp_blank,
        ],
        (tcp.cursor < tcp.header.hdr_len << 2) & (tcp.payload_len != 0),
    )

    tcp.selector = (
        [ip.header.saddr, tcp.header.sport],
        [ip.header.daddr, tcp.header.dport],
    )

    tcp.perm = tcp_data
    tcp.temp = tcp_temp

    CLOSED = PSMState(start=True)
    SYN_SENT, SYN_RCV, EST, FIN_WAIT_1, CLOSE_WAIT, LAST_ACK = make_psm_state(6)
    TERMINATE = PSMState(accept=True)

    tcp.prep = Assign(tcp.temp.data_len, tcp.payload_len)
    tcp.prep = (
        If(tcp.header.syn == 1) >> Assign(tcp.temp.data_len, 1) >> Else() >> tcp.prep
    )
    tcp.prep = (
        If(tcp.header.fin == 1)
        >> Assign(tcp.temp.data_len, tcp.payload_len + 1)
        + (
            If(tcp.current_state == EST)
            >> Assign(tcp.perm.fin_seq1, tcp.header.seq_num + tcp.payload_len)
            >> Else()
            >> Assign(tcp.perm.fin_seq2, tcp.header.seq_num)
        )
        >> Else()
        >> tcp.prep
    )


    def update_wnd(oppo_lwnd, oppo_wscale, oppo_wsize, cur_lwnd, cur_wscale, cur_wsize):
        x = If(tcp.header_contain(tcp_ws)) >> Assign(oppo_wscale, tcp.header.ws_value)
        x += Assign(oppo_wsize, tcp.header.window_size)
        x += Assign(oppo_lwnd, tcp.header.ack_num)
        x += Assign(tcp.temp.wnd, cur_lwnd)
        x += Assign(tcp.temp.wnd_size, cur_wsize << cur_wscale)
        return x

    tcp.prep += If(tcp.to_active == 1) >> update_wnd(
        tcp.perm.passive_lwnd,
        tcp.perm.passive_wscale,
        tcp.perm.passive_wsize,
        tcp.perm.active_lwnd,
        tcp.perm.active_wscale,
        tcp.perm.active_wsize,
    )
    tcp.prep += If(tcp.to_passive == 1) >> update_wnd(
        tcp.perm.active_lwnd,
        tcp.perm.active_wscale,
        tcp.perm.active_wsize,
        tcp.perm.passive_lwnd,
        tcp.perm.passive_wscale,
        tcp.perm.passive_wsize,
    )

    tcp.seq = Sequence(
        meta=tcp.header.seq_num,
        zero_based=False,
        data=tcp.payload[: tcp.temp.data_len],
        data_len=tcp.temp.data_len,
        window=(tcp.temp.wnd, tcp.temp.wnd + tcp.temp.wnd_size),
    )

    tcp.psm = PSM(CLOSED, SYN_SENT, SYN_RCV, EST, FIN_WAIT_1, CLOSE_WAIT, LAST_ACK, TERMINATE)

    tcp.psm.orphan = (CLOSED >> TERMINATE) + Pred(tcp.header.syn == 0)
    tcp.psm.hs1 = (CLOSED >> SYN_SENT) + Pred(
        (tcp.header.syn == 1) & (tcp.header.ack == 0)
    )
    tcp.psm.hs2 = (SYN_SENT >> SYN_RCV) + Pred(
        (tcp.to_active == 1) & (tcp.header.syn == 1) & (tcp.header.ack == 1)
    )
    tcp.psm.hs3 = (SYN_RCV >> EST) + Pred(tcp.v.header.ack == 1)

    tcp.psm.buffering = (EST >> EST) + Pred(
        (tcp.header.fin == 0) & (tcp.header.rst == 0)
    )

    tcp.psm.wv1 = (EST >> FIN_WAIT_1) + Pred(tcp.v.header.fin == 1)
    tcp.psm.wv2 = (FIN_WAIT_1 >> CLOSE_WAIT) + Pred(
        (tcp.v.header.ack == 1) & (tcp.v.header.fin == 0)
        & (tcp.perm.fin_seq1 + 1 == tcp.v.header.ack_num)
    )
    tcp.psm.wv2_fast = (FIN_WAIT_1 >> LAST_ACK) + Pred(
        (tcp.v.header.ack == 1) & (tcp.v.header.fin == 1)
        & (tcp.perm.fin_seq1 + 1 == tcp.v.header.ack_num)
    )
    tcp.psm.wv3 = (CLOSE_WAIT >> LAST_ACK) + Pred(tcp.v.header.fin == 1)
    tcp.psm.wv4 = (LAST_ACK >> TERMINATE) + Pred(
        (tcp.v.header.ack == 1) & (tcp.perm.fin_seq2 + 1 == tcp.v.header.ack_num)
    )

    for i, state in enumerate(tcp.psm.states()):
        setattr(tcp.psm, f"rst{i}", (state >> TERMINATE) + Pred(tcp.header.rst == 1))

    tcp.event.asm = If(tcp.psm.buffering) >> Assemble()
    return tcp
```

C.3 GTP Stack

The following code builds a GTP stack (Eth→IP→UDP→GTP→IP→TCP) by composing the reusable parsers.

```
stack = Stack()
stack.eth = eth_parser()
stack.ip1 = ip_parser()
stack.udp = udp_parser()
stack.gtp = gtp_parser()
stack.ip2 = ip_parser()
stack.tcp = tcp_parser(stack.ip2)

stack += (stack.eth >> stack.ip1) + Pred(1)
stack += (stack.ip1 >> stack.udp) + Pred(
    (stack.ip1.psm.dump | stack.ip1.psm.last) & (stack.ip1.header.protocol == 17)
)
stack += (stack.udp >> stack.gtp) + Pred(1)
stack += (stack.gtp >> stack.ip2) + Pred(stack.gtp.header.MT == 255)
stack += (stack.ip2 >> stack.tcp) + Pred(
    (stack.ip2.psm.dump | stack.ip2.psm.last) & (stack.ip2.header.protocol == 6)
)
```

Appendix D Peephole Optimizations

D.1 Direct Fast Forward

Pattern: `CreateInst` → `InsertSeq` → `Assemble`

Analysis: `CreateInst` creates a new instance. As such, the sequence must be empty and `InsertSeq` will insert the first block, which will be directly ejected by `Assemble`. To this end, the insertion is redundant and the assembled data is identical to the payload of this packet. To this end, the insertion and assemble instructions can be removed, and all the reference to p.`sdu` in this code block can be replaced with p.`payload`.

Output: `CreateInst`, and references to p.`sdu` in this code block can be replaced with p.`payload`, *e.g.*, `NextLayer(SDU)` should be modified to `NextLayer(Payload)`.

D.2 Fast Forward

Pattern: `InsertSeq` → `Assemble`

Analysis: Fast Forward optimizes the assemble operations for existing instances. To be specific, we could make a fast peek to the sequence before we insert the block: if the sequence is empty and the block's meta is aligned with the sequence's window, this block can be fast forwarded, *i.e.*, passing the payload instead of SDU; otherwise the code maintains the same.

Output: `If(IsEmpty&IsAlign)` →(replace SDU with payload) → `Else` → `InsertSeq` → `Assemble`

D.3 Fast Assemble

Pattern: `InsertSeq` → `Assemble` and without any `NextLayer` and `Callback`

Analysis: The `false` branch of the Fast Forward optimization means the current sequence is not empty or the current block is not aligned with the window. We can further optimize this branch, if it has no external function call, *i.e.*, `NextLayer` and `Callback`. Specifically, if the packet sequence is implemented using linked list like `libnids`, `Assemble` that collects the continuous blocks will invoke several times of data copy. However, if there is no external function that needs the assembled data, such copy is useless and can be eliminated. On the other hand, if the packet sequence is implemented using ring buffer like mOS, the data copy is still necessary when there are holes in the sequence and memory compaction is performed. In such cases, `Assemble` instruction can be eliminated. Note that we cannot eliminate `InsertSeq`, because this block may be useful for next packets' assemble in other branches.

Output: Remove `Assemble`.

D.4 Fast Destroy

Pattern: `CreateInst` → `DestroyInst`

Analysis: If an instance is created and destroyed by the same packet, it means that such instance will not impact any permanent data, and all sequence and PSM operations are meaningless. As a result, we can eliminate such creation and deletion as well as most of the instructions between them, except `Callback` and `NextLayer`.

Output: A mostly empty instruction block except `Callback` and `NextLayer`.

Flightplan: Dataplane Disaggregation and Placement for P4 Programs

Nik Sultana John Sonchack Hans Giesen Isaac Pedisich Zhaoyang Han
Nishanth Shyamkumar Shivani Burad André DeHon Boon Thau Loo
University of Pennsylvania

Abstract

Today's dataplane programming approach maps a whole P4 program to a single dataplane target, limiting a P4 program's performance and functionality to what a single target can offer. Disaggregating a single P4 program into subprograms that execute across different dataplanes can improve performance, utilization and cost. But doing this manually is tedious, error-prone and must be repeated as topologies or hardware resources change.

We propose Flightplan: a target-agnostic, programming toolchain that helps with splitting a P4 program into a set of cooperating P4 programs and maps them to run as a distributed system formed of several, possibly heterogeneous, dataplanes. Flightplan can exploit features offered by different hardware targets and assists with configuring, testing, and handing-over between dataplanes executing the distributed dataplane program.

We evaluate Flightplan on a suite of in-network functions and measure the effects of P4 program splitting in testbed experiments involving programmable switches, FPGAs, and x86 servers. We show that Flightplan can rapidly navigate a complex space of splits and placements to optimize bandwidth, energy consumption, device heterogeneity and latency while preserving the P4 program's behavior.

1 Introduction

Different kinds of hardware can be leveraged to make networks programmable, including CPU-based servers running "software-ized" Network Functions (NFs), NPUs, FPGAs and programmable ASICs. Although these hardware targets have complementary strengths when it comes to performance, flexibility, and power utilization, it is difficult to combine their strengths. Many NFV frameworks are CPU-centric, and chaining services across diverse hardware usually treat the hardware's capabilities as a black box. This is partly because toolchains for NPUs, FPGAs, and ASICs are alien to most software developers.

Code, documentation, tests and data can be downloaded from: `https://flightplan.cis.upenn.edu`

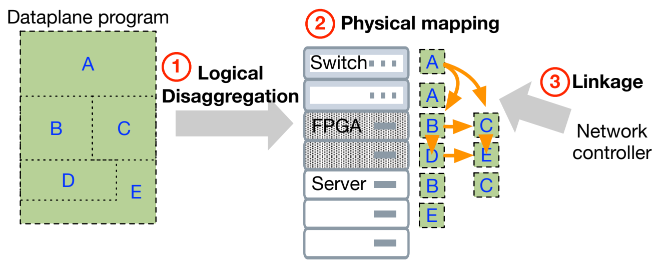


Figure 1: In this illustration, a program is split into 5 logical parts, A-E. ① A program is annotated with logical delimiters, manually or automatically. Flightplan splits the program into complementary parts using these annotations and provides coordination and linkage code between these parts, which the Flightplan control program configures at runtime. ② Each part of the original program is mapped to a physical device. In this illustration, A is mapped to Top-of-Rack (ToR) switches, B, C D, and E to network-attached FPGAs, and redundant instances of B, C, and E are mapped to execute on server CPUs. ③ The Flightplan control program can alter the program's linkage at runtime, to use different hardware targets, mitigate faults, or balance load.

While P4 is emerging as a common language for programming dataplanes across programmable ASICs, NPUs, FPGAs, and CPUs, P4 programs are limited to what can be run on a single target because they are programmed in an approach that maps a whole program to a single dataplane. This approach limits a dataplane program to what can be computed using a single target's resources and capabilities.

Through testbed experiments we measured how splitting a single P4 program into subprograms that execute across different dataplanes can improve performance, utilization and cost (§2, §7.2.3).

In principle, one could write a set of P4 programs that execute jointly across different dataplanes, combining their strengths. Further, P4 could be used as a convenient syntax for both NF authoring and for NF composition across different types of hardware. This set of P4 programs would be written to follow this pattern:

- If the dataplane program exceeds the resources provided by a single hardware target, then part of the program could be disaggregated to run on additional targets.
- If the dataplane program exceeds the capabilities provided by a *class* of targets, it could be disaggregated to run on two or more heterogeneous targets (e.g., ASIC and FPGA).
- Redundant spares can be provisioned for quick fail-over, leveraging the best available hardware for a given dataplane program.

However, writing distributed P4 programs manually is tedious and error prone. Current dataplane programming approaches lack abstractions for inter-program communication and mechanisms such as RPC [5] across different in-network functions. This complicates the implementation and composition of sophisticated in-network services because dataplane programmers are burdened with having to write and manage such coordination explicitly. It is also laborious to re-distribute the P4 program when new equipment or NFs become available.

We propose and evaluate Flightplan, a target-agnostic, P4-based programming toolchain that disaggregates P4 dataplane programs into a set of dataplane subprograms and runs them as a distributed system formed of several, possibly heterogeneous, dataplanes. It uses hardware performance and resource profiles to plan the allocation of subprograms onto dataplanes in the network on which to execute. The composite behavior of the resulting distributed dataplane program is the same as the original program; coordination and synchronization code is provided by the Flightplan runtime.

We reduce the P4 program, hardware performance profiles, and network topology constraints to a common rule-based formalism that Flightplan uses to map parts of the original program to dataplanes in the network. Through evaluation we show that it rapidly navigates a complex space of configurations which are then ranked according to the sought optimization criteria, and show that several disaggregated programs can be run simultaneously in the same network.

Figure 1 sketches our approach. It uses unmodified, commodity hardware and does not require changing the P4 language. A key enabler of this approach is that P4 enjoys toolchain support to target diverse hardware: ASIC [3], FPGA [23, 37, 38], NPU [24], and CPU [25]. These diverse targets' toolchains are based on P4's reference compiler [26], which we extend in our prototype. Unlike traditional NF service chaining [28], Flightplan works over diverse hardware.

We summarize our key contributions as follows:

- **Dataplane program disaggregation.** (§2) We propose the concept of dataplane program disaggregation and provide a motivating use-case. We show how disaggregation enables better utilization of existing resources.

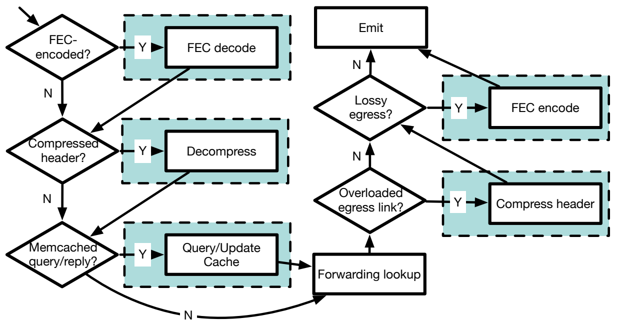


Figure 2: Our example dataplane program, `Crosspod.p4`, shown as a flowchart for compactness. The colored dashed rectangles surround functions that one might need to offload to other dataplanes in the network to free up resources on the switch, or because they exceed the computational abilities of the switch.

- **Automation for disaggregation.** (§3) We present a novel approach that partly automates dataplane disaggregation and does not require changes to the P4 language or to hardware targets and their toolchains. We extend the open-source P4 compiler to implement a program analyzer (§4) that discovers resource-use and dependencies that must be preserved once the program is split.
- **Flightplan runtime support.** (§5) We describe the requirement of in-dataplane and out-of-dataplane runtime support for disaggregated programs and how the runtime influences the process through which programs are disaggregated. We explore the design space by implementing 3 runtimes for Flightplan, offering different trade-offs between features and overhead.
- **Flightplan planner.** (§6) We implemented a prototype tool that generates configuration plans for disaggregated programs.
- **Evaluation.** (§7) We present a detailed evaluation of different aspects of this work, including testbed experiments involving heterogeneous hardware and microbenchmarks to measure resource transfer and disaggregation overhead.

2 Motivating Example: Crosspod

`Crosspod.p4` is an 800-line P4 program, sketched in Fig. 2, that will provide a running use-case. After describing how `Crosspod.p4` works, we describe the process of disaggregating it into several subprograms to run on three different classes of hardware.

We wrote `Crosspod.p4` to improve network reliability and performance using caching, compression, and forward-error correction in a way that is transparent to applications and users. Figure 3 shows its execution. Figure 4 shows two examples of disaggregation for this program.

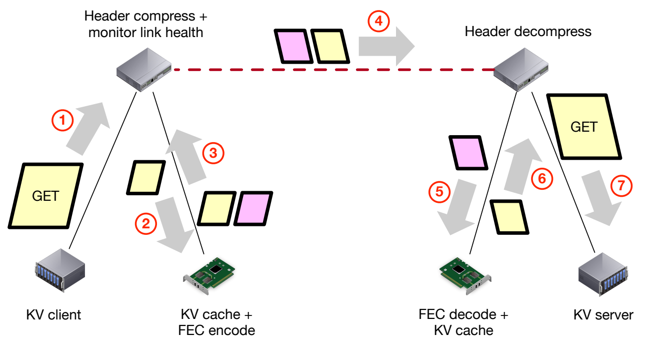


Figure 3: ① Key-Value (Memcached) client generates a GET request (yellow packet) which it puts onto the network for the KV server to respond to. ② A transparent, rack-level in-network KV cache is consulted. ③ In the event of a cache miss then the request is relayed onwards. The switch compresses its header, and upon detecting that the request shall cross a lossy link, it activates link-layer FEC which is computed in the network using reconfigurable hardware. ④ The KV packet and FEC-related redundancy (pink packet) cross the channel to the next switch. ⑤ Lost packets can be recovered during FEC decoding. ⑥ The packet's header is decompressed. A second inline KV can be consulted, to take pressure off the host-based KV server. ⑦ If all the caches were missed, then the request finally reaches the KV server, which sends its response back to the requesting client.

In-network functions. `Crosspod.p4` invokes a set of in-network functions to achieve its goal. Some of these functions are external to P4, and we implemented them to run on different types of hardware. In our example dataplane program, *reliability* is improved by (1) using forward-error correction (FEC) to mitigate faulty links; (2) application-specific caching to lessen congestion; and (3) header compression to lessen congestion. The *performance* of the network is improved by (1) reducing link utilization (through caching and header compression), (2) reducing latency (through caching closer to clients), and (3) reducing server utilization (through caching). In particular, caching is directed at Key-Value (KV) queries, a staple service in modern datacenter systems [10, 18, 21, 35].

Why Dataplane Disaggregation. In this example dataplane program, we combine in-network functions that cannot entirely be carried out within a single type of hardware for the following reasons: (1) **resources**: we cannot run the program entirely on a programmable switch ASIC because some of the functions (e.g., layer 2 FEC) exceed the computations that can be carried out in state-of-the-art devices, which typically do not include payload processing; (2) **performance**: we cannot run it entirely on an FPGA because this would severely constrain the throughput of this program. ASICs often support higher I/O bandwidth per chip and use less silicon for the standard dataplane switching operations, resulting in fewer or less expensive chips to handle the highest throughput data

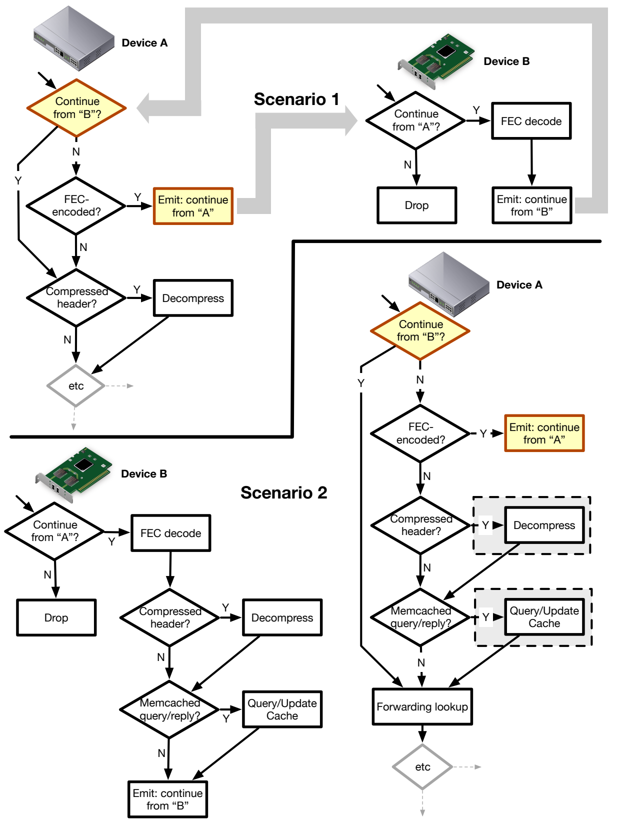


Figure 4: Two ways of splitting the program from Figure 2 between two devices: in **Scenario 1** a single function is offloaded from the switch (Device A) to an FPGA board (Device B) that immediately returns control back to the switch after executing the function; in **Scenario 2** a segment of the dataplane program is offloaded from the switch to the FPGA, which carries out several functions.

movement; (3) **expense**: even if we could place a program entirely on a single hardware dataplane, we do not want to use up resources unnecessarily and would prefer to move less-traversed code to a less expensive dataplane (e.g., an end-host). Conversely, if we have underutilized FPGAs, then we might prefer to use them rather than an end-host to save on power and cooling costs [14]. Last, (4) **availability**: we might want to failover quickly and autonomously from centralized monitoring and reconfiguration, by installing logic into the dataplane for self-management.

Why Automate Dataplane Disaggregation? Disaggregating a program involves i) deciding how to split it, ii) adapting the derivative subprograms to hand-over to one another, and iii) placing the subprograms on targets in the network.

Table 1 shows different characteristics of different hardware targets when executing the same function on the same workload. Automation spares the network operator from having to manually pick hardware combinations and track their

	Throughput (Gbit/s)			Power (W)
	Compression	FEC	KV Cache	
P.ASIC	9.07	-	-	110.5
FPGA	8.35	7.95	7.73	27.3
CPU	0.10	0.04	-	140.6

Table 1: Maximum Throughput and Average Power of network functions running on different hardware. Dashes indicate that we do not have implementations for network function to run on a particular target.

utilization.[1] In addition to power and performance, Flightplan can optimize for unit cost, utilization and latency.

The difficulty of this task is compounded by the likelihood that the choice of programs, topologies and target devices will change over time, requiring the whole process to be redone. If several programs are being disaggregated, then it becomes more challenging to optimize their placement jointly. Further, placement constraints and objectives may change over time, too: changing priorities over which links to protect with redundancy for example, or which traffic to compress, might require different function placements in the network.

This practical difficulty makes a strong case for automating dataplane disaggregation. But automation needs to be done carefully to avoid incurring the theoretical complexity of the automatic disaggregation problem. We estimate this to be exponential in the number of targets available on which to map subprograms and doubly exponential in the number of subprograms.[2] Flightplan uses heuristics to avoid this blow-up and our prototype also emits coarsening advice to opportunistically decrease the split granularity to better utilize the available hardware.

Deployment practicalities. The design and evaluation of Flightplan addresses the following practical considerations: (i) Multiple disaggregated programs can be used in the same network simultaneously: our evaluation in §7 describes how we ran 10 P4 programs in the same network, 6 of which were disaggregated, and of which 4 were different disaggregations of the same program. (ii) Upgrades can be done in a phased manner, as standard in deployment [34], by running the old and new versions of software simultaneously in the same network as described above. (iii) Debugging is done using standard techniques—inspecting packet traces, counters, etc.—complemented by using the Flightplan *control program* (§5) to conveniently query Flightplan-specific state of the disaggregated program across all the dataplanes on which parts of it are running. (iv) Failures can be detected and handled by Flightplan runtime support inserted into the dataplane itself or by remotely using the control program.

[1]More detailed information is provided in Table 3 in Appendix B.

[2]The full calculation is given in Appendix C.

3 Flightplan Overview

Flightplan produces a sequence of plans, consisting of a disaggregation of a dataplane program into multiple programs, and the allocation of these programs to dataplanes in the network. A plan targets the Flightplan *runtime* which provides the facilities to configure, start, and execute the disaggregated program. The role and design of Flightplan's runtime support is detailed in §5.

In this section, we outline all the inputs and outputs of Flightplan before going into more detail in the sections that follow. Figure 5 illustrates our workflow. ① We start with a P4 program and *segment* it. Segments are sequences of statements from the original program and provide the planner with the smallest granularity of program parts that it can then map to different targets. Figure 5 shows the program being decomposed into five segments as an example, labeled A-E. The `then` and `else` blocks of an `if` statement can go into separate segments, as hinted in the drawing in Figure 5 where segment A branches to B and C.

Programs are currently segmented manually, but this could be automated in the future. Segmentation consists of inserting additional lines in the program that are interpreted specially by our extension of the P4 compiler to delimit segment boundaries (§4.1.1). ② Our P4 compiler extension analyzes segments to generate a set of Prolog-like rules that expresses an *abstract program* completely automatically (§4.1.2). The abstract program consists of a DAG of segments, with P4 code replaced by the abstracted resources it depends on. Abstracted resources, such as tables and external functions, might not be available on all hardware or might have different capacity, power, throughput, and latency characteristics when invoked on hardware.

③ The *abstract resource semantics* is an additional set of rules used to check whether a dataplane satisfies a segment's resource needs. We use these rules to encode the performance profile which was summarized in Table 1. The contents and generation of rules is explained further in §4.

④ The planner is provided with a description of the network, including topology and device information, expressed in the same formal language used for our abstract semantics, and ⑤ objectives to optimize, consisting of variables changed by the abstract resource rules—such as Rate in Rule 1. ⑥ If the constraints can be satisfied for the given topology, abstract resource rules and segmented program, then the planner will lazily and exhaustively generate all plans to optimize objectives. A *plan* consists of three components. The *allocation model* describes how the abstract state—such as packet size and latency—is modelled to change as the program executes across dataplanes. This is used to understand the allocation that the planner has found. The *annotated program* consists of the original program with possibly coarsened segments, reflecting how the segments are going to be split into subprograms. In this example, segments B and C are united into F

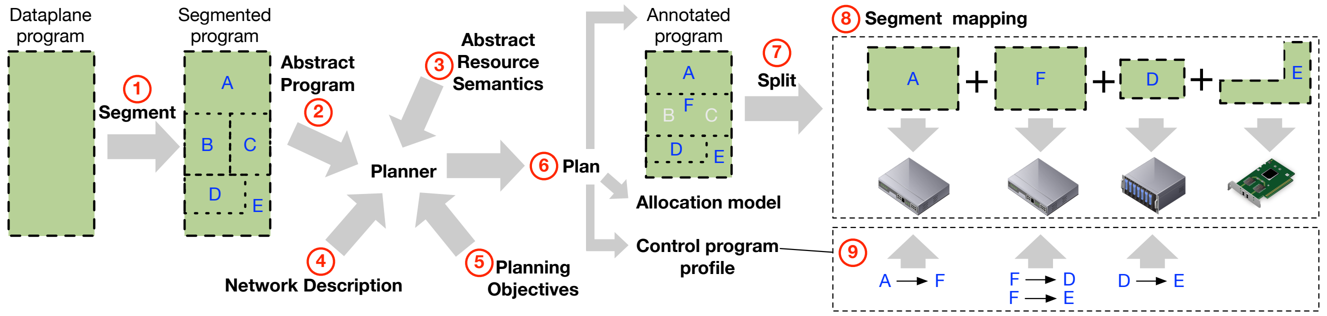


Figure 5: Flightplan's workflow, described in §3.

$$\frac{\text{CPU} \quad \text{Rate} < 2 \times 10^8 \quad \text{PacketSize} > 1000}{\text{header\_compress}} \left[\begin{array}{l} \text{Lat.} \mapsto \text{Lat.} + 7.4 \times 10^{-3} \\ \text{Rate} \mapsto \text{Rate} \times \frac{189.9}{194.75} \\ \text{once Power} \mapsto \text{Power} + 150\,\text{W} \\ \text{once Cost} \mapsto \text{Cost} + 5 \end{array} \right]$$

Rule 1: This rule states that we can execute Header Compression (HC) on packets *if* we are running on a CPU, and the throughput and packet size are within given bounds. We form different rules to describe the same function operating under different bounds—e.g., different Rate or PacketSize—and on different hardware targets, such as our FPGA implementation. If this rule can be used then the *effect* of executing HC in this instance is shown in the [...]-enclosed finite function on user-defined $\mathbb{R}$-valued variables. In addition to adding latency (Lat.), the function reduces Rate because of its compression of network traffic. 'once' indicates that a function is only executed once whenever using a specific target—in this case using an x86 server raises power estimate by 150 W regardless of how many segments are allocated to that particular target. 'Cost' is a normalized unit cost we use for different types of devices on our network. The [...]-enclosed function is allowed to mutate the planner's representation of the program's state, modelling the effect of invoking the resource.

for mapping to the same dataplane. Finally, the *control program profile* will be used by Flightplan's control program to configure the distributed program's runtime, start it, and query its state. The profile contains port, state information, and segment information, and is specific to a disaggregated dataplane program. ⑦ The annotated program is split into subprograms. This process involves augmenting each segment with a copy of the Flightplan runtime to configure, test, and hand-over between these programs. Some dataplane functions may be external to the P4 program, such as the FEC from the previous section. Flightplan treats such functions as black boxes, and it does not generate non-P4 code for specific targets, such as FPGAs or CPUs. ⑧ The program segments are compiled using the target-specific toolchain provided by the target's vendor. ⑨ The control program configures the Flightplan runtime of each target on which a segment is allocated by *linking* the segments together: i.e., indicating on which port to hand-over to its peer segments. Once the system's configuration is complete, the control program can start its execution.

Listing 1: Snippet from `Crosspod.p4` (§2) showing an example segmentation. Highlights show segmentation annotations in orange and resource-related syntax in green.

```
bit<1> compressed_link = 0;
bit<1> run_fec_egress = 0;
...
flyto(Compress);
// If heading out on a multiplexed link, then header compress.
egress_compression.apply(meta.egress_spec, compressed_link);
if (compressed_link == 1) {
    header_compress(forward);
    if (forward == 0) {
        drop();
        return;
    }
}
flyto(FEC_Encode);
check_run_FEC_egress.apply();
// If heading out on a lossy link, then FEC encode.
if (run_fec_egress == 1) {
    ...
    classification.apply(hdr, proto_and_port); // Sets hdr.fec.isValid()
    if (hdr.fec.isValid()) {
        encoder_params.apply(hdr.fec.traffic_class, k, h);
        update_fec_state(hdr.fec.traffic_class, k, h,
                         hdr.fec.block_index, hdr.fec.packet_index);
        hdr.fec.orig_ethertype = hdr.eth.type;
        FEC_ENCODE(hdr.fec, k, h);
        ...
```

4 Rules in Flightplan

Rules of different kinds play a central role in Flightplan. These rules are combined to describe the abstract semantics of a P4 program and how the resources it needs to use are satisfied by hardware targets in a network.

Steps ②, ③ and ④ in Figure 5 involve producing rules for the planner to use. This section describes the content of these rules and how they are generated.

An important design feature in Flightplan is that rules do not have to be encoded literally by users. As explained in this section, rules are either created automatically from P4 programs by the Flightplan analyzer, or are generated from a table that describes the network and profiles of devices in the network. This input is then automatically converted into rules based on Definite Horn clauses [9] that rely on a simple propositional language that is explained further in §I.

4.1 Abstract Program

Our P4 program analyzer turns a P4 program into an abstract form consisting of a set of Prolog-like rules. To better describe the generation of rules for step ② we first elaborate on step ①.

4.1.1 Program Segmentation

Step ① involves adding demarcating statements to P4 code. These statements consist of calls to the special function 'flyto()', passing it a unique name to be used for the new segment. A segment extends until the next flyto() or the control block's end, whichever is reached first. flyto() statements have no effect in P4, they are interpreted by our P4 compiler extension which we refer to as the *Flightplan analyzer*. These statements provide syntactic markers that define the sub-program granularity for the rest of the planning process. Enclosing the whole program in a *single* segment is a degenerate segmentation that will require the planner to place the program *entirely* on a single dataplane for execution while satisfying all other constraints.

The snippet in Listing 1 shows three segments beginning at lines 4, 14, and implicitly at line 1. The first segment is named FlightStart by default.

Segmentation is done manually by the programmer or automatically by a tool. In our prototype the programmer manually segments the code, and subsequently the planner will merge contiguous segments if they are to be placed on the same dataplane.

Once a segmentation has been made, the Flightplan analyzer discovers data-dependencies through static analysis of P4 code and determines whether a segment break is allowed for the intended runtime. Flightplan runtimes will be detailed in a later section.

4.1.2 Program Abstraction

For a given segmentation we next generate rules in step ②. We devised a framework for *abstract* semantics for P4 that is focused on resource dependence. Each segment is reduced to the resources it needs to execute, and everything else is abstracted away. In Listing 1 such resource-related syntax is highlighted in green.

The Flightplan analyzer is a P4 compiler extension that carries out static analysis of P4 code to gather which resources are relied upon by each segment. Resources include invocation of external functions and table lookups. The analyzer then automatically generates a Prolog-like [36] set of rules for that segment. We call the collection of segments' rules the *abstract program* (§4.1) derived from the original P4 program. The abstract program is emitted into a JSON file that the analyzer parses; users do not need to write, inspect, or change the contents of this file.

One rule is generated for each segment and describes the resources used along each path through that segment.

Rule 2 describes the third segment from Listing 1 (Lines 4-13). In this case there are three paths through the segment: lines (5-7,13), (5-9,12,13), (5-13), and Rule 2 is showing the path whose requirements subsume those of other paths. The analyzer emits the resource requirements of each path, and

$$\frac{\text{egress\_compression} \qquad \text{header\_compress} \qquad \text{drop}}{\text{Compress}}\ [\text{Id}]$$

Rule 2: Program segments are abstracted into rules showing resource dependence. This rule states that segment Compress can be mapped if the target dataplane can provide implementations of egress_compression, header_compress and drop: these are satisfied by means of other rules that are provided at input. We saw a rule for header_compress in Rule 1. Id is the identify function: using this rule does not mutate abstract state.

$$\frac{\text{CPU} \quad \text{Rate} < 8 \times 10^7 \quad \text{PacketSize} > 1050}{\text{fec\_decode}} \left[\begin{array}{l} \text{Lat.} \mapsto \text{Lat.} + 0.09 \times 10^{-3} \\ \text{Rate} \mapsto \text{Rate} \times \frac{64.47}{77.36} \\ \text{once Power} \mapsto \text{Power} + 150\,\text{W} \\ \text{once Cost} \mapsto \text{Cost} + 5 \end{array} \right]$$

Rule 3: Abstract resource rule for running fec_decode on a CPU.

the planner (§6) checks these are satisfied before allocating that segment to a prospective execution target.

4.2 Abstract Resource Semantics

In step ③ we supply the semantics of each program-used resource, such as calls to external functions, in terms of the measurable costs incurred for a program to use that resource on a specific hardware target. Such costs include latency, throughput, power, and the cost of the hardware.

In our prototype, the user encodes this information as a table of CSV entries, and can reuse this information across all invocations of Flightplan. A script then turns this table into a JSON encoding of the rules that are used by the planner. The measurements in these entries are derived empirically using the workflow described in §J. This involves carrying out profiling experiments that measure the characteristics of using resources on different hardware targets.

Rule 3 shows the characteristics of applying our FEC decoding function on a CPU. Rule 4 in §I shows a rule for applying Header Compression on an FPGA. Compared with Rule 1 it supports much higher throughput over smaller packets. Our table can be refined to include more details about the specific CPU and FPGA parts that were used, and capture other information about the target or the workload, without changing our general approach.

4.3 Network Representation

In step ④, the last set of rules states facts about the topology, the devices it comprises, and their ports. For example, that the proposition "CPU" holds on a specific network element, or the bandwidth limit of a specific port.

For a given port π, we gather facts in a set called π_{Provides}. We also associate constraints with π that must be satisfied for π to be crossed. We call this set π_{Requires}. [3]

[3] Examples of π properties are given in §I.

In our prototype we use JSON to encode the network's topology and the capabilities of each device and port. We reuse this information across all our invocations of Flightplan, and it only need to be changed when the network hardware or topology change.

5 Flightplan Runtime Support

A disaggregated program cannot function without runtime support. At the very least, the runtime provides the gluing code to *link* different parts of the disaggregated programs together for the computation to flow through them as it would in the original program.

The choice of runtime needs to be made first since it influences the rest of the process. This choice is communicated to the analyzer and splitter (§6.4). In Flightplan, the following information is in scope for the runtime to manage: (1) runtime metadata, such as (1a) which part of the program needs to be executed next, (1b) values of live variables to be preserved across dataplane hand-overs, (1c) state related to in-dataplane failure detection and handling, and (2) switch metadata—such as ingress and egress ports, since these might be read or written during the program's execution, including table lookups and actions.

There are different choices of runtime features and ways of implementing them. Our different runtime implementations show how the core idea in Flightplan—that of programmable-dataplane disaggregation—spans a design space and not a single implementation.

5.1 Runtime Design

Runtime support for Flightplan consists of two components, the first running outside the dataplane and the second running inside it. The first is a control program that configures the latter and queries its state. Configuration information is stored in register values and table entries used by the second component to support the execution of each segment of the distributed P4 program. In our prototype, the control program—called `fpctl`—emits dozens of commands to the P4 controller, which in turn interacts with the dataplane. The inputs to `fpctl` consist of the network topology, the control program profile, and a command (such as `configure` or `start`) and its parameters.

The second component runs inside the dataplane and must be written in P4. It is combined with each constituent of a distributed P4 program. In Fig. 1, each constituent A-E would be instantiating the same runtime, albeit in possibly heterogeneous dataplanes.

5.2 Runtime Diversity

We developed three runtimes, each representing a different point in a design space. The three are called 'Full' (§5.3), 'Headerless (IPv4)', and 'Headerless'. We use `fpctl` to interact with all of them. All our implementations use standard features of $P4_{16}$ and target the simple_switch BMv2 target [25], whose features intersect with most P4 devices.

The Full runtime uses a special Flightplan header, but the other two do not. Our IPv4 headerless approach steals bits from the IPv4 Fragment Offset header for Flightplan state. However, we found that this was not always usable since not all of our network traffic has IP headers—specifically the FEC parity frames lack these. This spurred us to develop a completely headerless approach, described in §5.4.

Recalling the example from Listing 1, the Full runtime allows us to preserve the value of `meta.egress_spec` by serializing it into the Flightplan header during hand-over. We can alternatively use the Headerless runtime since, while preserving switch metadata such as `meta.egress_spec` is generally not possible in our Headerless approach, we encode this information in the wiring and table entries. This limits the behavior of the original program, trading off flexibility for less overhead, since the Headerless approach essentially creates a circuit through dataplanes.

For a further example of how the Full runtime allows more flexibility for segmentation, we could add a flyto() between lines 6 and 7 in Listing 1. The Full runtime allows us to hand-over the value of `compressed_link` from Compress to the new segment, but the Headerless runtime would not be able to support this segmentation.

5.3 Full Runtime

The Full runtime provides the most flexibility and resilience among our runtimes. It uses a custom header to accommodate all of the features (1a-c,2) listed earlier in §5. The header's definition is provided in §D.

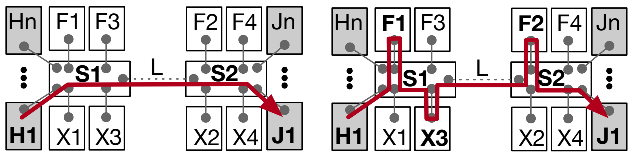
The header consists of two kinds of fields: scratch space in which metadata and program variables are serialized and state fields for encoding status flags and values, such as which segment from the original program is to be executed next.

Part of the distributed program might be unreachable because of network or node failures, thus compromising the overall program's execution. This runtime implements a feedback loop between connected dataplanes to push fault detection and handling into the dataplane. The details are in §E.1.

`fpctl` can be used to query the runtime's state, for instance to find out if it is close to meeting a fail-over threshold, or if it has failed-over. It can also overwrite this state and force a fail-over remotely.

5.4 Headerless Runtime

This approach does not encode any state about the ongoing computation across dataplanes. Consequently this runtime provides less flexibility than the 'Full' approach: metadata is not preserved, it does not support in-dataplane failure detection and handling, and values of live variables are not preserved—thus segments must be coarser to avoid needing to hand-over the variables' values. This runtime is described further in §E.2.



(a) Shortest path from H_1 to J_1. (b) A detour path from H_1 to J_1.

Figure 6: Abstract network used for running example in §6.

To compensate for the lack of header, we rely on the ingress and egress port information. An interesting consequence of the Headerless approach's feature-paucity is that it can be made to work with older SDN switches that are not P4 programmable. To enable this, `fpctl` generates flow-table configurations from the control program profile. We report on the use of non-P4 SDN switches in our testbed evaluation.

6 Flightplanner

Flightplan combines graph-based and formal methods to find execution plans for disaggregated programs over dataplanes in the network.

Execution of the disaggregated program occurs along paths in the network, and therefore the planner needs to be told about the network's topology and hardware capabilities. Fig. 6 resembles our physical testbed and is used in this section to help explain an execution plan. In this network, S_i are switches, F_i are network-connected FPGA boards, X_i are CPU-based network elements that can run P4 programs, and H_i and J_i are servers to which P4 programs cannot be offloaded.

Flightplan explores resources around switches to carry out detours in the forwarding along dataplanes onto which some of the computation can be offloaded as shown in Fig. 6b. In this example S_1 offloads computations to F_1 and X_3, and S_2 offloads to F_2.

6.1 Abstract Program State

The planner maintains a small amount of state as it explores plans. This state consists of values for a special set of variables V that appear in rules, such as PacketSize and Rate. Using these variables we form Bound expressions such as PacketSize > 1000 from Rule 1.[4]

The *abstract program state* θ consists of a total map $V \to \mathbb{R}$ that encodes the value of each V at one instant during the program's execution. The values of V can be transformed by rules in their $[\ldots]$-function, while the values of propositions can only be derived through proof, as will be explained shortly. During the planner's execution, all Bound expressions are ground, making the evaluation process straightforward.

6.2 Proof-based Segment Allocation

Producing a plan involves two kinds of inference: (i) deciding whether a given dataplane can execute a given segment in

[4]Further details of our modeling language are in §I.

$$\mathrm{R}2\,\dfrac{\mathrm{R}5\,\dfrac{\theta,\pi_{\mathrm{Prov.}}\dfrac{}{\mathrm{CPU}}}{\text{egress\_compres.}} \qquad \mathrm{R}1\,\dfrac{\theta,\pi_{\mathrm{Prov.}}\dfrac{}{\mathrm{CPU}} \quad \mathrm{Rate} < 2 \times 10^8 \quad \mathrm{PacketSize} > 1000}{\text{header\_compress}} \qquad \mathrm{R}6\,\dfrac{\theta,\pi_{\mathrm{Prov.}}\dfrac{}{\mathrm{CPU}}}{\text{drop}}}{\text{Compress}}$$

Proof 1: Flightplan may allocate segment Compress to π only if a proof can be derived using θ, program+resource+network rules and π_{Provides}.

a way that satisfies all related constraints; and (ii) finding a plan—a sequence of dataplanes over which all of a program's segments are executed. This section describes how item (i) is done in Flightplan. The next section builds on this to describe (ii).

For a given port π (§4.3), to decide whether a segment Seg can be mapped to π's dataplane, we need to do two things. First, ensure that all of π's constraints are satisfied. For example, the current transmission rate must not exceed the ports limit. Second, we use facts provided by π to ensure that Seg is *derivable* from the rules we have available. For example, all external functions called in Seg must have viable implementations once they cross that port.

This derivation involves building a Prolog-style formal proof. For example, if $\pi_{\text{Provides}} = \{\text{CPU}\}$ then Compress is derivable as shown in Proof 1 if θ satisfies the bound-related constraints of the rules used in this derivation.

In addition to obtaining assurance that a target can execute a segment, we use proofs to compute the transformations of abstract program state. This involves composing the $[\ldots]$-enclosed functions by doing a post-order traversal of the proof tree, then applying this function to the search state θ to obtain the new search state. For Proof 1 the state transformation γ is $\gamma = \gamma_{\mathrm{R}2} \circ \gamma_{\mathrm{R}6} \circ \gamma_{\mathrm{R}1} \circ \gamma_{\mathrm{R}5}$

6.3 Plan-finding

Given a DAG of abstract program segments (§4.1) we search for a succession of dataplanes that packets can be made to traverse such that all program segments can be executed over those packets. Since devices in the same class are identical, we factor the solution space by the different device classes to avoid returning quasi-duplicate solutions. The user can choose whether they want the best solution—by having the tool explore all possibilities. Alternatively, they are given the solution found using a simple greedy heuristic on optimization objectives—for example, by choosing the next dataplane that least increases latency.

The network operator needs to provide three additional pieces of information: (i) an initial abstract program state θ_0 (§6.1), (ii) the switch on which the disaggregation program's execution will be centered (e.g., S_1 in Fig. 6), and (iii) the set of devices to which the switch may offload to (e.g., $\{F_1, F_3, X_1, X_3\}$ in Fig. 6).

The planner carries out a breadth-first search while attempting to allocate segments as described in the previous section. At each hop it updates the abstract state using γ to compute an approximation of resource-usage across the network. This will be used to evaluate constraints in the rest of the potential plan.

When a plan is found, it is converted into the three outputs shown in Figure 5. First, contiguous segments that are to be mapped to the same target are unified into larger, coarser-grained segments. Second, the *allocation model* is produced by emitting the trace of abstract program states and the mapping from segment names to targets in the network. Finally, a profile is produced for use by `fpctl`—Flightplan's control program—to configure, start, and query the disaggregated P4 program. A profile consists of a mapping of generated P4 programs to a subset of the network. Flightplan uses the profile to configure a dataplane target through the target-specific control plane interface.

6.4 Program Splitting

Next we generate a separate, well-formed and self-contained P4 file for each segment. The program splitting phase performs three tasks: 1) extract the P4 code from each segment, forming subprograms; 2) analyze the subprograms to gather runtime-related context, such as variables whose values must be included in the hand-over between dataplanes; 3) inject runtime-dependent code for handing-over between subprograms. These are described further in §K. The Flightplan analyzer prototype also emits split programs satisfying points 1 and 2. The interfacing to the runtime can be automated in the future.

7 Evaluation

We evaluate Flightplan using virtual and physical networks to answer various questions about its features and the implications of disaggregation choices.

7.1 Scale, Overhead and Disaggregation

We use a virtualized network (§7.1.1) to test logical qualities of a Flightplan deployment, using P4 applications both that we wrote and from third-party sources.

To add realism to our experiments, we implemented a generator for complete configurations of fat-tree networks [1] to run on our setup, which is built on Mininet [19] and BMv2 [25]. We used $k = 4$ in this evaluation.

We implemented routing logic for this topology in a P4 program called `ALV.p4`. It provides a baseline P4 program that implements minimal functionality in the network—routing. We embedded it in four other P4 programs: (i) `Crosspod.p4` (§2), (ii) `firewall.p4`, (iii) `qos.p4` and (iv) `basic_tunnel.p4`. The latter three are third-party open-source programs from the P4 tutorial repository [27].

Initially our virtual network had `ALV.p4` executing on all switches. We then installed the other P4 programs on some

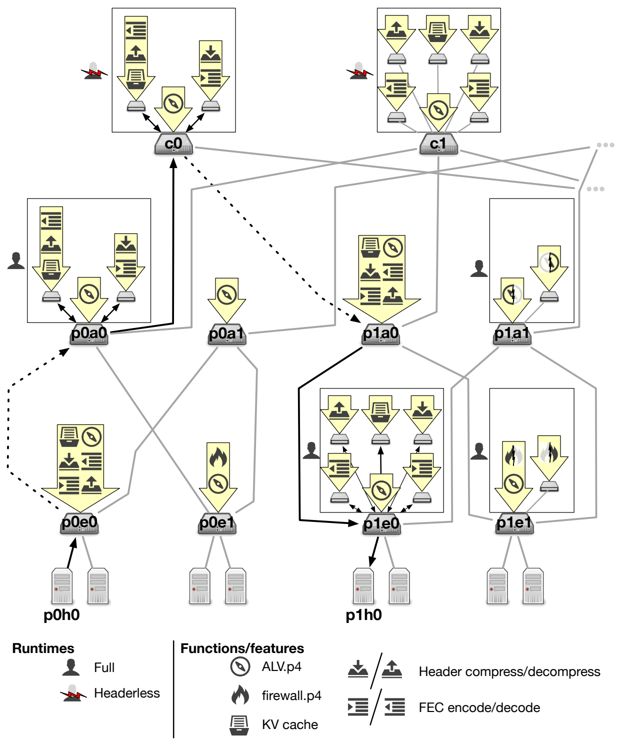


Figure 7: Part of our fat-tree network showing core routers (cN) and pods (pN[a,e,h]M) containing aggregation and edge switches, and hosts. Configured as explained in §7.1. Yellow arrows show which (sub)programs are installed on each device. Links between devices are shown in grey, except for links traversed by packets flowing between p0h0 and p1h0, which are shown in black. Dotted lines show faulty links.

switches and disaggregated them in various ways and to use different Flightplan runtimes, as described next. All disaggregated P4 programs, including those of third-party programs, were tested for behavioral correctness by checking that they produced similar results as the original programs.

7.1.1 Flightplan deployment example

Fig. 7 shows part of our network and the variety of P4 programs we ran simultaneously on different switches in the same network. Switches p0a1 (`ALV.p4`), p0e1 (`firewall.p4`), p0e0 and p1a0 (both `Crosspod.p4`) run non-disaggregated P4 programs, while all other switches run disaggregated programs. Of the latter, c0 and c1 use the headerless Flightplan runtime, and the remainder use the Full runtime.

Switches that run disaggregated programs are shown with an adjacent box in Fig. 7 showing the supporting devices on which parts of the original program were executed. For example, p1a1 carries out some of its table lookups locally, while others are offloaded to an associated device. Different switches can have different numbers of associated devices over which a dataplane program can be disaggregated—for example, c0 has 2 while c1 has 5. The presence and resourcing of such devices is decided by the network operator. In this part of the evaluation we treat all such devices as being identical, but in the next section we distinguish between heterogeneous devices based on their resources and capabilities.

We also disaggregated `qos.p4` and `basic_tunnel.p4` and tested performance overhead in our Mininet-based setup. We found that the lower-bound overhead to client-perceived RTT was 8.2%. A more accurate measurement using a hardware-based experiment, but on a simpler topology, is given in §L.1.

7.1.2 Network Scalability and Operation

The network from Fig. 7 helps demonstrate two features. First, it shows how the use of Flightplan scales with network size. Since we constrain Flightplan's planning scope to only resources adjacent to a switch—as illustrated in Fig. 1—Flightplan's scalability is independent of network size in this network topology. Thus planning can scale to a network containing a large number of switches.

Second, it shows that different Flightplan runtimes and disaggregations can operate simultaneously in the same network, and that these can be configured and started independently of each other, to deliver the practical features described at the end of §2.

7.1.3 Overheads

We compare program-level overheads of dataplane disaggregations. These include overheads on: the network due to header inclusion, port count, data memory (total register bits, number of tables and their entries) and code memory (code from the runtime, and extra branching because of splitting). Device-level overheads—such as those on throughput and power—are evaluated in §7.2 based on testbed experiments.

Table 2 shows overheads for two sets of disaggregations of `Crosspod.p4`: one set using the Full runtime and the other using Headerless. In our prototype, the in-network programs do not respond to MTU path discovery, therefore to use the Full approach in general we needed to lower the MTU size to provide headroom for the Flightplan header. This uniformly reduces network capacity by 2.4% to create enough headroom.

Program-level overhead can be calculated before compilation, independent of toolchain. The general analysis of both runtimes' overheads is provided in §F. If a P4 program is split to run across several P4 devices, we can calculate the overhead introduced onto the network and the devices independently of the devices' toolchains.

7.2 Testbed Evaluation

We use a physical testbed deployment to measure device-level resource and performance impact of Flightplan-based disaggregation. This evaluation is based on the Headerless approach, which gives us a lower-bound on overhead among our runtime approaches. Using `Crosspod.p4` (§2), we: (i) evaluate the Flightplan planner, and (ii) measure the end-to-end performance and power consumption of a heterogeneous dataplane running Flightplan-generated splits of `Crosspod.p4`.

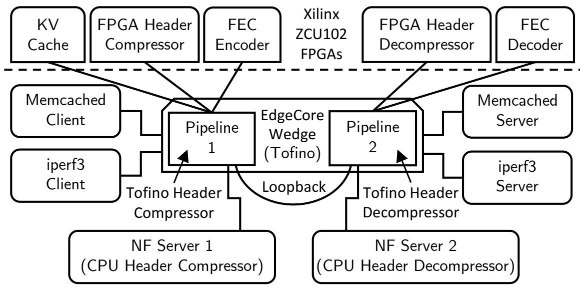


Figure 8: Evaluation testbed

7.2.1 Experimental setup.

Fig. 8 illustrates our testbed. It contains an EdgeCore Wedge 3.2-Tbps switch with a two-pipeline Barefoot Tofino P4 ASIC, five 4×10-Gbps Xilinx ZCU102 FPGA boards each with one Xilinx XCZU9EG FPGA, four traffic generation servers with 8-core Intel Xeon 2450 CPUs, and two network function servers with 10-core Intel Xeon Silver 4114 CPUs. In our configuration, the Tofino pipelines act as two independent switches, `S1` and `S2`, that service different ports and each have their own dedicated resources (SRAM, TCAM, etc.). All packets between `S1` and `S2` traverse a physical 10-GbE cable. We also evaluate a legacy scenario, by swapping the Wedge for an Arista 7050-QX32 with a Broadcom Trident II ASIC and OpenFlow support.

Crosspod.p4 plans. Our benchmarks focus on three plans output by Flightplan, designed for different objectives.

- **"Maximum Performance"** optimizes end-to-end performance by placing each function on the fastest available platform, i.e., header compression on the Tofino and all other functions on FPGAs.
- **"Resource Saver"** offloads compression from the Tofino to the FPGA to save compute and memory resources.
- **"Legacy Extender"** uses the Trident II in place of the Tofino to achieve functional equivalence with an OpenFlow switch.

Measurements. We benchmark four axes of performance: throughput, packet loss, latency, and power consumption. Throughput is measured at the application layer, using either `iperf3` or DPDK-pktgen. We measure packet loss using the Tofino's port counters. For latency, we use a simple telemetry function for the Tofino that timestamps (nanosecond precision) each packet ingressing or egressing out of monitored ports and clones a digest to a collection server. Finally,

	Feature	Runtime=**Full**									Runtime=**Headerless**								
Net.	Header (b)	288									0								
	Ports [*D*]	1-2 [3]			1-5 [6]						2 [3]			5 [6]					
	Ex. in Fig. 7	p0a0			p1e0						c0			c1					
	Seg ID	0	1	2	0	1	2	3	4	5	0	1	2	0	1	2	3	4	5
Dataplane	Tables	6			6						5	1	1	5	1				
	Entries (b)	179	89	89	445	89					154	22	22	230	22				
	Registers (b)	547	309	309	1261	309					4			4					
	Ctrl Struct	6	+2	+2	6	+2					2	+1	+1	2	+1				
	Externs	0	3	2	0	1					0	3	2	0	1				

Table 2: Overheads incurred by different disaggregations of `Crosspod.p4`, organized into **Network** and **Dataplane** overheads. The *Externs* row does not show overheads, but serves to show the distribution of extern function invocations across the splits. *D* is the number of segments. Each segment is given a separate identifier in the *Seg ID* row to distinguish them in the rows below. *Ports* is the number of switch ports that are required: the Full runtime's use of a Flightplan header allows the use of a single port (connecting to a single supporting device), while the Headerless runtime requires an exact number of ports. *Entries* gives the total size of all Flightplan table entries, in bits. *Ctrl Struct* is a measure of code complexity of the transformed program, counting the number of conditional statements introduced by the disaggregation in addition to the runtime's code.

we measure power consumption of each device at the outlet, polling at a 200ms interval.

End-to-end benchmarks. We use workloads that mix two types of flows: 1) large TCP flows (`iperf3`) representing traffic with high bandwidth demands and 2) UDP Memcached request streams representing latency-sensitive traffic. This workload models the bimodal distribution of packet sizes and traffic patterns in datacenters [4, 32]. We measure throughput using `iperf3` and calculate latency as the difference between the time at which a packet ingresses from its source host and the time at which it egresses to its destination host.[5]

7.2.2 Flightplan Planner

We used the Flightplan planner to analyze the solution space for `Crosspod.p4`. Our evaluation involves 239 rules, divided as follows: 140 are profile rules including Rules 1, 3, and 4; 68 are network rules (including those in Fig. 17), and the rest are program rules (including Rule 2).

The network is the one shown around c1 in Fig. 7. We used this to explore program-segment mappings to the switch and its supporting devices in such an arrangement. We ran different variants of the experiment to contrast the implications of using different types of equipment, following the plans described in §7.2.1.

Fig. 9 summarizes our results, and we evaluate the best-performing plans on our physical testbed in the next section. To avoid clutter we exclude "Resource Saver" because of its hybrid nature between the other two categories, and instead we show more extreme forms of "Legacy Extender" in the "Server Offload" family. These rely on a single switch and a pool of CPU-based servers. The latency advantage of "Server Offload (Tofino)" relative to "Server Offload (Arista)" seems too optimistic, and we believe that might be because of a lack of accuracy in the model we use.

[5]As well as the end-to-end experiments on `Crosspod.p4`, we also separately evaluate its constituent in-network functions end-to-end in §A.

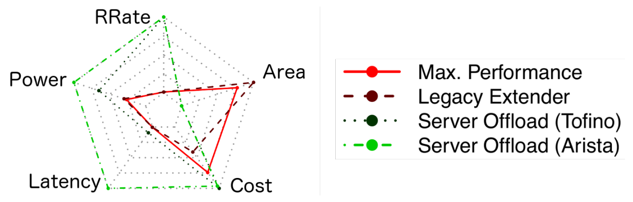


Figure 9: Best-rated plans found by Flightplan, described in §7.2.2. In each dimension, less is better. Each dimension is normalized by the maximum value from all plans. 'RRate' is the inverse of the bandwidth of the distributed program. 'Cost' refers to hardware costs and excludes running costs, which are captured by 'Power'. 'Area' refers to FPGA-resource usage.

7.2.3 Crosspod.p4 Benchmarks

The goal of `Crosspod.p4` is to improve the performance of applications bottlenecked by inter-rack links suffering from congestion and partial failure [39]. We evaluate how Flightplan's `Crosspod.p4` plans achieve this goal in a network with a 10 Gbps inter-rack link.

End-to-end performance. First, we evaluate the **"Maximum Performance"** `Crosspod.p4` plan. Fig. 10 shows application-level performance in a series of trials where the network functions are activated one by one. The figure plots the latency and success of GET and SET requests to the Memcached KV store and the throughput of TCP `iperf3` sessions, on two client/server pairs, as illustrated in Fig. 8.

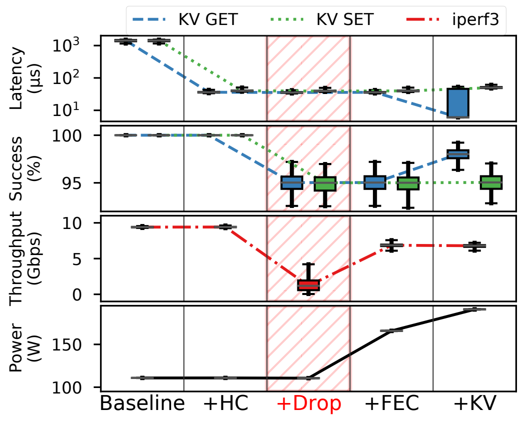


Figure 10: Throughput, latency and power utilization of a disaggregated `Crosspod.p4`. In the top and bottom panels, lower values are better. In other panels, higher values are better. Boxes show data quartiles and whiskers show 10th and 90th percentile, aggregated over 5 repetitions.

The leftmost panel in Fig. 10, labeled `Baseline`, shows application performance with no network functions enabled. Here, the problem is congestion, caused by the `iperf3` data transfers that saturate the inter-rack link. Because of the congestion, application-level Memcached latency is high.

To reduce the impact of congestion, we enable header compression of the `iperf3` traffic. This reduces the load on the bottleneck inter-rack link, allowing queues to drain and thus reducing latency. As the `+HC` column of Fig. 10 shows, header compression reduces Memcached request latency by 97% without impacting `iperf3` throughput. Since Flightplan maps the compression and decompression functions to the switch pipelines, no new network devices are required for this addition, and power consumption does not noticeably increase.

To study FEC, we first model partial link failure by enabling a dropping function in the switch that drops 5% of packets at random on the inter-rack link. The middle panel of Fig. 10, labeled `+Drop`, demonstrates application-level effect: a reduction in throughput to 1.19 Gbps and a loss of 5% of Memcached packets. The FEC encoder and decoder FPGAs are then introduced to protect TCP traffic on the link. It restores 57% of the lost TCP throughput, as the `+FEC` phase of Fig. 10 shows. Power consumption increases by about 55 W, reflecting the addition of the two FPGAs.

Finally, we introduce the Key-Value (Memcached) cache network function FPGA on the client side of the inter-rack link. The `+KV` column shows that the approximate 60% cache hit rate causes median latency of GET packets to reduce to 7 µs (while leaving the upper quartile above 40 µs) and restores successful responses to about 60% of previously dropped requests. SET latency and success are unaffected because the cache is inline, so these packets must traverse the shared link. Power consumption increases by another 25 W due to the addition of one more FPGA.

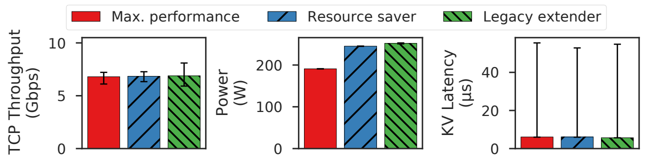


Figure 11: Application performance and network power consumption for three different `Crosspod.p4` plans. Bars mark medians, whiskers 5th and 95th percentile.

Alternate plans. Next, we evaluate the viability of other Flightplan-generated `Crosspod.p4` plans. As Fig. 11 shows, the **Resource Saver** and **Legacy Extender** provide almost identical application-level benefits. All three plans improve performance in the face of packet loss and congestion, increasing TCP throughput from 1.19 Gbps to 6.25 Gbps and decreasing median KV's GET request latency to under 10 µs.

Although all three plans are viable from the application's perspective, they offer different benefits for network operators. As Figure 11 shows, the power consumption of **maximum performance** is 50 W lower than the alternatives because it maps HC to the Tofino, thus requiring two fewer FPGAs.

On the other hand, by offloading HC to FPGAs, **Resource Saver** frees compute and memory resources in the Tofino. Most significantly, it reduces the number of pipeline stages from 10 to 4, the utilization of stateful ALUs from 25% to 2%, the number of tables from 46 to 11, and the utilization of SRAM by approximately 600KB per 1024 concurrent flows.

Finally, by targeting the Trident II, **Legacy Extender** lets OpenFlow switches achieve similar functionality and application-level performance as P4 switches. This provides a path to deploying `Crosspod.p4` in networks with limited programmability [32] or at a lower budget.

8 Related Work

Compared to Active Networking [2, 15, 33] and related recent work such as TPP [17], Flightplan leans towards performance and safety at the expense of flexibility: computation allocated to each dataplane is established at compile time, rather than being packet-carried.

Automated approaches to software splitting [6, 13, 22, 30] are usually done for vulnerability mitigation, whereas Flightplan is directed at improving performance and utilization. Closer to Flightplan, Floem [29] focuses on CPU-NIC co-processing of network applications, but Flightplan focuses on packet processing across distributed dataplanes which do not necessarily include CPUs.

As in network calculus [20] (NC) our reasoning technique for plans is concerned with flows at steady-state. Flightplan allows using arbitrary functions to describe the transformation γ of each rule on the abstract state, not only NC's operators. Future work can improve the modeling precision of our approach and generalize to better characterize workload-sensitive functions such as the in-network cache.

Acknowledgments. We thank Heena Nagda, Rakesh Nagda, our shepherd Nate Foster and the reviewers for their feedback. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contracts No. HR0011-19-C-0106, HR0011-17-C-0047, and HR0011-16-C-0056. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA.

References

[1] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A Scalable, Commodity Data Center Network Architecture. *SIGCOMM Computer Communication Review*, 38(4):63–74, August 2008.

[2] D. Scott Alexander, Marianne Shaw, Scott M. Nettles, and Jonathan M. Smith. Active Bridging. *SIGCOMM Computer Communication Review*, 27(4):101–111, October 1997.

[3] Barefoot Networks. Barefoot Tofino. `https://www.barefootnetworks.com/technology/`, 2016.

[4] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Understanding Data Center Traffic Characteristics. In *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking*, WREN '09, pages 65–72, New York, NY, USA, 2009. ACM.

[5] Andrew D. Birrell and Bruce Jay Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[6] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting Applications into Reduced-privilege Compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 309–322, Berkeley, CA, USA, 2008. USENIX Association.

[7] IOS Cisco. Quality of Service Solutions Configuration Guide. *Congestion Avoidance Overview. Cisco, Accessed*, 18, 2014.

[8] Carlos R Cunha, Azer Bestavros, and Mark E Crovella. Characteristics of www client-based traces. Technical report, Boston University Computer Science Department, 1995.

[9] William F. Dowling and Jean H. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *The Journal of Logic Programming*, 1(3):267 – 284, 1984.

[10] Facebook Inc. Introducing mcrouter: A memcached protocol router for scaling memcached deployments. `https://bit.ly/2wDbLml`, September 2014.

[11] D. C. Feldmeier, A. J. McAuley, J. M. Smith, D. S. Bakin, W. S. Marcus, and T. M. Raleigh. Protocol boosters. *IEEE Journal on Selected Areas in Communications*, 16(3):437–444, September 2006.

[12] Hans Giesen, Lei Shi, John Sonchack, Anirudh Chelluri, Nishanth Prabhu, Nik Sultana, Latha Kant, Anthony J McAuley, Alexander Poylisher, André DeHon, and Boon Thau Loo. In-network Computing to the Rescue of Faulty Links. In *Proceedings of the 2018 Morning Workshop on In-Network Computing*, NetCompute '18, pages 1–6, New York, NY, USA, 2018. ACM.

[13] Khilan Gudka, Robert N.M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G. Neumann, and Alex Richardson. Clean Application Compartmentalization with SOAAP. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1016–1031, New York, NY, USA, 2015. ACM.

[14] Vishal Gupta, Ripal Nathuji, and Karsten Schwan. An Analysis of Power Reduction in Datacenters Using Heterogeneous Chip Multiprocessors. *SIGMETRICS Performance Evaluation Review*, 39(3):87–91, December 2011.

[15] Ilija Hadžić and Jonathan M. Smith. Balancing Performance and Flexibility with Hardware Support for Network Architectures. *ACM Transactions on Computer Systems*, 21(4):375–411, November 2003.

[16] Van Jacobson. Compressing TCP/IP Headers for Low-Speed Serial Links. RFC 1144, 1990.

[17] Vimalkumar Jeyakumar, Mohammad Alizadeh, Yilong Geng, Changhoon Kim, and David Mazières. Millions of Little Minions: Using Packets for Low Latency Network Programming and Visibility. *SIGCOMM Computer Communication Review*, 44(4):3–14, August 2014.

[18] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 121–136, New York, NY, USA, 2017. ACM.

[19] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, New York, NY, USA, 2010. ACM.

[20] Jean-Yves Le Boudec and Patrick Thiran. *Network calculus: a theory of deterministic queuing systems for the internet*, volume 2050. Springer Science & Business Media, 2001.

[21] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 137–152, New York, NY, USA, 2017. ACM.

[22] Shen Liu, Gang Tan, and Trent Jaeger. PtrSplit: Supporting General Pointers in Automatic Program Partitioning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2359–2371, New York, NY, USA, 2017. ACM.

[23] Netcope Technologies. Netcope P4. https://www.netcope.com/en/products/netcopep4, June 2017.

[24] Netronome Inc. Agilio CX SmartNICs. https://www.netronome.com/products/agilio-cx/, 2016.

[25] P4 Language Consortium. P4 Behavioral Model. https://github.com/p4lang/behavioral-model, November 2018.

[26] P4 Language Consortium. P4 reference compiler. https://github.com/p4lang/p4c, November 2018.

[27] p4lang/tutorials. p4lang/tutorials. https://github.com/p4lang/tutorials.

[28] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: A Framework for NFV Applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 121–136, New York, NY, USA, 2015. ACM.

[29] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: A Programming System for NIC-Accelerated Network Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 663–679, Carlsbad, CA, 2018. USENIX Association.

[30] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing Privilege Escalation. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 16–16, Berkeley, CA, USA, 2003. USENIX Association.

[31] A.B. Roach. A Negative Acknowledgement Mechanism for Signaling Compression. RFC 4077, 2005.

[32] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the social network's (datacenter) network. *SIGCOMM Computer Communication Review*, 45(4):123–137, August 2015.

[33] Beverly Schwartz, Alden W. Jackson, W. Timothy Strayer, Wenyi Zhou, R. Dennis Rockwell, and Craig Partridge. Smart Packets: Applying Active Networks to Network Management. *ACM Transactions on Computer Systems*, 18(1):67–88, February 2000.

[34] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Hong Liu, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. *Communications of the ACM*, 59(9):88–97, August 2016.

[35] Twitter Inc. Twemcache: Twitter Memcached. https://github.com/twitter/twemcache, June 2013.

[36] M. H. Van Emden and R. A. Kowalski. The Semantics of Predicate Logic As a Programming Language. *The Journal of the ACM*, 23(4):733–742, October 1976.

[37] Han Wang, Robert Soulé, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. P4FPGA: A Rapid Prototyping Framework for P4. In *Proceedings of the Symposium on SDN Research*, SOSR '17, pages 122–135, New York, NY, USA, 2017. ACM.

[38] Xilinx Inc. Xilinx SDNet. https://www.xilinx.com/products/design-tools/software-zone/sdnet.html, 2017.

[39] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Klaus-Tycho Förster, Arvind Krishnamurthy, and Thomas Anderson. Understanding and Mitigating Packet Corruption in Data Center Networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 362–375, New York, NY, USA, 2017. ACM.

A Individual Function Evaluation

To validate their effectiveness, we measured the performance of individual in-network elementary functions—such as FEC, Memcached caching, and header compression—in our physical testbed. These functions are instances of *protocol boosters* [11].

A.1 Function 1: Forward-Error Correction

We developed a forward-error correction (FEC) link-layer network function [12] on FPGA to mitigate against corrupting links [39]. It introduces parity packets which are used downstream to recover corrupted packets. Our implementation uses a block code, supplementing every block of k (data) packets

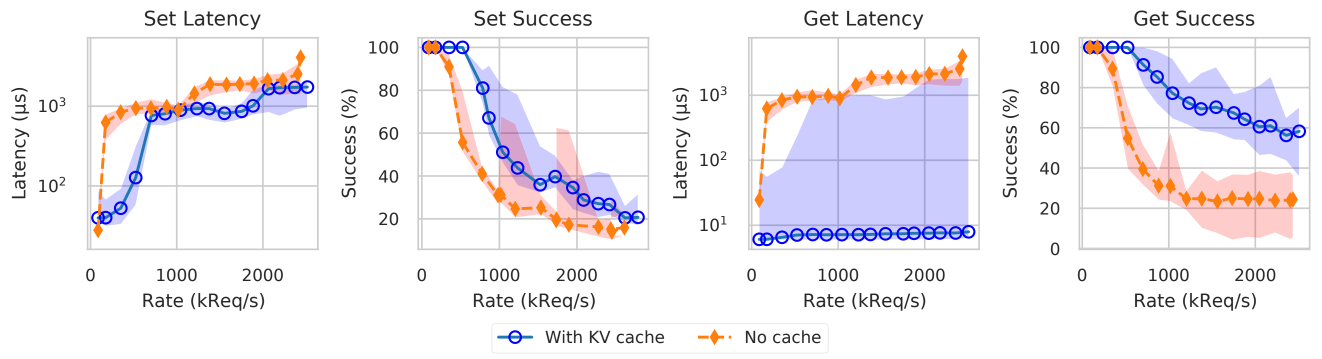


Figure 12: Performance metrics for Memcached with and without the KV-cache function. Markers show median values, while shaded regions show 5th and 95th percentiles. Success is measured as the percentage of responses received for every 500 requests.

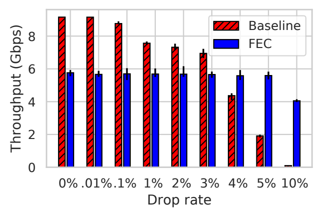


Figure 13: Effect of FEC on TCP throughput at varying rates of link loss

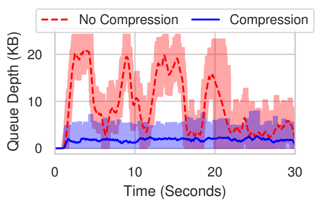


Figure 14: Effect of header compression on switch queue occupation

with h parity packets. The parity allows the in-network function to reconstruct up to h packets per block. The parity packets are computed by an implementation of the Reed-Solomon erasure code.

Forward-Error Correction reduces the need to retransmit packets in TCP flows and helps TCP sustain larger congestion windows across corrupting links. This is demonstrated by Figure 13 for an experiment performed with `iperf3` with 10 simultaneous TCP connections. Traffic was encoded with $k = 5$ and $h = 1$. Each bar represents the median throughput of 10 experiments, each of which lasted 1 min. Error bars show the minimum and maximum. For a packet drop rate of 5%, we observe that FEC increases the throughput by almost 3×. At 10% drop rate, throughput has all but vanished, but FEC manages to recover 40% of the 10-Gbps link capacity.

A.2 Function 2: Key-Value cache

We implemented an inline Memcached cache on FPGA to accelerate key-value queries. This improves the throughput, success rate, and latency of a Memcached deployment. The cache has the capacity to hold 1000 entries in its local hash table.

It was benchmarked against a standard Memcached server with default settings, including the use of four threads. Requests consisted of 8-byte keys and 512-byte values. 90% of requests sent were GET requests, while the remaining 10% were SET requests. Keys were chosen randomly from 10,000 candidates according to a Zipf distribution with an exponent of -1, consistent with measurements of document access frequency on the Web [8]. This distribution of request keys resulted in a cache hit-rate of around 50%.

Figure 12 shows the differential effects of the cache on SET and GET requests. As SET requests must still reach the Memcached server before they are acknowledged, the cache has minimal effect on SET packets, resulting only from the reduced load on the server. The inline cache keeps the median latency of GET requests below 10 µs regardless of the request rate. With the cache in place, approximately 50% less packets are lost than without the cache, consistent with its 50% hit rate.

A.3 Function 3: Header compression

We implemented an in-network function on the FPGA, Tofino and CPU for lossless compression of packet headers between neighboring switches. Our implementation is a simplified version of Van Jacobson compression [16] and many fixed-function implementations [7]. By compressing packets, bursts occupy less buffer space. Fig. 14 shows this effect for a highly-utilized (99%) network, as measured on the EdgeCore switch. Lines show the median, and the shaded region the range, over a rolling window of 100k samples.

B Single target micro-benchmarks

In addition to the end-to-end benchmarks of network functions, each individual device was also evaluated with fixed-rate packet replays using DPDK-pktgen.

Table 3 summarizes the maximum throughput and average latency that each function achieves on the CPU and FPGA targets.

Target	Function	Throughput (Mbit/s)	Latency (µs)
CPU	HC Compress	95.2 (0.038)	5800 (33.9)
CPU	HC Decompress	198.75 (1.5)	5900 (20)
FPGA	HC Compress	8350 (150)	5.24 (0.005)
FPGA	HC Decompress	8890 (17)	4.47 (0.002)
CPU	FEC Encode	35 (0.23)	300 (1400)
CPU	FEC Decode	64.47 (0.007)	90 (5.7)
FPGA	FEC Decode	7950 (27)	32.8 (1.4)
FPGA	FEC Encode	8130 (18)	4.75 (0.004)
FPGA	KV Inline cache	7730 (230)	15.9 (6.7)

Table 3: Throughput and latency values from per-function and per-target micro-benchmarks. Values shown are mean and standard deviation.

C Complexity Analysis of Dataplane Disaggregation

In this section we calculate the number of ways to split a program and allocate its splits to execute on nodes in the network subject to certain constraints. Starting with the number of ways to split a program, we abstract this as the number of contiguous subsequences $\{(1,\ldots,k_1),(k_1+1,\ldots,k_2),\ldots(\ldots,n)\}$ of a sequence $1,\ldots,n$ that represents the lines of the program. There are $(n-1)$ ways to bisect the sequence $1,\ldots,n$. Bisecting it again—to yield three subsequences—presents $(n-2)$ choices. In the general case, the number of k-ary dissections of an n-sequence (where $k<n$) is:

$$(n-1)\times(n-2)\times\ldots\times(n-k) = \left(\prod_{i=1}^{k}(n-i)\right)\in O\left(n^k\right) \quad (1)$$

Thus the number of ways to split a program grows exponentially in the number of splits sought.

Turning now to growth relative to the topology. Placement of subprograms occurs along a path between two hosts on the network. For simplicity assume that there is a single shortest path between the two, call it p. Let $A(p)$ be the set of vertices connected to p through a single edge, we call this the set of adjacent nodes. Subprograms can be placed along p or offloaded onto adjacent nodes drawn from $A(p)$. Since we are deriving an upper-bound, assume that each subprogram can be placed on any of the elements in $A(p)$ or placed on p. Let $|p|$ bet the length of path p, and $|A(p)|$ be the cardinality of set $A(p)$. The set of placement choices is $2^{|p|+|A(p)|}-1$, discounting the choice where no placement is made on any node. For a k-split program the number of placement choices is $\binom{2^{|p|+|A(p)|}-1}{k}=$

$$\prod_{i=1}^{k}\left(\frac{2^{|p|+|A(p)|}+1-i}{i}\right)\in O\left(\left(2^{|p|+|A(p)|}\right)^k\right)$$

Since in this theoretical model the choices of splitting and placement are independent then, modulo the simplifying assumptions, the space of disaggregation choices grows exponentially with the number of targets available on which to map subprograms, and doubly with the number of subprograms: $O\left(n^k\left(2^{|p|+|A(p)|}\right)^k\right)$. ■

Listing 2: Flightplan header definition

```
#define SEGMENT_DESC_SIZE 4
#define SEQ_WIDTH 32
#define STATE 8

header Flightplan_h {
  // Includes Ethernet header to simplify parsing, and handling by black-box
        external functions that aren't aware of the Flightplan header.
  bit<48> dst;
  bit<48> src;
  bit<16> type;

  bit<SEGMENT_DESC_SIZE> from_segment;
  bit<SEGMENT_DESC_SIZE> to_segment;
  bit<STATE> state;
  bit<BYTE> byte1;
  bit<BYTE> byte2;
  bit<BYTE> byte3;
  bit<BYTE> byte4;
  bit<QUAD> quad1;
  bit<QUAD> quad2;
  bit<QUAD> quad3;
  bit<SEQ_WIDTH> seqno;
}
```

D Flightplan header

Listing 2 shows the P4 definition of the header which Flightplan uses when sending packets between connected dataplanes.

E Runtime Implementations

This section elaborates on the description provided in §5 of our implemented Flightplan runtimes and their resource overheads.

E.1 Full Runtime

This section expands the description given in §5.3.

To detect network distortion such as drops, reordering, and duplication, our scheme includes a sequence number in the Flightplan header. A corresponding amount of state is kept at the sending and receiving dataplanes to determine whether the sequence of packets has been interrupted. Negative acknowledgment [31] is used for a downstream dataplane to signal loss of synchronization with the upstream dataplane. Positive acknowledgments are used by the upstream dataplane to poll liveness of the downstream dataplane.

This mechanism sets up a feedback loop between connected dataplanes. Our scheme is illustrated in Figure 15. Using this scheme we push fault detection into the dataplane, to react to faults in the network by triggering a relink or escalating a notification.

Among our runtimes, Full provides the most flexibility when splitting a program: control-flow can be stopped anywhere in the program and resumed later on a different dataplane using context that was serialized into the Flightplan header. Flightplan has no visibility into dependencies on state that is controlled from outside the dataplane, such as extern

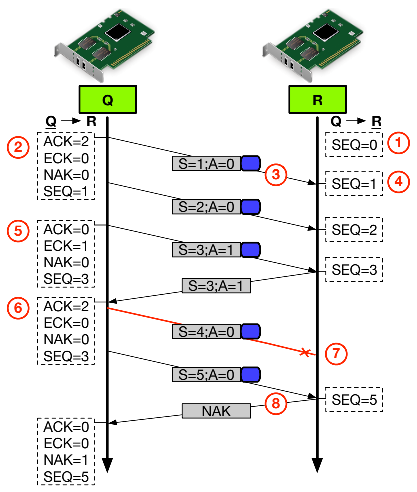


Figure 15: ① Synchronization state is initialized at downstream dataplanes. ② Synchronization state is initialized at upstream dataplanes. ③ Synchronization information is included in the Flightplan header (shown in grey) that is added to packets (shown as the blue square). ④ Sequence number is incremented. ⑤ Periodically upstream dataplane will poll downstream dataplane for activity, by raising the ACK bit in the Flightplan header, and the ECK (Expecting ACK) locally. ⑥ ECK is reset when an ACK packet is received. ⑦ Packets may be lost in either direction, leading to loss of synchronization. ⑧ Loss of synchronization is eventually detected and action is taken. Negative acknowledgment (NAK) seeks to update the upstream dataplane.

function state and tables; those need to be synchronized externally.

The runtime keeps track of the following sets of information: N (the *next* dataplanes that a dataplane might transfer to), P (*previous* such dataplanes), S_N (which fail-over alternative to use for each next-dataplane), and N_F and P_F (metadata for failure detection and handling, such as sequence numbers and whether an acknowledgement has been demanded).

On each dataplane on which part of the disaggregated program is run, `fpctl` configures dataplanes to fail-over to use other downstream dataplanes in the event of reaching a threshold of received NAKs or missed ACKs. Both thresholds are absolute values (not ratios, for instance) that are configured by `fpctl`, which can also periodically poll the state of each runtime to determine if it has seen an increase in failures. Various actions can be taken depending on such an observation, for example: i) the threshold could be raised—again, using `fpctl`—to buy time to understand the source of the loss, ii) rate limits could be applied to applications using the link, iii) the state and configuration of the downstream dataplane can be inspected or reset, iv) regardless of whether the cause has been understood or not, the ACK and NAK counts could be reset periodically if they are no longer seen to increment regularly. These actions are not directly implemented in

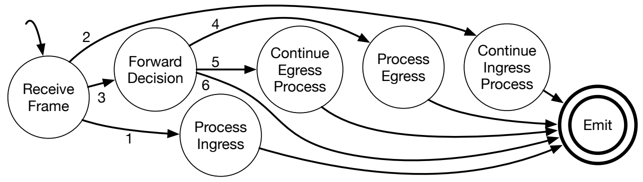


Figure 16: Core logic of the Headerless runtime.

`fpctl` but they could be automated by the network operator by using the functionality provided by `fpctl`.

If no further fail-overs are possible, then the dataplane shuts down; in turn, upstream dataplanes will eventually fail-over to use a different dataplane, if possible, when their threshold is met.

E.2 Headerless Runtime

The Headerless approach works by building a circuit between dataplanes. Compared to the Full approach, in which a program can be split almost arbitrarily, Headerless provides less flexibility.

A program is assumed to have three parts: the ingress subprogram, the forwarding subprogram, and the egress subprogram. The ingress or egress subprograms, or both, may be empty. If non-empty, they are segmented for offloading into supporting devices, which we call *helper* dataplanes.

Thus the HL runtime differentiates starkly between two types of dataplanes: the switch, on which the forwarding subprogram is kept, gets the HL_S version of this runtime, while the helpers get the much simpler HL_H part. The HL_H part of the runtime receives offloads from HL_S, performs a computation, and returns control back to HL_H. Disaggregations using HL have a single instance of HL_S and potentially several of HL_H. The ingress and egress subprograms, if non-empty, are executed on a dataplane running HL_H.

Fig. 16 shows the core logic of this circuit, which takes place on HL_S. It reflects the three parts in which the program is organized: first, one or more ingress subprogram segments (I_N) are executed, after which a forwarding step is made to determine whether to execute the egress segments (E_N). Upon arrival, if the ingress port is in I then transition 1 is taken, else if the port is in I_N then transition 2 is taken. Otherwise a forwarding decision is made, and if the egress port is in E then transition 4 is taken, or if the port is in E_N then transition 5 is taken; otherwise transition 6 is taken if the program's egress segment does not apply to the egress port.

A consequence of using the Headerless runtime is that disaggregation must be coarser—only linear segments are possible, not at DAG, and downstream live-variables must be included in the same segment that updates the variables. As a result, the complexity of segmenting is strictly lower than the general case, as shown in §H. The suitability of using Headerless runtime with a given segmentation is detected statically using our P4 compiler extension.

F Runtime Overhead

Table 4 accounts for the overheads of each approach. The overheads are described as functions over runtime-specific parameters described in §E.

The constants in each function were derived by counting the state used in each runtime, such as the width of registers, and the width of entries from a table's P4 specification. Further, we counted the number of table entries resulting from typical configuration of each runtime.

For example, the function $N \cdot 13$ for HL(IPv4)'s total size of entries (in bits) is derived from the number of entries N—the number of next dataplanes that a segment might forward to—which is known at compile time, multiplied by 13 which consists of 9 bits for the port number and 4 bits for the segment number. The state used by register arrays is the product of the array size and the register size. For example, the term $(N_F + P_F) \cdot 228$ contributing to Full's register overhead consists of register arrays whose register sizes total 228 bits, one of which is of size N_F and the other P_F. The parameters are described in the previous section and in the table's caption.

These overheads are not program-specific, but rather they are additional to the program's overheads. Also, these overheads are per-dataplane, not per-program. So if a dataplane program is disaggregated into more subprograms then it will incur more cumulative overhead, but the overhead's factor will depend on the runtime involved.

So for example if an L-line P4 program used R register bits and T tables, whose entries occupied $|T|$ bits, were disaggregated into N subprograms, then if it were to use the Headerless runtime, each subprogram S would be of size $L_S + 70$, where $\frac{L}{N} \approx L_S \leq L$ is the subprogram's line count. HL$_S$ would have $R+4$ register bits, $T+5$ tables. Assuming that $I = E = 1$, and that half of the N subprograms relates to the ingress segment (and therefore the other half relates to the egress segment), we have that $I_N = E_N = \frac{N}{2}$, and therefore the memory needed for table entries is

$$\begin{aligned} & |T| + \left(22 + \tfrac{N \cdot 22}{2} + 22 + \tfrac{N \cdot 22}{2} + 10\right) \\ = \quad & |T| + N \cdot 22 + 54 \end{aligned}$$

bits. The overhead of HL$_H$ can be calculated similarly.

G Complexity Analysis of Headerless Disaggregation

Below we define the complexity function C of headerless disaggregation, in terms of the number of possible choices that can be made during this process. The asymptotic complexity of disaggregating a program to use the Headerless runtime is $\max(C(S_I), C(S_E))$, where S_I is the sequence of lines in the ingress segment, and S_E the sequence of lines in the egress segment.

Let S be a sequence of lines of code, $n = |S|$ be the number of lines in S, and $1 \leq B \leq n$ the arithmetical average, in lines of code, of blocks in S. Top-level lines of code that do not form part of a block are counted as blocks of length 1. The number of blocks in S is $1 \leq \frac{n}{B} \leq n$.

At the limit, each block will be mapped to a different segment. But there might be constraints that force us to merge segments or objectives that are benefitted by such merges. Let $M \in \mathbb{N}, 0 \leq M \leq \left(\frac{n}{B} - 1\right)$ be the *merge opportunity*: i.e., the number of merges of (adjacent) segments we can carry out to the maximally segmented program. Increasing M has the effect of making segmentation coarser by merging more blocks together. Blocks are merged to eliminate data-flow dependencies between blocks since the Headerless runtime does not allow context serialisation and transfer between dataplanes. Blocks are also merged to make better use of physical resources; in our setup this merge decision is made by the planner.

The value of C consists of the sum of merge choices for each value of M, which by the binomial theorem simplifies to an exponential function:

$$C(S) = \sum_{M=0}^{\frac{n}{B}-1} \binom{\frac{n}{B}-1}{M} = 2^{\frac{n}{B}-1} \in O\left(2^{\frac{n}{B}}\right) \qquad (2)$$

■

H Comparative complexity

In this section we show that for programs having more than 2 lines, our Headerless disaggregation (§G) presents fewer choices than general disaggregation (§C).

We start by clarifying the number of choices presented by general disaggregation. From Equation 1 in §C, the complexity of general disaggregation is $O\left(n^k\right)$. Recall that $k < n$. Expanding for all possible values of k, which represents the number of splits we may choose, we obtain:

$$O\left(n^1 + \cdots + n^{n-1}\right) \subseteq O\left(n^{n-1}\right)$$

From Equation 2, the complexity of Headerless disaggregation is $O\left(2^{\frac{n}{B}}\right)$.

Assuming $n > 2$, our goal is to show $O\left(2^{\frac{n}{B}}\right) \subset O\left(n^{n-1}\right)$.

Recall that $1 \leq B \leq n$, and note therefore that $O\left(2^{\frac{n}{B}}\right) \subseteq O(2^n)$.

Without changing our assumption regarding n, we use the transitivity of $\subseteq$ to reformulate our goal as $O(2^n) \subset O\left(n^{n-1}\right)$. Take 2^n and n^{n-1} as the bounding functions of the two classes respectively. We show that $2^n < n^{n-1}$ first through application

[6] In addition to the resources quantified in Table 4, the Full approach also uses mirroring sessions to provide feedback. One session is used for each upstream dataplane to which it needs to provide feedback. These are configured automatically by `fpctl`.

[7] The header can be enlarged to afford more scratch space if further headroom is made available from the interfaces' MTU.

[8] We repurpose the Fragment Offset field instead of adding a header.

Runtime	#LOC	#Tables	Entries(#bits)	Registers(#bits)	Header(#bits)
Full[6]	400	6	$N \cdot S_N \cdot 17 + 2N \cdot 36 + 2N_F \cdot 36 + 2P_F \cdot 40$	$71 + (N_F + P_F) \cdot 228 + N \cdot 10$	288[7]
HL(IPv4)	75	1	$N \cdot 13$	11	0[8]
HL $\begin{cases}\text{HL}_S\end{cases}$	70	5	$I \cdot 22 + I_N \cdot 22 + E \cdot 22 + E_N \cdot 22 + 10$	4	0
HL $\begin{cases}\text{HL}_H\end{cases}$	70	1	22	4	0

Table 4: Dataplane and network overheads for each runtime. In HL we differentiate between HL$_S$ and HL$_H$: the switch dataplane and the helper dataplanes. These overheads were calculated by counting the resources used by each runtime's P4 implementation, and they are inherited by every dataplane involved in running a disaggregated program that uses that particular runtime. N is the number of *next* dataplanes that a dataplane might transfer to, P is the number of *previous* such dataplanes. $S_N \geq 1$ is the number of fail-over states: if $S_N = 1$ then there is no fail-over. $N_F \leq N$ and $P_F \leq P$ is the number of next and previous dataplanes for which a *feedback loop* is configured. I and E are the number of ingress and egress ports for which a program is running, and $I_N \geq 0$ and $E_N \geq 0$ are the number of intermediate hand-overs during the ingress and egress stages of the program.

of identities:

$$\begin{aligned}
& 2^n < n^{n-1} \\
\Leftrightarrow\ & \log_2(2^n) < \log_2(n^{n-1}) \\
\Leftrightarrow\ & n\log_2 2 < (n-1)\log_2 n \\
\Leftrightarrow\ & n < (n-1)\log_2 n \\
\Leftrightarrow\ & 0 < (n-1)\log_2 n - n
\end{aligned}$$

Then by induction on n, using $n = 3$ for the base case, verifying that $0 < 2 \cdot 1.58 - 3$, and then assuming the induction hypothesis $0 < (n-1)\log_2 n - n$ to show $0 < n\log_2(n+1) - (n+1)$. We start with application of identities:

$$\begin{aligned}
& 0 < n\log_2(n+1) - (n+1) \\
\Leftrightarrow\ & 0 < n(\log_2 n + \log_2\left(\tfrac{n+1}{n}\right)) - (n+1) \\
\Leftrightarrow\ & 0 < n\log_2 n + n\log_2\left(\tfrac{n+1}{n}\right) - n - 1 \\
\Leftrightarrow\ & 0 < \log_2 n + (n-1)\log_2 n + n\log_2\left(\tfrac{n+1}{n}\right) - n - 1 \\
\Leftrightarrow\ & 1 < ((n-1)\log_2 n - n) + \log_2 n + n\log_2\left(\tfrac{n+1}{n}\right)
\end{aligned}$$

Using the induction hypothesis we conclude that $((n-1)\log_2 n - n) > 0$, and using the assumption that $n > 2$ it follows that $\log_2 n > 1$ from the definition of $\log_2$, and $\log_2\left(\frac{n+1}{n}\right) > 0$ from the definition of $\log_2$ and since $\frac{n+1}{n} > 1$. ■

I Modelling Language

We use a simple formal language to describe relations between segments, computational resources and network resources. The relations rely on symbols used to represent those entities, such as "Compress", "PacketSize", and "FPGA". We require the user to declare these symbols by specifying a *signature*. A Flightplan signature consists of two finite sets: propositions Prop and variables V.

Signatures can be reused, at least partly, by different programs on the same network, or by the same program being deployed to different networks. Our P4 compiler extension generates an initial signature together with the abstract programs and the initial resource rules. The user can then extend and maintain this as needed.

$$\frac{\text{FPGA} \quad \text{Rate} < 9.5 \times 10^9 \quad \text{PacketSize} > 100}{\text{header\_compress}} \left[\begin{array}{l} \text{Lat.} \mapsto \text{Lat.} + 6.44 \times 10^{-6} \\ \text{Rate} \mapsto \text{Rate} \times \frac{9.15}{9.3} \\ \langle \text{LUTs} \rangle \mapsto \text{LUTs} + 24.4\% \\ \langle \text{BRAMs} \rangle \mapsto \text{BRAMs} + 54.4\% \\ \langle \text{FF} \rangle \mapsto \text{FF} + 15.8\% \\ \text{once Power} \mapsto \text{Power} + 30\,\text{W} \\ \text{once Cost} \mapsto \text{Cost} + 2 \end{array}\right]$$

Rule 4: Running Header Compression on an FPGA. Angled brackets indicate that the variables they reference depend on the device—e.g., instead of updating the flip-flop (FF) count for all FPGAs, we increment those of the FPGA to which HC is mapped using this rule.

$$\frac{\text{CPU}}{\text{egress\_compression}} \,[\text{Id}]$$

Rule 5: Flightplan's static analysis identifies egress_compression as a resource, and specifically as a table. We add a rule that allows this resource to be used on a CPU-based target.

Resource-related syntax is mapped to distinct propositions from Prop. We encountered the proposition `header_compress` in Rule 1. That rule had another proposition, "CPU", that is external to the program but is used to qualify the semantic to a specific hardware.

The variables in V are used to construct a second syntactic category in our specification language called Bound. For each $v \in V$ and $r \in \mathbb{R}$, the following are valid bounds: $v \,\text{op}\, r$ for op $\in \{=, <, >, \leq, \geq\}$. Rules 1 and 4 show examples of bounds usage from our formalization.

Propositions and bounds are also used to express constraints of network hardware. The planner (§6) uses these rules to explore implications of using those devices in plans. Fig. 17 shows a subset of the rules used for the evaluation described in §7.2.2, which model the network described in §7.2.1.

$$\frac{\text{CPU}}{\text{drop}}\ [\text{Id}]$$

Rule 6: Flightplan's static analysis identifies drop as an external function. We add a rule that allows this to be called on a CPU-based target.

$$\begin{aligned}
&\text{edgecore-2}:\\
&\quad \pi_{\text{Requires}} = \left\{\text{Rate} \leq 10^{11}\right\},\\
&\quad \pi_{\text{Provides}} = \{\text{PSwitch}\}\\
&\text{arista-1}:\\
&\quad \pi_{\text{Requires}} = \left\{\text{Rate} \leq 4 \times 10^{10}\right\},\\
&\quad \pi_{\text{Provides}} = \{\text{Switch}\}\\
&\text{ZCU102-5}:\\
&\quad \pi_{\text{Requires}} = \left\{\text{Rate} \leq 10^{10}\right\},\\
&\quad \pi_{\text{Provides}} = \{\text{FPGA}\}\\
&\text{Xeon2450-1}:\\
&\quad \pi_{\text{Requires}} = \left\{\text{Rate} \leq 10^{10}\right\},\\
&\quad \pi_{\text{Provides}} = \{\text{CPU}\}
\end{aligned}$$

Figure 17: A subset of the network-description rules used for the evaluation described in §7.2.2. Unlike other rules we have seen, these rules describe the constraints on using each device and the features enabled by each device. The planner uses this information when exploring the space of solutions.

J Generating rules from empirical profiles

Table 3 provided a summary of our profiling experiments. This section describes how the profiling experiments were done, and how their results were used to generate the *profile-related rules* such as Rules 1, 3 and 4. Other types of rules are derived from the program or from the network's description and are generated differently as described in §4.

J.1 Methodology

For each external function involved in a program, we measure its peformance characteristics on different classes of devices and use this information to estimate the performance of program segments in which those functions occur.

The performance profile is made by installing the function on a member of each class on which it can run and running a workload to sample the function's performance. This is used to create an entry in a table consisting of the following columns:

$$(\text{Function}, \text{PacketSize}, \text{Target}, \text{Time}_{\text{Arrive}}, \text{Time}_{\text{Leave}},\\ \text{Rate}_{\text{Arrive}}, \text{Rate}_{\text{Leave}})$$

The first three columns consist of the function's name, the packet size used in the workload, the name of the hardware target class. The next two columns consist of the timestamps of when a frame is received by the function, and the processed frame is sent by the device.[9] Similarly, the last two columns consist of the arrival rate to the function and the sending rate. This captures the effect of the function on the link's capacity: for example FEC uses more of the downstream capacity compared to upstream, while header compression uses less.

Further, we also make two measurements that are less function- and workload-dependent: Power consumed by the target when executing the function on that workload, and the Cost of the hardware target. Instead of literal equipment costs we used relative quantities: $\{\text{FPGA} = 1, \text{Switch} = 2, \text{CPU} = 5, \text{PSwitch} = 10\}$.

Limitations on precision. This approach will not accurately account for differences between devices in the same class—e.g., different FPGA devices—or different configurations of the same device—e.g., kernel bypass on CPU targets—but it will give us good-enough characterization for our purposes, and a starting point for more accurate characterizations. Like all profiling, it is also sensitive to the workload used. Despite this, our approach is amenable to refinement in the following way: devices classes can be refined into subclasses, different kinds of workloads can be distinguished by adding a variable to each rule, and additional variables can be sampled by measurements and added to the generated rules. For example, the $\langle \cdots \rangle$ in Rule 4 involves an extension we made to more accurately characterize the usage of FPGA resources. Since the utilization of FPGA resources is a simple additive approximation we do not constrain the planner by their values—for instance to ensure that an FPGA's LUTs are not exceeded—but instead we do a post-hoc check.

Limitations on scale. To scale with the size of the network, we avoid creating a profile for every target in the network. Further, to scale with program and disaggregation diversity, we avoid creating a profile for every segment. Instead we focus on external functions, and measure their performance characteristics on different classes of devices as described above. In future work some of this characterization work can be automated further, to improve both its scalability and its precision.

J.2 Compiling the table

For each Function and Target we run a series of experiments in which we gradually increase PacketSize and $\text{Rate}_{\text{Arrive}}$. As long as we do not notice any drops—which would indicate that the function is not coping with the arrival rate for that packet size—we create a table entry for the information described above.

We then process the raw table to create a second table consisting of the following columns, some of which overlap with the first table we described:

$$(\text{Function}, \text{PacketSize}, \text{Target}, \Delta\text{Latency}, \text{Rate}_{\text{Arrive}}, \Delta\text{Rate},\\ \text{Power}, \text{Cost})$$

[9] Both timestamps were generated by the EdgeCore Wedge switch and were mirrored to a collection server on the side.

$$\frac{\boxed{\text{Function}} \quad \text{Rate} < \left(\boxed{\text{Rate}_{\text{Arrive}}} + T_{\text{R}}\right) \quad \text{PacketSize} > \left(\boxed{\text{PacketSize}} - T_{\text{PS}}\right)}{\boxed{\text{Target}}} \left[\begin{array}{l} \text{Latency} \mapsto \text{Latency} + \boxed{\Delta\text{Latency}} \\ \text{Rate} \mapsto \text{Rate} \times \boxed{\Delta\text{Rate}} \\ \text{once Power} \mapsto \text{Power} + \boxed{\text{Power}} \\ \text{once Cost} \mapsto \text{Cost} + \boxed{\text{Cost}} \end{array}\right]$$

Rule 7: Template for performance-related rules. T_{R} and T_{PS} are tolerance offsets for rate and packet-size respectively. We set these to small values to retain fidelity while using the inequality operators.

where

$$\begin{aligned} \Delta\text{Latency} &= \text{Time}_{\text{Leave}} - \text{Time}_{\text{Arrive}} \\ \Delta\text{Rate} &= \frac{\text{Rate}_{\text{Leave}}}{\text{Rate}_{\text{Arrive}}} \end{aligned}$$

All the records from the first table are automatically processed to create the second table. This is an example record from which we will show how to generate Rule 1:

$$(\text{header\_compress}, 1000\,\text{B}, \text{CPU}, 7.4 \times 10^{-3}\text{s}, 2 \times 10^{8}\text{Gbps}, \frac{189.9}{194.75}, 150\,\text{W}, 5)$$

J.3 Converting table into rules

Finally, the table of measurements described above is converted into rules. This is done automatically as follows: for each row in the second table we instantiate the template shown in Rule 7 by replacing the black-background fields in the rule with the corresponding fields in the row. The meaning of these rules is explained in the caption of Rule 1 in §3, and in §I.

That is how we generated the performance-related rules used by our tools, including the following rules shown in this paper: Rules 1, 3 and 4.

K Program Annotation & Splitting

After finding a mapping from segments to dataplanes as part of a plan, contiguous segments that are mapped to the same dataplane are grouped into a single segment. This is done by deleting intermediate `flyto()` statements. At this point the program is said to be annotated for splitting.

Next we need to generate a physical, well-formed and self-contained P4 file for each segment. The program splitting phase performs three tasks: 1) extract the P4 code from each segment, forming subprograms; 2) analyze the subprograms to gather runtime-related context, such as variables whose values must be included in the hand-over between dataplanes; 3) inject runtime-dependent code for handing-over between subprograms.

These tasks involve simple but laborious analysis and modification of data structures in the compiler, which we outline in this section.

Subprogram generation Our implementation reuses the P4 compiler's data structures as much as possible, adding a thin layer to facilitate the analysis and transformation needed for splitting. We follow these steps: **1)** Analyze the program's abstract-syntax tree (AST) to identify all occurrences of calls to `flyto`. **2)** Generate a *virtual AST* (vAST) for each `flyto`. A vAST is an overlay of the AST for each split. Thus we obtain a single data structure using which we can simultaneously reason about the original program and all derived programs. **3)** Generate the *segment-level topology*: a graph where nodes consist of segments and where an edge denotes a code-path leading to the hand-over between adjacent segments.

Context gathering The synthesis of code between segments to pass data across dataplanes involves computing the transitive closure of required state, to ensure that downstream dataplanes shall have all the information they need to compute on. Writing such interfacing code by hand is tedious and error-prone. We continue the steps taken in the compiler extension: **4)** Traverse each vAST to compute its set of free variables (variables that are used, but not defined, in that vAST); **5)** Traverse the segment-level topology to compute the *extended scope of variables*: that is, variables defined in one vAST will need to be propagated to downstream vASTs in which they are used.

Code injection, Flightplan header This task uses the information gathered during the previous stage to ensure that the hand-over will happen correctly between all connected subprograms: **6)** If supported by the runtime, then generate stubs to propagate two kinds of information across logical dataplanes: a) the values of extended-scope variables and b) additional Flightplan metadata for instrumented programs or fault detection (§5). This is added to a special Flightplan header that encapsulates the original packet, piggy-backing logistical metadata.[10] **7)** Embed the vAST in the original program, replacing the main control block. The downstream compiler will purge unnecessary declarations and definitions. For each vAST, we update the deparsing stage to use the stub for the downstream logical dataplane. We also update the parser block to opportunistically parse the Flightplan header, and, if successful, then the control block branches to process the continuation of Flightplan code.

L Disaggregation microbenchmarks

L.1 Latency Microbenchmarks

In this section, we measure the amount of latency added by splitting a dataplane and benchmark the responsiveness of Flightplan's failover mechanisms.

End-to-end latency. We benchmark application layer end-to-end latency by measuring round-trip time (RTT) between two servers connected to switch `S1` in the topology described in §7.2.1. A concurrent ping and 10 `iperf3` streams went from the `iperf3` client to the Memcached client, with the forwarding table initially stored on `S1` and offloaded to `S2`. Fig. 18 shows the change in ping RTT distribution. Latency

[10] The Flightplan header is described in Appendix D.

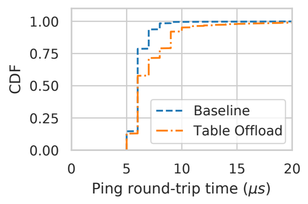


Figure 18: Table-offload overhead measured by pings across end-hosts.

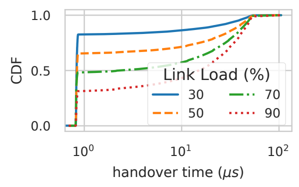


Figure 19: Effect of network load on handover latency.

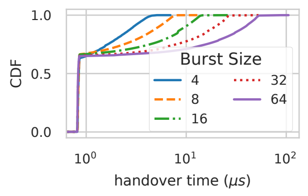


Figure 20: Effect of burstiness on handover latency.

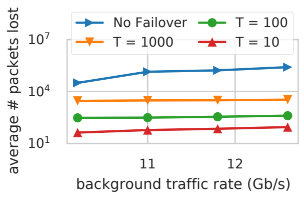


Figure 21: Thresholding (T) of packet loss.

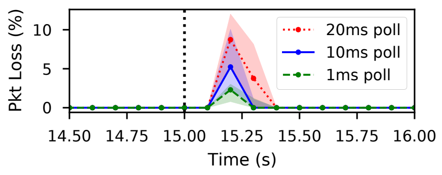


Figure 22: Packet loss during failover, as measured by `iperf3`. At time 15 s, the FEC encoder fails and is automatically replaced by another FPGA. Polling for detection of the failed link at tighter intervals results in less loss during the failover procedure.

was higher because packets had to traverse two additional queues in each direction. These queues were congested due to `iperf3`, which adds latency as discussed in Sec. 7.2.3. In this experiment we measured an 18.7% increase in average RTT and 22.3% increase in maximum RTT in our setup, but this increase will be smaller when the communicating servers are separated by more hops, for example if they are not both connected to `S1`.

Handover latency. To understand overhead at a finer scale, we measure *handover time*, the time between the invocation of the function containing the offloaded table and the beginning of its execution. Fig. 19 shows the distribution of handover time as link utilization varies. At a utilization level of 30%, reflecting high load in a data center scenario [4], handover time was under 1 μs for over 80% of the traffic. At higher rates, handover time increases due to added congestion in the queues between `S1` and `S2`. As Fig. 20 shows, larger packet bursts also have an effect, but only stretch the tail of the handover time distribution. In a 50% load scenario, the handover time remained under 1μs for over 60% of packets.

L.2 Failover Microbenchmarks

We evaluate two kinds of fail-over mechanisms. The first involves controller-based failure-detection and effecting of fail-over. The second involves in-dataplane failure-handling. A deployment can involve both simultaneously, with the in-dataplane mechanism reacting more quickly in most cases, and the controller-based approach handling cases where the in-dataplane mechanism is incapacitated because of device failure.

Controller-directed failover. In the case of total failure of a link or device, the loss of connection can be directly detected by the controller, allowing it to automatically initiate the failover procedure without further intervention.

Fig. 22 shows the packet loss rate of a UDP `iperf3` session running at 1 Gbit/s through a dataplane employing FEC during the failure of the FEC encoder FPGA. At time 15 s, the link to the FPGA is disabled. The controller, which polls for the presence of the link at regular intervals, redirects traffic to a failover FPGA once the down link is detected.

With a polling interval of 1 ms, at most 3% of packets are lost in the 100 ms interval immediately following the disabling of the link.

Dataplane NAK Failover. We benchmark the NAK mechanism described in §5.3 with a simple program on `S1` that offloads a no-op function `F`. There are two instances of `F` (`F.1` and `F.2`) that both run on switch `S2`, but service different links connected to `S1`. We measure the number of packets lost in a scenario where failover from `F.1` to `F.2` occurs due to congestion on the link to `F.1`.

Fig. 21 shows how the number of packets lost varies with the rate of congestion-inducing background traffic. The Flightplan failover mechanism bounds packet loss to a low (< 10) integer factor of the NAK threshold (T). After the Tth NAK is received, only in-flight packets already queued to `F.1` are lost because every subsequent packet is routed to the failover instance. Without the NAK mechanism, the number of packets lost is unbounded.

MilliSort and MilliQuery: Large-Scale Data-Intensive Computing in Milliseconds

Yilong Li* Stanford University
Seo Jin Park* MIT CSAIL
John Ousterhout Stanford University

Abstract

Today's datacenter applications couple scale and time: applications that harness large numbers of servers also execute for long periods of time (seconds or more). This paper explores the possibility of *flash bursts*: applications that use a large number of servers but for very short time intervals (as little as one millisecond). In order to learn more about the feasibility of flash bursts, we developed two new benchmarks, MilliSort and MilliQuery. MilliSort is a sorting application and MilliQuery implements three SQL queries. The goal for both applications was to process as many records as possible in one millisecond, given unlimited resources in a datacenter. The short time scale required a new distributed sorting algorithm for MilliSort that uses a hierarchical form of partitioning. Both applications depended on fast group communication primitives such as shuffle and all-gather. Our implementation of MilliSort can sort 0.84 million items in one millisecond using 120 servers on an HPC cluster; MilliQuery can process .03–48 million items in one millisecond using 60-280 servers, depending on the query. The number of items that each application can process grows quadratically with the time budget. The primary obstacle to scalability is per-message costs, which appear in the form of inefficient shuffles and coordination overhead.

1 Introduction

One of the benefits of datacenter computing is the ability to run large-scale applications that harness hundreds or thousands of machines working together on a common task. Using frameworks such as MapReduce [11] and Spark [67], developers can easily create applications in a variety of areas such as large-scale data analytics.

Until recently, large-scale applications have executed for relatively long periods of time: seconds or minutes. This was necessary in order to amortize the high cost of allocating and coordinating a collection of servers. Similarly, frameworks such as MapReduce and Spark have traditionally operated on very large blocks of data, in order to amortize high network latencies. As a result, they cannot be applied to real-time tasks. Instead, real-time queries must return precomputed results, such as those produced in overnight batch runs. This means that the queries must be carefully planned in advance; ad-hoc queries cannot easily be supported.

Streaming frameworks such as Flink [15] or Spark Streaming [66] operate on incoming data in real time, but they do this by incorporating new data into queries or transformations that are planned long in advance. To support real-time queries, the scale of computation triggered by each event is quite limited.

In recent years, new serverless platforms such as AWS Lambda [5], Azure Cloud Functions [45], and Google Cloud Functions [20] have made it possible to run short-lived tasks (as small as a few hundred milliseconds) in datacenters. However, the unit of execution in these environments is an individual function call. It is difficult to harness multiple machines in a single serverless computation; for example, direct communication between lambdas is not officially supported, and existing workarounds [17, 3] have high latency and low bandwidth.

This paper explores the possibility of extending serverless computing in two ways: first, by further reducing the timescale; and second, by reintroducing scale, so that large numbers of servers can work together. We use the term *flash burst* to describe a computation that has a short lifetime yet harnesses large numbers of servers. Flash bursts offer the potential of analyzing large amounts of data in real time. This could enable the creation of new applications that execute customized queries on large datasets in real time, without the need to predict queries hours or days ahead of time.

Rather than making incremental improvements on existing systems, our goal is to push the notion of flash bursts to the extreme, in order to understand the limits of this style of computation. In particular, we set out to answer the following questions:

- What is the smallest possible timescale at which meaningful flash bursts can operate?
- What is the largest number of servers that can be harnessed at such a timescale?
- What aspects of current systems limit the duration and scale of flash bursts?

We hypothesized that timescales as small as 1 ms might be possible, so we set that as our initial goal.

Making flash bursts practical will require advances in many different areas. This paper focuses on one aspect of the problem: whether the core algorithms likely to be used in flash bursts can operate efficiently at very small timescales. We do not address issues such as the time required to load applications and data, or how to achieve high resource utilization in the face of short-lived computations; we leave these for future work.

We implemented two focused applications that capture patterns of computation and communication we expect to be common in flash bursts. The first application is MilliSort: given unlimited resources in a datacenter, what is

*Co-first authors, listed alphabetically.

- Sort 40,000 10-byte keys using 8 cores [7].
- Copy 5 Mbytes of data from memory to memory sequentially.
- Send or receive 5 Mbytes of data with a 40 Gbps NIC.
- Invoke 300 back-to-back remote procedure calls on one core, using kernel bypass [50, 52, 32].
- Send or receive 2–5k small messages on one core [50, 32, 48].
- Take 10,000 back-to-back L3 cache misses on one core.

Figure 1: Examples of tasks that can be completed in one millisecond on modern hardware.

the largest number of small records that can be sorted in one millisecond? The second application is MilliQuery, which consists of three representative SQL queries from the tutorials for Google BigQuery. The queries range from a simple scan-filter-aggregate query to a distributed join requiring multiple shuffles. As with MilliSort, our goal was to understand how much data can be analyzed, and how many servers can be harnessed, in timescales around 1 ms.

The process of developing and measuring these applications has yielded interesting results in three categories:

- Measurements: MilliSort and MilliQuery demonstrate that large-scale data analytics can operate efficiently even at timescales of 1–10 ms. MilliSort can sort 0.84 million small records in one millisecond using 120 servers running on 30 machines. The MilliQuery benchmarks process .03–48 million records in one millisecond using 60–280 servers, depending on the query.
- Observations: the development of MilliSort and MilliQuery yielded several interesting insights about flash bursts, which are summarized in Section 7. For example, we found that the amount of data that a flash burst can handle grows quadratically with the time budget (both the amount of data per server and the number of servers grow at least linearly with the time budget). Some of the most important and common limits to scalability are shuffle cost and coordination overhead (both can be attributed to per-message overheads).
- Algorithms: while implementing MilliSort we developed a new low-latency algorithm for partitioning the keys, which uses a hierarchical series of distributed sorts. We also developed a novel splitter selection algorithm that improves the balance among data partitions. Overall, MilliSort runs with efficiency comparable to other systems that operate at much larger timescales.

2 Background

One millisecond is not very long. Figure 1 lists a few things that can be done in one millisecond on today's machines; these create fundamental limitations for flash bursts.

2.1 Limited data per server

One of the most important limitations evident from Figure 1 is that each server can only manipulate a small amount of data (on the order of a few Mbytes). For example, a single server core can only access a few megabytes of data in one millisecond, and network bandwidth allows only a few megabytes to be transmitted in one millisecond.

Given the large number of servers and small data per server, data must stay evenly distributed throughout a flash burst. If even a small fraction of data accumulates on a single server, the network link into that server will become a bottleneck.

2.2 Coordination cost

Given that each server can only access a small amount of data, the overall scale of a flash burst will be limited by the number of servers that can be harnessed. But, the small time scale makes it difficult to coordinate very many servers; at some scale one millisecond isn't even enough time to notify all the servers to start working. Thus, coordination overheads play a fundamental role in flash bursts, since they limit the scale. "Coordination" includes such activities as engaging all of the servers, determining work assignments for each server, and sequencing the phases of the algorithm. Existing large-scale applications such as Spark store much larger amounts of data per server and also run for longer time periods; this combination makes coordination overheads less important.

2.3 Multiple communication costs

For many existing large-scale applications, the only communication cost that matters is network bandwidth. Systems such as MapReduce [11] and Spark [67] are explicitly designed to exploit bandwidth and hide communication latency. However, for flash bursts three different costs may become important. In addition to *bandwidth*, which matters when sending large blocks of data, and *latency*, which matters when sending small chunks of data, a third cost plays an important role in flash bursts: *per-message overhead* (the CPU time required to send and receive short messages). Per-message overhead comes into play when a server has a collection of small requests that can be sent to other servers concurrently; it limits how quickly a series of messages can be issued. Per-message overheads are particularly important in flash bursts because they dominate the cost of group communication primitives (discussed below), which in turn dominate the cost of coordination.

2.4 Group communication

If a collection of servers is to cooperate closely, the servers will probably need to exchange data frequently and in small chunks. However, in a flash burst, where there are hundreds or thousands of servers operating on a very small time scale, it is not practical for each server to communicate directly with all of the other servers. For example, if a server broadcasts data to 1000 other servers by sending a separate small message to each of them, the broadcast will consume a substantial fraction of a millisecond, due to per-message overheads.

Thus, *group communication* plays an important role in flash bursts. In group communication, many or all of the servers in a cluster transmit data concurrently to carry out a cluster-wide goal. The HPC community has identified and implemented a variety of useful group communication mechanisms [60], of which four play a role in MilliSort and MilliQuery:

Broadcast: data that is initially present on a single server must be distributed to every server in the group.

Gather: the reverse of broadcast. A single server must collect distinct data from each of the servers in the group.

All-gather: initially each server in the group stores a distinct data item; the all-gather operation must arrange for every server in the group to receive a copy of all the items.

Shuffle: each server initially stores a separate data item for every other server in the group; the shuffle must transmit all the items to their intended targets.

Group communication provides two benefits. First, it harnesses multiple servers operating concurrently to speed up the communication; for example, several servers can be used to complete a broadcast more quickly by distributing messages via a tree structure. Second, it can sometimes replace many small messages with a few larger messages; this reduces the impact of per-message overheads. For example, a hypercube implementation [62] of all-gather requires only $MlogM$ messages for M servers, vs. M^2 messages if each server communicates independently with every other server.

3 Applications

One of the challenges in exploring flash bursts is that there are no flash burst applications available today (unsurprising, given that there is no infrastructure capable of supporting them). Fortunately, there appears to be a small set of patterns of computation and communication that are used commonly across a variety of large-scale applications and account for much of their performance [4, 19, 33]. These are referred to as *dwarfs* by Asanovic et al. [4] and others. For example, matrix operations, sorting, and statistics computations are dwarfs for big-data and AI workloads. Rather than guessing at full applications, we have implemented two small applications that capture several dwarfs. Although the behavior of these dwarfs will not be a perfect predictor of any real application, this approach has two advantages. First, lessons learned from the dwarfs are likely to apply to many real applications. Second, it is easier to understand the properties of the dwarfs if we study them in isolation, rather than as part of a full application with many interacting parts.

Our first application is a sorting benchmark called MilliSort. Sorting has been used to evaluate system performance for many decades, originating with a challenge proposed by Jim Gray and others in 1985 [2]. Sorting plays an important role in many distributed computations. For example, sorting can be used as a data preprocessing step to support efficient range queries, to improve data locality for graph partitioning [44], and to perform load balancing [44, 11, 23]. Sorting is also very challenging because it requires intensive and unpredictable communication (any record can potentially end up on any server). We expected this to create challenges both for the algorithm and the underlying infrastructure.

```
/* MilliQuery Q1: count article views on Wikipedia by language */
SELECT language, SUM(views)
FROM `bigquery-samples.wikipedia_benchmark.Wiki1B`
GROUP BY language

/* MilliQuery Q2: top 10 IPs by the number of edits to Wikipedia */
SELECT contributor_ip AS ip, COUNT(*) AS count
FROM `publicdata.samples.wikipedia`
GROUP BY ip ORDER BY count DESC LIMIT 10

/* MilliQuery Q3: complex data analytics on GitHub data */
WITH
 repo_authors AS (   -- Build the intermediate author table
  SELECT repo_name, author.name AS author
  FROM `bigquery-public-data.github_repos.commits`,
   UNNEST(repo_name) AS repo_name
  GROUP BY repo_name, author),
 repo_languages AS (  -- Build the intermediate language table
  SELECT lang.name AS lang, lang.bytes AS lang_bytes, repo_name
  FROM `bigquery-public-data.github_repos.languages`,
   UNNEST(language) AS lang)
SELECT lang, author, SUM(lang_bytes) AS total_bytes
FROM (repo_languages JOIN repo_authors USING repo_name)
GROUP BY lang, author ORDER BY total_bytes DESC LIMIT 100
```

Listing 1: The three SQL queries used in the MilliQuery benchmark

For MilliSort, the goal is to sort as many small records as possible in intervals around one millisecond, using any number of servers in a datacenter. Each record contains 100 bytes, consisting of a 10-byte key and a 90-byte value. Before starting the benchmark, the MilliSort application is preloaded and the unsorted records are distributed evenly among the available machines in DRAM. The data on each server is structured with all of the keys in a single block of memory and all the values in another block, in corresponding order. Upon completion, the data must be redistributed across the same servers, sorted such that one server contains the records with the smallest keys, and so on. At the end of the sort, the data on each server is structured in two blocks of memory, one containing the keys in sorted order and another containing the values in the same order as their keys. The challenge does not require that the result data be distributed evenly across the servers, but this turns out to be essential for good performance.

Our second "application" is a collection of three SQL queries from Google's BigQuery documentation[63, 38]; we refer to these queries collectively as MilliQuery. We expect that many flash burst applications will perform data analytics to provide real-time results from ad hoc queries; the goal for MilliQuery is to capture dwarfs that will be common in these applications.

We chose three queries with different levels of complexity, in order to span a range of SQL behaviors (see Listing 1).

Q1: counts the number of article views on Wikipedia by language (there are at most a few hundred different languages).

Q2: finds the top 10 IP addresses by the number of edits they made to Wikipedia articles.

	Coordination	Shuffle(s)	Dwarf(s)
MilliSort	Heavy	≥2	Sort
MilliQuery Q1	None	0	Aggregate
MilliQuery Q2	Light	1	Repartition,Aggregate
MilliQuery Q3	Light	3	Repartition,Join

Table 1: A comparison of the applications used for studying flash bursts.

Q3: for each combination of author and programming language, sum all of the bytes in that language in any repository for which the author was a contributor; returns the top 100 author-language pairs.

In each case, the goal is to process as much data as possible within 1 ms using unlimited datacenter resources, assuming that the application is pre-loaded and data is initially distributed uniformly across the nodes.

Table 1 compares these applications in terms of the complexity of coordination (how difficult it is to divide the work among the participating nodes and coordinate their behaviors), the number of shuffle steps required, and the dwarfs captured. Together, MilliSort and MilliQuery cover a significant range of interesting behaviors.

Our goal for MilliSort and MilliQuery was to push the idea of flash bursts to the extreme. We don't know what burst sizes will prove useful to applications, so we set out to characterize the range that is practical: what is the smallest time interval and what is the largest number of servers that can be harnessed efficiently? We also hoped to learn what are the technical factors that limit flash bursts. We chose a one millisecond time limit because it seemed like an extreme goal: at the outset, we were unsure whether it would be possible to do anything useful in such a short interval.

4 The MilliSort algorithm

Although distributed sorting is not new, designing a sorting algorithm to operate at a timescale of one millisecond introduces new challenges due to the high cost of coordination. This section describes MilliSort's algorithm in detail and presents a novel hierarchical approach to data partitioning that enables efficient coordination even at very small timescales.

Most distributed sorting algorithms use a partitioning approach, and MilliSort follows this tradition. First, the data is partitioned by deciding which key ranges should end up on each server; then the records are shuffled between servers to implement the chosen partitions. This approach optimizes the use of network bandwidth by transmitting each record only once, during the shuffle phase. The partitioning approach makes sense because it optimizes the use of network bandwidth, which has traditionally been the scarcest resource in distributed sorting. Alternative approaches, such as those that use multi-stage merge sorts, require data to be transmitted over the network multiple times, so they have proven slower than the partitioning approach.

More precisely, MilliSort implements a variant of distributed bucket sort, with one bucket for each server. It consists of four phases:

1. Local sort: each server sorts its initial data.
2. Partition bucket boundaries: the servers collectively determine the key ranges that will end up on each server after sorting.
3. Shuffle: each server transmits its keys and values to the appropriate targets.
4. Local rearrangement: the data arriving on each server during the shuffle phase must be rearranged into two totally sorted arrays, one for keys and one for values.

The sections below discuss each of these phases in more detail. We start with the partitioning phase, since it is the most complex phase and also the most interesting phase from an algorithmic standpoint.

4.1 Histogram sort

One of the most widely used partition-based sorting algorithms is histogram sort, which computes the final key ranges by iteratively refining an existing partition until the keys are evenly distributed on the servers. A typical workflow of histogram sort is as follows. In the beginning, a central server picks $M-1$ splitters, which divide the key space into M buckets, and broadcasts them to the other servers. Then, each server computes a local histogram of its keys in the buckets and sends it back to the central server. Finally, the central server computes a global histogram by summing up the local histograms and adjusts the splitters to reduce the imbalance. The process of histogramming and refinement is repeated until an even partition is achieved. In addition to the one mentioned above, there are other variations of histogram sort which use more splitters for histogramming or increase the number of splitters as they progress.

However, histogram sort is undesirable for MilliSort since it requires many iterations to converge, and each iteration incurs significant message delays. To avoid overloading the central server, histogram sort often uses group communication to broadcast splitters and reduce local histograms in a tree structure. As a result, each iteration incurs $2log(M)$ back-to-back message delays. In our environment, the combined cost of broadcast and gather is at least 50 µs for 100 servers. With just 10 iterations, message delays alone will take away half of our 1 ms time budget. Finally, the actual cost of partitioning will be even higher due to other overheads; the reported partitioning times of some recent histogram sort implementations easily exceed 50 ms for 512 HPC nodes [35, 21].

4.2 Sample sort partitioning

MilliSort takes a different approach to partitioning, selecting a larger number of initial keys so that it can estimate the distribution in a single iteration. MilliSort's partitioning algorithm is based on sample sort with regular sampling [58, 40]; the basic idea is to select many keys from the starting data, use these to estimate the distribution of keys, and pick partition boundaries based on the estimated distribution. Figure 2 shows the basic idea. After sorting its local data, each server samples its keys at equally-spaced intervals within the sorted

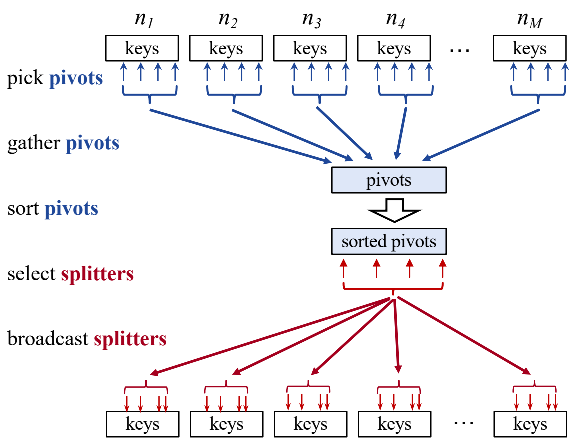


Figure 2: A basic sample sort partition mechanism; $n_1 - n_M$ are the MilliSort nodes.

records; we refer to these samples as *pivots*. The pivots of all servers are collected and sorted (more details on this below). Finally *splitter* values are chosen from the sorted pivots. If there are M machines participating in the sort, $M-1$ splitters are chosen, which divide the sorted pivots into M equal-size groups. The splitters determine how records are divided between servers during the shuffle phase: server i will eventually hold all records with keys greater than or equal to the ith splitter and less than the i+1th splitter.

Because of the sampling used by this approach, there is no guarantee that each server will end up with exactly the same number of records at the end of the sort. If there are N total records divided among M machines and each machine chooses sM pivots, then, in the worst case, a server may end up with as many as $(1+1/s)(N/M)$ records (N/M is the ideal number in a perfect partition) [58, 40]. For MilliSort, we use $s = 1$ (each machine chooses M pivots), so the final partition sizes are guaranteed to be within a factor of 2 of the ideal size. In practice the distribution of records is considerably more uniform than suggested by the worst-case formula above.

With this approach, the total number of pivots to sort is sM^2. This means that as the number of machines increases, partitioning will take more and more time, even if all of the machines share the work. Given a limited amount of time for the sort, partitioning cost will limit the number of machines that can be harnessed.

4.3 Recursive partitioning

One way to perform the partitioning is to gather all of the pivots on a single coordinator server, sort them locally on that server, then broadcast the splitters back to all of the servers. However, this approach is too inefficient for MilliSort. If 300 machines participate in the sort, there will be 90,000 pivots; as shown in Figure 1, a single server can only sort about 40,000 keys in one millisecond, so the sorting alone would take more than 2 ms. The overhead of receiving all the pivots on a single server is also problematic. Thus, millisecond-scale sorting requires partitioning to be performed in a distributed fashion.

MilliSort uses a recursive approach to partitioning: the pivots are sorted in a distributed fashion using a smaller instance of MilliSort, as shown in Figure 3. A subset of the machines, called *pivot sorters*, sort the pivots and select splitters; each of the other servers is assigned to one of the pivot sorters. To begin the sort, each pivot sorter gathers the pivots from its assignees. The pivots arriving from each source machine are already sorted, so the pivot sorter can use merge sort on the arriving data to produce a sorted list of all the pivots for which it has responsibility. Then each pivot sorter samples its pivots to choose a smaller number of *level 2 pivots*; the level 2 pivots are passed to a coordinator, which sorts them and produces a set of *level 2 splitters*. The coordinator broadcasts the level 2 splitters back to the pivot sorters, which then perform a shuffle to redistribute the pivots among the pivot sorters in sorted order.

At this point the pivots have been sorted and splitters must be chosen (i.e., we must select every sM^{th} pivot across all of the pivot sorters). We would like for each pivot sorter to independently select splitters from its pivots, but in order to do this, the pivot sorter must know its rank (i.e., how many pivots stored on other servers are smaller than the pivots that it stores). The rank is not immediately obvious because pivots are not distributed uniformly across the pivot sorters. The solution is to distribute rank information during the shuffle phase of the pivot sort. When a pivot sorter sends a group of pivots to another pivot sorter during the shuffle, it includes the *local rank* of that group (i.e., the number of pre-shuffle pivots from that pivot sorter that are smaller than those in the group). Each pivot sorter can determine its rank by summing the local ranks in all of the shuffle messages it receives. Once a pivot sorter knows its rank, it can identify the splitters that it stores.

Finally, once the splitters have been determined, they must be disseminated to all of the machines participating in the sort. One approach would be for each of the pivot sorters to broadcast its splitters to all of the M machines; however, this would have a high cost because of the large number of messages that would result. Instead, MilliSort uses a two-step approach to distribute the splitters. In the first step, an all-gather operation is used to exchange the splitters among the pivot sorters, so that each pivot sorter has all $M-1$ splitters. Then each pivot sorter broadcasts the complete set of pivots to all of the machines assigned to it.

If the number of servers is very large, the 2-level approach described above will still take too long. If that is the case, additional levels may be used in the partition. For example, in a 3-level approach the level 2 pivots will not be sorted on a single coordinator; instead, they will be collected by a smaller number of second-level pivot sorters, which will then select a set of level 3 pivots. The level 3 pivots will be collected and sorted on a single coordinator, resulting in level 3 splitters, which are used to shuffle the level 2 pivots. The approach can be extended to an arbitrary number of levels, though our experiments suggest that 2 or 3 levels is appropriate for MilliSort.

We use r to refer to the reduction factor at each level of recursive sorting. For M total machines, there will be M/r

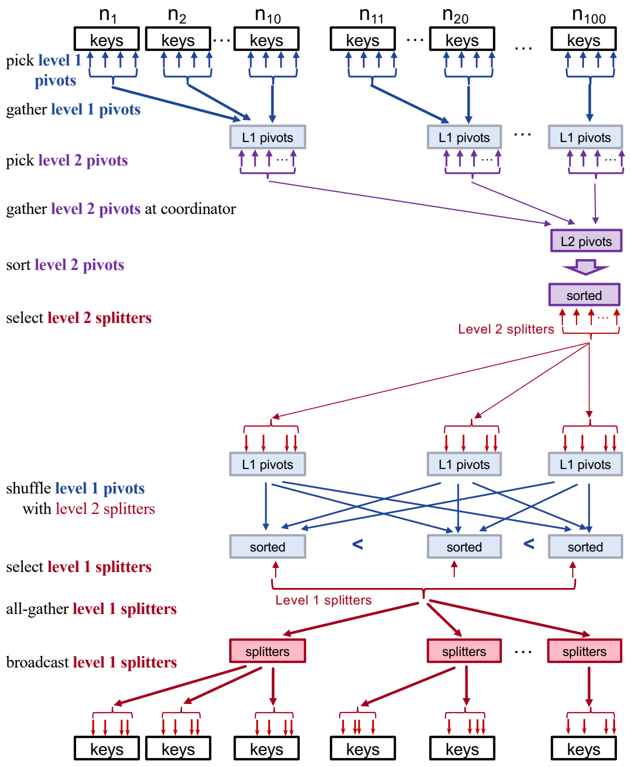


Figure 3: A two-level partitioning example with $M = 100$, $s = 1$, $r = 10$.

pivot sorters, M/r^2 second-level pivot sorters, and so on. For the MilliSort implementation we used a reduction factor of 10.

4.4 Improved splitter selection

The algorithm described above for choosing splitters from the sorted pivots results in uneven key distribution, where the first server is likely to be assigned 50% more keys than the average, and the last server is likely to be assigned 50% fewer keys than the average. We developed an improved splitter selection algorithm that eliminates this imbalance; because of space limitations, we describe that algorithm in Appendix A.1.

4.5 Local sort

Now that the partitioning mechanism has been described, the next few subsections discuss the remaining phases of MilliSort. These phases are more straightforward than the partitioning phase, though their performance is still important.

MilliSort relies on prior work for the local sort. The problem of sorting on a single compute server is well-studied, with many solutions that can take advantage of multiple cores. In our experiments we use the In-place Parallel Super Scalar Samplesort (IPS^{4}o) algorithm [7] as a reasonable representation of a multi-core comparison-based sorting algorithm.

Once the keys have been sorted, the values must be rearranged to match the order of the keys. This can be overlapped with the partitioning phase and the key shuffle; the rearranged values are not needed until the value shuffle.

4.6 Shuffle

Once the splitters have been selected and distributed to all of the machines, MilliSort uses an all-to-all shuffle to transmit each key and value to the appropriate server. The keys and values on each server were already sorted during the local sort phase. Each server uses the splitters to determine the range of keys to send to each other server. It then sends a message to that server containing the appropriate range of keys, followed by the data associated with those keys.

4.7 Local rearrangement

Within each message that a server receives during the shuffle phase, the keys and values are in sorted order. The server must then combine these chunks into two sorted arrays, one containing all the keys and other containing all the values. To do this, each server first performs a merge sort on the arrays of keys; then it rearranges the values to match the order of the keys. Each key contains its index in the incoming shuffle message (which is the same as the index of the value in the message). Once the keys have been sorted, a sequential scan is made over the key array; the index stored with each key is used to find the corresponding value and the value is then stored at the next sequential location in the result array. This two-step approach is faster than a one-step approach that merges both keys and values together, because it copies the (larger) values only once.

5 Implementation

5.1 Group communication

We created a C++ library that implements the group communication primitives described in Section 2.4; both MilliSort and MilliQuery make extensive use of this library. The group communication primitives are implemented using the network transport infrastructure from RAMCloud [50]. RAMCloud's transports use kernel bypass with either DPDK [10] or the Infiniband verbs interface to provide low latency (5 µs round-trips) and high throughput (up to 25 Gbps). However, RAMCloud requires all network communication to pass through a single dispatcher thread, which limits message throughput to about 1.6 million messages per second. The group communication primitives contain 1500 lines of C++ code, not counting code in RAMCloud.

Broadcast, gather, and all-gather were implemented using well-known approaches [62, 65]. For broadcast, MilliSort uses a k-nomial tree, with the topology optimized based on precise knowledge of message latency and per-message cost. For gather, MilliSort uses a k-nomial tree with $k = 6$, in order to utilize as much network bandwidth as possible at each step. For all-gather, MilliSort uses a hypercube approach, extended to handle group sizes that are not even powers of two [56].

Shuffle is the most important of the group communication primitives: it accounts for half or more of the end-to-end time in the highest performing MilliSort configurations and it is also used in two of the three MilliQuery queries. However, achieving high-performance for shuffle is challenging. Ideally, shuffles should utilize the full bandwidth of the network, with each host simultaneously sending and receiving at the speed of its uplink. However, achieving this goal is difficult. One problem is small messages, which are more likely to occur in flash bursts than other applications (see below). Another problem is that shuffle requires full bisection bandwidth in the underlying network. Fortunately, our test cluster has full bisection bandwidth. Unfortunately, we were unable to harness all of the available bandwidth.

Achieving full bisection bandwidth requires a near-perfect bipartite matching, where at any given time each source transmits at full bandwidth to a different target. This is difficult to achieve, for two reasons. First, two servers might attempt to transmit simultaneously to the same target. When this occurs, bandwidth is wasted on the senders, since the target can only receive from one of them at a time; in addition, some other server will not be receiving anything at all, which wastes its incoming bandwidth. Second, achieving full bisection bandwidth requires perfect load-balancing across the network fabric. Unfortunately, our test cluster does not support packet-level load-balancing. Instead, it uses flow-consistent hashing, where all packets from a particular source to a particular destination are routed over a single (randomly chosen) path through the fabric. Even if two sources send to different targets, their routes might traverse a common intermediate link, resulting in bandwidth underutilization. With large clusters, routing conflicts are virtually guaranteed.

After implementing shuffles in the most obvious way and observing poor performance, we attempted a lock-step approach to ensure a bipartite matching. In the lock-step approach, in step i each server n transmits to server $(n+i) \bmod M$, and the start of step $i+1$ is delayed until step i has completed. This mostly eliminated the problem where two senders transmit to the same target, but it works best when all messages are large and the same size. If messages are small, or if messages have different sizes, the act of maintaining lock-step wastes most of the network bandwidth (e.g. each step must wait for the longest message in the preceding step to complete). Furthermore, lock-step is still vulnerable to conflicts in network routes. We were unable to achieve a satisfactory level of performance with this approach.

We then switched to a nearly-opposite approach, implementing shuffles in a "high-entropy" fashion that is granular, concurrent, and random. The first step is to ensure that messages sent during shuffles are relatively short (at most 40 KB in the current implementation). In many cases, the messages are inherently short; if the data from one server to another exceeds a threshold size, it is divided into multiple short chunks, which are sent as separate messages. Each host sends multiple messages concurrently; the targets are chosen randomly, but a given source will have at most one message outstanding to a given target at a time. With this approach there will still be conflicts but they will not stall senders; conflicts simply result in packet queueing in the network. Since each sender has multiple outstanding messages, it is likely that there will be incoming data for each server at all times. Since messages are short, buffer overflows in the network are unlikely and a sender doesn't waste much time on a busy receiver before directing its bandwidth elsewhere. Stalls will occur only at the end of the shuffle, when a sender is waiting for its last few messages to complete. The high-entropy approach resulted in much better performance than the alternatives we tried before it.

Shuffles are more challenging in a flash burst than in more traditional environments that operate at large timescales. A flash burst scales by increasing the number of servers, with less data on each server, in order to complete more quickly. Less data on each server means the size of each shuffle message will drop; more servers means that each server's data is split among more shuffle messages, which also results in less data per message. The result is a rapid drop in shuffle message size as a flash burst scales. This leads to low network bandwidth utilization and high per-message overheads.

One possible solution is to use a two-level shuffle to reduce the number of messages sent and received by each server from $M-1$ to $2\cdot(\sqrt{M}-1)$. Two-level shuffle arranges all servers into a virtual mesh and proceeds in two rounds: each server first exchanges data with other servers in the same row, and then in the same column. However, this approach doubles the network bandwidth consumed, since most data must be transmitted twice. For our experiments the increased bandwidth usage of a two-level shuffle was more problematic then per-message overheads for a single-layer shuffle; we did not find any situations where multi-level shuffles are advantageous.

5.2 MilliSort

The MilliSort implementation is fully decentralized: each server operates independently in an event-driven style, with no central coordinator (central coordination is intolerable for flash bursts, both because of the latency it adds and also because central coordination often involves lock-step operation at the end of each stage, which suffers from stragglers). While the order of stages executed in a server is well defined, the timing is affected by remote procedure calls (RPCs) from other servers. At various points, the progress of the server will stall until certain pieces of data have arrived. As one example, a pivot sorter cannot select level 2 pivots until it has received pivots from all of the servers in its group.

Different servers must have different behaviors during the sort. For example, only a subset of the servers will act as pivot sorters. Each server has an identifier ranging from 0 to $M-1$, which determines the various roles it will serve during the sort. For example, servers with identifiers that are $0 \bmod r$ serve as pivot sorters and they receive pivots from servers with the following $r-1$ identifiers. At the start of the sort each server knows its identifier, the value of M, and the addresses of the other servers.

Achieving the highest overall performance for MilliSort requires careful optimization of both communication and computation. Section 5.1 has already discussed the challenges associated with shuffles. Using cores efficiently is another challenge, because the number of threads changes rapidly over the life of the sort. Thus, MilliSort uses Arachne [54] for efficient user-level thread and core management. For example, Arachne allowed us to quickly spawn a group of threads to parallelize a task, such as local sort, and place them precisely on all available cores. In addition, MilliSort creates a new thread for each incoming RPC and leaves it to Arachne to schedule those threads on cores that are less busy.

5.3 MilliQuery

We created a special-purpose implementation of each of the three MilliQuery queries, largely based on the query plans generated by BigQuery. These queries were much easier to implement than MilliSort, particularly given the availability of the group communication library. The total implementation time was only a few days, and the three queries contain 250, 300, and 800 lines of C++ code, respectively.

The implementation of Q1 follows a simple scan-aggregate pattern: each server scans its data independently to count the views by language, then the local results are gathered back to one server, using a k-nomial tree and combining the statistics at each node. Since the number of distinct languages is only a few hundreds, all messages in the gather phase are small.

Similar to Q1, Q2 can also be implemented as local scan followed by a gather. But, the number of distinct IP addresses is quite large, so the gather phase would consume too much network bandwidth with this naive approach. Thus, before the gather phase, a shuffle is used to collect all the counts for each address in one place, using a hash partition. Then each node in the gather tree collects local results from all its children, but only needs to send the top ten IP addresses to its parent.

Q3 is considerably more complex and requires a total of three shuffles. First, two shuffles are used to materialize two intermediate tables, distributing the records for each table using a hash partition with the `repo_name` field. Then, the two tables are joined locally using the same key, and a third shuffle redistributes the joined records by hash partitioning on `(lang, author)`. Finally, top-100 statistics are computed locally and aggregated as in Q2.

For simplicity, we didn't implement additional mechanisms to handle hot keys in hash partitioning for Q2 or Q3; however, it's possible to use MilliSort's recursive partitioning scheme to create a more balanced range partitioning, at the expense of higher partitioning cost.

5.4 Communication infrastructure

We chose to build our MilliSort and MilliQuery prototypes atop RAMCloud [50] mainly to reuse its flexible network infrastructure. However, this choice comes with the cost of limited message throughput for two reasons:

- The single dispatch thread in RAMCloud has relatively high per-message costs (all messages must pass through the dispatch thread, which results in expensive cross-core communication) and presents a central bottleneck which prevents us from achieving higher throughput by adding more cores.

CPU	Xeon Gold 6148 (2 sockets $\times$ 20 cores @ 2.40GHz)
RAM	384 GB DDR4-2666
Networking	100Gbps Intel Omni-Path Interconnect

Table 2: The hardware configuration used for benchmarks. All machines ran CentOS 7.3.1611 (Core) with hyperthreading disabled. The network fabric used a two-level fat-tree topology to provide full bisection bandwidth at 100 Gbps per machine.

	Total records processed (# servers used)			
Time budget	**MilliSort**	**Q1**	**Q2**	**Q3**
1 ms	0.84 M (120)	47.6 M (280)	6.72 M (140)	0.034 M (60)
10 ms	26 M (280)	980 M (280)	224 M (280)	2.24 M (280)
Scaling	31.0x (2.3x)	20.6x (1x)	33.3x (2x)	67.9x (4.7x)

Table 3: Overall performance of MilliSort and MilliQuery.

- The underlying Omni-Path PSM2 library [26] we use to send and receive packets is actually not a packet I/O driver but a reliable message-passing transport with its own congestion/flow control, which adds considerable overhead; in addition, the Omni-Path host fabric interface (HFI) does not have a lightweight mechanism for transfering small chunks of data via DMA, so we have to send/receive packet data using programmed I/O, which leads to higher CPU overhead and, even worse, cache pollution.

As a result, a single server cannot fully utilize its network bandwidth when the messages are relatively small. In our experiments, we decided to place multiple servers on each machine, each with its own dispatch thread, to increase the network utilization. This approach can scale the overall message throughput of each machine linearly, but it also incurs higher coordination overhead due to the increased number of servers. Future implementations that are based on a more efficient networking stack such as [32, 18] could eliminate the need to run multiple servers per machine.

6 Performance measurements

Our goal in evaluating MilliSort and MilliQuery was to answer the following questions:

- How much data can be processed in one millisecond (or ten milliseconds), and how many servers can be harnessed efficiently in each interval?
- How does application behavior change if the time budget is increased or decreased?
- What factors limit the applications' performance and scalability, and what stresses do these applications create for underlying infrastructure?
- How effective is MilliSort's new partitioning mechanism?
- How efficient is MilliSort compared to purely local sort or other distributed sorts?

We used the hardware configuration shown in Table 2 to evaluate MilliSort and MilliQuery. To better utilize the network bandwidth (discussed in Section 5.4), we ran four independent servers on each machine, two on each socket. We had access to 70 machines in the cluster, which allowed up to 280 servers; and there were no competing tasks running on the machines. All MilliSort experiments used 2-level partitioning.

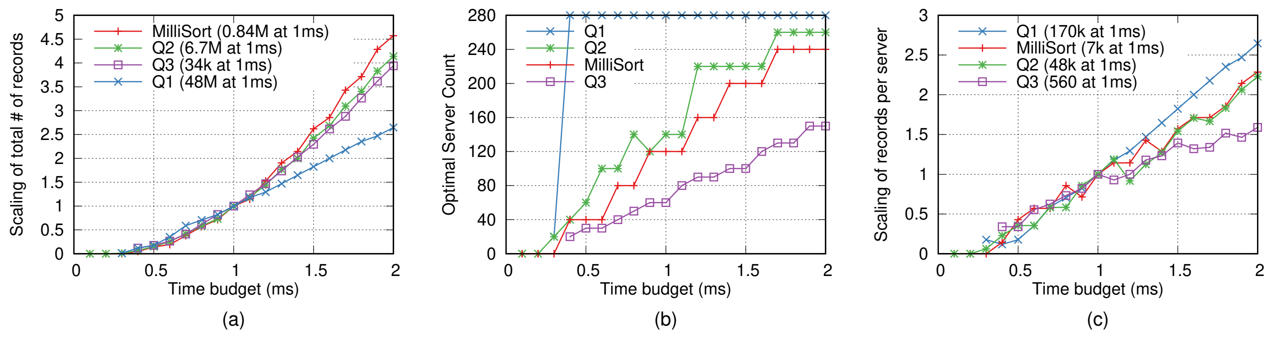


Figure 4: Scaling properties of MilliSort and MilliQuery as a function of time budget: (a) size of dataset processed (normalized to a value of 1.0 for a 1 ms time budget for each benchmark); (b) cluster size that yields the largest dataset processed; (c) records per sever at largest dataset processed (also normalized to a value of 1.0 for a 1 ms time budget).

	120 servers (0.84M records)				**240 servers** (1.68M records)			
Phase	Mean	Max	%	Max/Min	Mean	Max	%	Max/Min
Local Sort	147.0	216.3	22.2%	1.83	137.8	202.3	12.7%	1.77
Partitioning	200.5	214.4	22.0%	1.16	410.4	428.6	26.9%	1.11
Pivot Shuffle	83.2	87.9	9.0%	1.13	219.3	240.1	15.1%	1.27
Shuffle	377.2	402.8	41.3%	1.16	738.9	789.0	49.6%	1.14
Rearrangement	128.1	142.7	14.6%	1.18	146.9	173.3	10.9%	1.26
Total	942.3	976.0	100%	1.09	1523.8	1591.1	100%	1.08

Table 4: Time breakdown of each MilliSort phase with two different cluster sizes, where per-node dataset sizes are fixed at 7000 records. All times are in µs and reflect the median over 200 runs. "%" is the time spent by the slowest server for that phase, as a fraction of total time. "Max/Min" is the ratio of times for the slowest and fastest servers for that phase. "Partitioning" includes the time spent on "Pivot Shuffle".

6.1 Overall performance

We varied the amount of data per server and the size of the cluster to find the largest amount of data that can be processed by each of the applications in either 1 ms or 10 ms; Table 3 shows the results, along with the best configurations for each interval. With a 10 ms time budget all of the applications can harness all 280 available servers and they can process 2.2–224M records, depending on the application. A 1 ms time budget is more challenging for most of the applications. Only MilliQuery Q1 can use all of the servers; the other applications ranged from 60–140 servers. The amount of data processed in 1 ms varied dramatically among the applications, from a low of 34K records in MilliQuery Q3 to a high of 48M records in Q1.

6.2 Quadratic scaling

The total number of records that can be processed in an interval varies quadratically with the size of the interval. If the time budget increases by a factor of X, we observe that both the data handled by each server and the number of servers increase by at least X, for a total increase in throughput of at least X^2. This effect can be seen in Figure 4. Figure 4(a) shows that all of the benchmarks except Q1 exhibit quadratic or better scaling as the time budget increases from 1 ms to 2 ms. Q1 would scale exponentially if more servers were available, but it already used all of the available servers at 1 ms, so it can only scale linearly by increasing the amount of data per server. Figure 4(b) shows optimal cluster size as a function of time budget. All except Q1 exhibit almost linear scaling. Figure 4(c) shows the scaling of the per-server dataset size as the time budget increases. All except Q3 scale at least linearly, after a fixed initial overhead. Q3 scales per-server records slightly less than linearly.

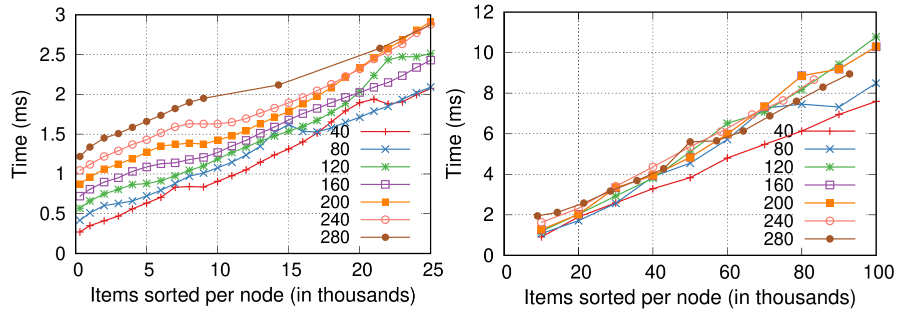
The quadratic behavior can also be seen for Q3 in Table 3: its throughput scales by 68x as the time budget increases from 1–10 ms. Scaling for the other applications becomes limited by the available servers long before reaching the 10 ms time budget; Figure 4(b) shows that MilliSort, Q1, and Q2 have consumed almost all the available servers with a 2 ms time budget. Linear scaling of servers suggests that these applications could harness at least 1200 servers with a 10 ms budget (5x the number at 2 ms).

6.3 Scaling below 1 ms

Quadratic scaling also means that throughput drops rapidly with time budgets less than 1 ms. This is visible in Figure 4(a). With a time budget of 0.5 ms, throughput has dropped by more than 4x for all of the applications, and none of the applications has appreciable throughput for budgets less than 0.5 ms. For these applications, the lower bound on useful timescale is around 0.5–1.0 ms with our current per-message overheads.

6.4 Limiting factors for scalability

There are two primary factors that limit the ability of the applications to scale up in servers or down in time: coordination and shuffles. The costs of both activities increase with the cluster size; when the amount of data per server is zero, these are essentially the basic costs of harnessing servers. Table 4 illustrates the effect of these two factors by showing the cost of each MilliSort phase for two different cluster sizes with the records per server held constant. In the 120-server configuration, partitioning and shuffles take 63% of the total running time. If the cluster size is doubled, the time taken by these two activities is also doubled (1218 µs vs. 617 µs), even though each server still processes roughly the same amount of data. The fraction of total running time consumed by coordination and shuffles increases from 63% to 77%. At the same time, the combined cost of other phases remains almost constant in both configurations. In general, when the time budget is held constant, larger clusters result in larger basic costs, so less time is left for actual work such



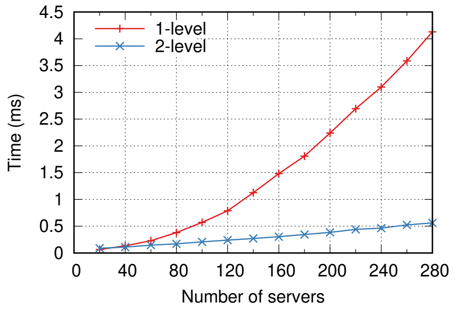


Figure 5: Total sorting time for MilliSort as a function of records per server, with different cluster sizes. The left figure assumes a 1 ms goal and the right figure assumes 10 ms. Each line represents a different cluster size (the number in the legend is the server count). Each data point is the median time from 200 runs.

Figure 6: MilliSort's partitioning time as a function of cluster size with different partitioning schemes.

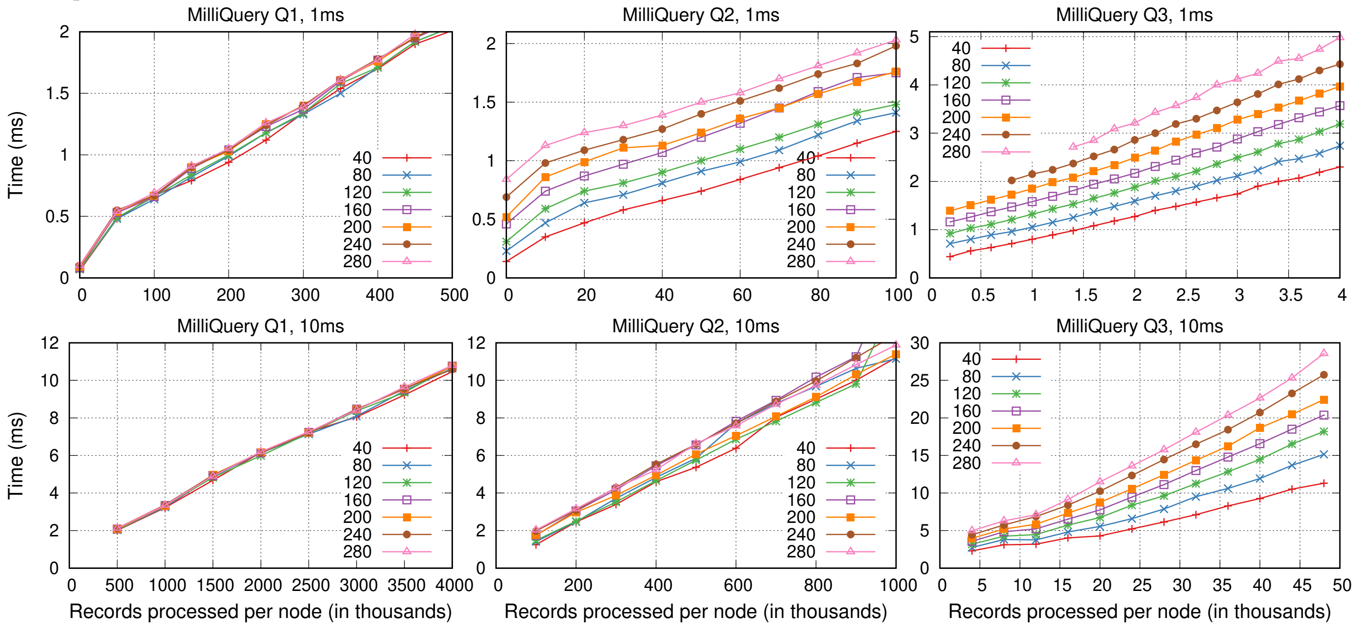


Figure 7: Total processing time for the three MilliQuery queries as a function of records per server, with different cluster sizes. From left to right, the figures represent Q1, Q2, and Q3 respectively. Each data point is the median time from 100 runs.

as local computation and data transfer. As a result, this limits our ability to harness more servers within the time budget or perform meaningful flash bursts in smaller timescales.

Figure 5 and 7 provide more details of the two limiting factors for MilliSort and MilliQuery by showing how the total processing time changes with the cluster size and amount of data per server. In most graphs of Figure 5 and 7, the lines for different cluster sizes are roughly parallel, indicating that the marginal cost of handling additional data is about the same for all sizes (the marginal cost starts higher when the amount of data per server is smaller, but plateaus out quickly once the benefit of batching diminishes). However, larger cluster sizes have larger fixed overheads (y-intercepts of the lines), which consist of the partitioning cost plus the per-message costs of the shuffles (a shuffle must send one message to each peer, even if it only contains a single record). These figures also show the diminishing returns at timescales less than 1 ms: even as the number of records per server approaches zero, running times remain 0.3–1.2 ms and 0.15–0.84 ms for MilliSort and MilliQuery Q2 respectively, depending on cluster size.

MilliQuery Q3 difers from the other applications in that its total processing time increases more than linearly with the input records per server, and the rate of increase is higher for larger clusters (the gaps between the lines in the right graphs of Figure 7 become wider for larger numbers of input records). This is because Q3 uses join operations where the number of output records tends to increase quadratically with the number of input records, so the number of input records doesn't reflect the actual number of records each server has to process.

Different applications have very different fixed overheads of harnessing servers. Q1 has the smallest fixed overheads and they are about the same for all cluster sizes. This is because Q1 requires very little coordination and doesn't use shuffle; the only communication primitives used by Q1 are broadcast and gather, whose overheads increase only in log scale with the cluster size. MilliSort and Q2 are very similar: their fixed overheads increase almost linearly with the cluster size since both applications require a big shuffle at the end. However,

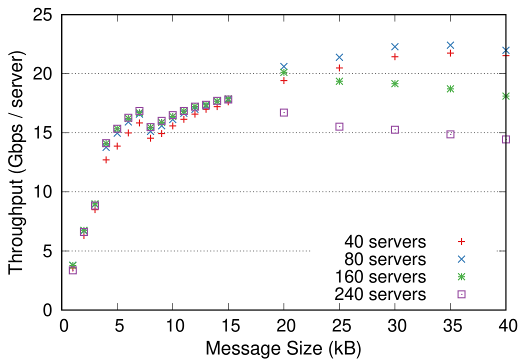


Figure 8: Stand-alone shuffle benchmark performance as a function of the cluster size and the message size (amount of data sent from each source to each destination).

for the same cluster size, the fixed overhead of MilliSort is higher due to the additional partitioning cost (with 2-level partitioning, this cost also increases almost linearly with the cluster size; this will be discussed in Section 6.6). For the same reason, the lines of Q2 in the graphs are closer than the lines of MilliSort. Finally, the fixed overheads of Q3 are about 3x higher than Q2 for all cluster sizes (unfortunately, it's hard to see in the graphs), as Q3 requires a total of three shuffles.

Figure 5 and 7 also show that efficiency improves with increasing records per server. For MilliSort, once the number of records per server reaches about 25000, the coordination costs become small compared to other factors and the shuffle efficiency improves, so there is little difference in overhead between different cluster sizes (the 40-server cluster remains significantly more efficient than the other cluster sizes because it avoids contention in the network core; this will be discussed in Section 6.5). The closeness of the lines in these 10 ms graphs indicates that all applications except Q3 could easily harness many more than 280 servers efficiently with a 10 ms time budget.

The next subsections discuss shuffle and partitioning costs in more detail.

6.5 Shuffle efficiency

Shuffles present the most significant challenge to scalability in our experiments, especially at 1 ms time scales. This is reflected in Table 3: Q1, which has no shuffles, can harness far more servers and process far more data than the other benchmarks, while Q3, which has three shuffles, is the least scalable. MilliSort and Q2, which each use one shuffle, fall in between. In Table 4, when the number of servers doubles while fixing the number of records per server, 85% of the time increase comes from additional shuffle costs; shuffle costs affect not only the main shuffle phase, but also the partitioning phase.

To get a better understanding of shuffle costs, we ran a stand-alone shuffle benchmark in which each of a group of servers sends a fixed-size message to each other server. Figure 8 graphs shuffle performance in terms of throughput per server. We ran four servers per machine, so the ideal throughput per server would be 25 Gbps.

We observed two different potential reasons that contribute to the higher shuffle time with more servers. First, as the message size decreases, efficiency drops because of per-message overheads. With 120 servers and 0.96M total records, the average message size for shuffle is about 6.7KB, which still provides a good throughput. However, with 240 servers and 1.92M total records, the average shuffle message size drops to about 3.3KB, resulting in less than 10 Gbps throughput per server. This also justifies the quadratic scaling property discussed in Section 6.2; when the number of servers and number of records per server are scaled simultaneously, we have a better chance of maintaining shuffle efficiency since the average message size for shuffle is fixed.

Second, even with large messages, throughput per server drops as the cluster size increases. It drops from 22 Gbps per server (with 40 servers) to less than 15 Gbps per server with 240 or more servers. This is because the cluster network uses flow-consistent load balancing, rather than packet-level load balancing. With large numbers of active transmissions, paths conflict in their link usage, resulting in congestion on those links and under-usage of other links. As the cluster size increases, shuffles consume a larger fraction of the core bandwidth, which makes congestion more likely, and our high-entropy approach to shuffles cannot completely compensate. As a result, shuffles cannot harness the full bisection bandwidth offered by the network. We speculate that more granular in-network load-balancing techniques such as [47, 64] may help remove this bottleneck in the future.

6.6 Partitioning cost

Figure 6 shows the results of a stand-alone experiment that measures partitioning time as a function of the number of servers. The multi-level partitioning algorithm we developed for MilliSort provides a significant peformance benefit: by the time the number of servers reaches 200, the 2-level approach is more than 5x as fast as the 1-level approach. As the number of servers increases, the partitioning time increases super-linearly for both 1-level and 2-level partitioning schemes. However, for 2-level partitioning, the increase in time is almost linear up to 280 servers (each additional server adds about 2 µs in our current implementation). A 1-level approach is too slow to harness 100 or more servers in a 1 ms budget, but a 2-level approach can easily coordinate 120 servers within 1 ms and 280 servers within 10 ms. In the best configurations for 1 ms and 10 ms time budgets, the partitioning cost only accounts for about 20% and 5% of the total time, respectively. Unfortunately, even with 2-level partitioning, the coordination cost for 280 servers is already more than half a millisecond; this prevents us from harnessing hundreds of servers in 1 ms for MilliSort. We didn't implement 3-level partitioning and beyond, but we expect that increasing the number of levels will further reduce the coordination cost for large clusters.

6.7 MilliSort efficiency

It might seem that flash bursts must sacrifice throughput (or efficiency) in order to operate at millisecond timescales.

	Throughput (efficiency)	
	120 servers (0.84M records)	**280 servers** (26M records)
MilliSort	7172 (19.6%)	10378 (33.1%)
Ideal Distr. Sort	11272 (30.8%)	13600 (43.3%)
Local Sort	36656 (100%)	31398 (100%)

Table 5: MilliSort efficiency of the best configurations for 1 ms and 10 ms time budgets, compared to local sort and ideal distributed sort. Throughput numbers are in "records/ms/server". Efficiency is the relative throughput compared to local sort.

	CPU	**SMT**	**BW/core**	**Throughput**
MilliSort	Xeon Gold 6148 @ 2.4GHz	1	3.1	1297 (1.00x)
TencentSort	IBM POWER8 @ 2.9GHz	8	5.0	1977 (1.52x)
CloudRAMSort	Xeon X5680 @ 2.9GHz	2	2.7	707 (0.55x)

Table 6: Per-core throughput comparison of MilliSort, Tencent Sort [28], and CloudRAMSort [34]. "SMT" (i.e., simultaneous multithreading) is the number of hardware threads per core. "BW/core" is the average network bandwidth per core, in Gbps. "Throughput" numbers are in "records/ms/core". MilliSort's per-core throughput is computed from the 10 ms configuration in Table 5, while Tencent Sort and CloudRAMSort's numbers are computed from their published results [28, 34]. Tencent Sort's reported throughput is for sorting from disk to disk; we estimate that less than half of the time is spent on disk I/O, so we double its reported throughput for a fair comparison.

However, our results suggest that this need not be true. Table 5 shows the overall efficiency of MilliSort compared to purely local sort and ideal distributed sort. The ideal distributed sort represents an imaginary situation where the partitioning cost is zero and data are shuffled at full network bandwidth; this provides an upper bound on the throughput of any distributed sort. Despite operating at a 1 ms timescale, MilliSort is relatively efficient (e.g., the maximum possible throughput for distributed sorts is only about 50% higher); MilliSort is even more efficient at a 10 ms timescale.

MilliSort's throughput is also on par with state-of-the-art sorting systems [34, 28] that have much longer running times. Table 6 summarizes the hardware differences and presents the throughput numbers in "records/ms/core" for comparison. In particular, Tencent Sort [28] is the current record holder for the GraySort benchmark [61], and its total running time is about 100 seconds. MilliSort's throughput per core at 10 ms is only 33% lower than Tencent Sort, even though Tencent Sort's POWER8 cores have 8x the hardware threads and 1.6x the average network bandwidth of MilliSort cores.

Table 5 also provides insight into why distributing data at very fine granularity makes sense. Network communication is the largest source of overhead for distributed computation (the throughput of local sort is 3–5x higher than MilliSort in Table 5). However, most of this cost is paid immediately when scaling beyond a single machine. For example, performing a distributed sort with just two machines requires half of the data to be sent over the network. There is not much additional loss in efficiency when scaling out further. This suggests that if you can afford to distribute at all, you can afford to distribute a lot.

7 Observations

This section summarizes the key observations that emerged from our MilliSort and MilliQuery experiments:

Granular data distribution. Distributing data in fine granularity allows one to harness more CPU cores and aggregated network bandwidth for faster computation and communication. Our experiment also confirms that, at least for some dwarfs, it is possible to scale out quite efficiently even at millisecond timescale. Thus, we predict that future distributed data-parallel systems will need to be optimized for large scale-out architectures with smaller data per server. This will likely present interesting challenges to the design of future systems since they will be required to scale down gracefully (i.e., operate efficiently even on smaller data).

Efficient group communication is essential. Even simple dwarfs have complex communication patterns internally; all of the benchmarks except MilliQuery Q1 depend heavily on group communication. Even when carefully optimized, they account for 50%–60%, 35%–40%, and 50%–65% of the overall running time in the best configurations of MilliSort, MilliQuery Q2, and MilliQuery Q3, respectively.

Per-message overhead is critical. Network bandwidth and latency are well known to be important for the performance of large-scale systems, and they remain important for flash bursts. Flash bursts differ from traditional large-scale systems in that per-message costs are at least as important as the traditional metrics. This is because group communication primitives tend to generate many small messages when operating at small time scales with large cluster sizes.

Coordination must be structured hierarchically. It is well known in the HPC community that communication must be structured hierarchically to handle large cluster sizes. Thus, group communication primitives are commonly implemented in a tree or hypercube topology [62], so that each machine only communicates with a small number of its peers. This helps to mitigate per-message overheads and harness more aggregated network bandwidth. This principle also applies to coordination. As an example, MilliSort's partition phase is structured as a hierarchical series of distributed sorts, so that the computation doesn't become a central bottleneck.

Low-latency shuffle is challenging. Of all the group communication primitives, shuffle is the most challenging in flash bursts. There are several reasons for this. First, per-message overheads become critical since shuffles require each server to send many small messages. Unlike other primitives, a hierarchical approach to shuffles generally isn't practical because it doubles the network bandwidth utilization. Second, shuffles need to use the full network bandwidth of each server for best performance, but many networks do not provide full bisection bandwidth, especially at large scale. Even where full bisection bandwidth is available, transient congestion can occur due to imperfect bipartite matching, either in the network itself (because of routing) or in the application.

Finally, hash partitioning often results in non-trivial data imbalance (e.g., in MilliQuery Q2, the most unlucky server typically has to process 1.5x much data as the average).

Quadratic scaling. As the time budget increases, the number of machines that can be harnessed effectively increases at least linearly. The amount of data each server can process also grows at least linearly with the time budget, so the overall dataset size grows at least quadratically with time budget. For some workloads, such as MilliQuery Q1, where the only coordination overhead is a final aggregation, the number of machines can grow exponentially with the time budget, so the dataset size grows faster than quadratically. The flip side of this is that reducing the time budget results in superlinear reductions in the number of servers and overall dataset size, which creates a lower limit on the timescale at which the system can operate efficiently. In our experiments, 1 ms appears to be tractable for all of the workloads except MilliQuery Q3. Future systems that are built on top of better networking infrastructure will likely enable these workloads to run efficiently at an even smaller timescale.

Ultimate limits of strong scaling. If we increase the number of machines while fixing the dataset size, efficiency inevitably decreases; this eventually prevents us from harnessing more machines to speed up the job. While it is well known that the theoretical speedup of a parallel program is ultimately limited by its serial portion (Amdahl's Law), this is not the limiting factor in our experiments. Instead, two other factors result in the drop in efficiency. First and foremost, coordination cost rises. This is mostly due to per-message overheads (e.g., manifested as higher shuffle costs). Coordination algorithms also take longer to run (e.g., MilliSort's partitioning time increases linearly with the number of servers, even with the recursive scheme). Second, straggler effects are more significant when there are more servers and less data per server (e.g., even in MilliQuery Q1, the overall efficiency drops about 50% when scaling from 40 servers to 280 servers).

8 Applicability of results

We conducted our experiments with limitations and assumptions that may not apply to today's computing systems. In this section we discuss some of these factors and argue that our results will be relevant for future systems.

Is 1–10 ms the right target? 1–10 ms is not the timescale at which most people think of large-scale distributed computation today. For example, response times for users of a few hundred milliseconds are considered adequate, and it can take 100 ms just to communicate between browser and datacenter. However, applications such as AR/VR require response times of 10 ms or better, and new edge computing offerings such as AWS Local Zones and WaveLength [6] can already provide single-digit millisecond latency between the end-user and an edge cloud. Furthermore, there are increasing numbers of applications that make real-time decisions without humans in the loop. Examples include controllers for autonomous vehicles [42] and IoT devices [24], which make decisions on the order of 10 ms, and financial applications [22, 51], for which there appears to be no lower bound on desirable latency. Flash bursts can enable these applications to run data-intensive algorithms at millisecond timescale.

Idealized experiment setup. Our experiments are conducted on a bare-metal HPC cluster with no interference from competing workloads. A dedicated cluster is not economically feasible in practice; however, we believe that recent work on colocating latency-critical and batch jobs [48, 18] can be applied to achieve high CPU efficiency without hurting the performance of flash bursts. In addition, we assume input data are resident in memory when the experiments start. This may not be a realistic assumption today, where data is stored on flash or disk. However, future datacenter systems are likely to store data in nonvolatile memories (NVMs) with access times not far above today's DRAM [27]. Ideally, future applications will run directly on NVM-based storage servers; in the worst case an initial shuffle step will be needed to extract data from the storage servers to computational nodes.

Missing pieces. Given the tight time budget, a flash burst requires every component involved in its lifetime to support low latency. There are several elements that our work did not address, such as application loading, but there are many other projects attacking these pieces, such as storage systems [49, 12, 41, 36, 25], cluster schedulers [31], networking infrastructure [46, 32, 43, 37, 14], fast threading and dispatching [8, 52, 30, 48, 54], and lightweight virtualization [1]. Many interesting problems have yet to be solved [39] in order to create a unified flash burst infrastructure.

9 Related work

There are several efforts underway to support shorter task durations in a cloud setting. For example, major serverless platforms [5, 45, 20] can support tasks, also called serverless functions, as small as 100 ms. Although serverless functions were initially intended for simple microservices, many projects have successfully used them to parallelize jobs over thousands of cores in the cloud. To overcome the relatively high overheads of the serverless platform, these projects typically targeted compute-intensive workloads and large time budgets such as 1 min. ExCamera [17] harnessed 3600 cores for 2 mins for video encoding; gg [16] harnessed 384 cores for 1.5 mins for software compilation; PyWren [29] sorted 1TB of data in 204 seconds with 1000 workers; Sprocket [3] harnessed 1000 cores for less than 1 min for video processing. Finally, these projects have mainly focused on exploiting large parallelisms in applications and reducing monetary cost, as opposed to revealing or addressing the new problems that emerged by coordinating many nodes at small timescales.

In addition to serverless computing, HPC is another area that greatly inspires the development of flash bursts. Scalability is of utmost importance to HPC applications, and highly-optimized HPC applications can scale beyond ten thousand compute nodes in a supercomputer. Furthermore, unlike many data analytic workloads, traditional HPC workloads such as scientific simulation typically require frequent communication between compute nodes (e.g., at the end of each simulation timestep). Flash bursts share the goal of running communication-intensive workloads in a scalable way, and we borrow techniques that are well explored in the HPC community (e.g., efficient group communication primitives) to achieve this goal. However, flash bursts differ from HPC in two important aspects. First, HPC workloads usually run for much longer time (hours or more), so they don't expose the interesting problems of operating at millisecond timescales. Second, HPC workloads usually run in a cleaner environment which has less interference from other jobs, so system-level challenges such as performance isolation and resource utilization are less of a concern to HPC.

Shuffle is known to be one of the most expensive operations in distributed data analytics. Many projects have focused on optimizing its performance. Riffle [68] and Magnet [57] are two recent shuffle services designed to optimize disk-to-disk shuffle performance in Spark. Locus [53] implemented shuffle in a serverless setting by mixing different cloud storage services to balance shuffle performance and storage cost. Dataflow Shuffle [9] is an in-memory shuffle tier used by Google BigQuery. These systems focused on shuffling large data and did not address the challenges in low-latency shuffle.

Distributed sorting has been studied extensively for many decades, with a variety of well-established benchmarks [61]. Most prior work has focused on sorting on-disk data at large scale (e.g., TritonSort [55]). One exception is CloudRAMSort [34], which is designed to sort in-memory data to speed up database operations. It can sort 1 TB data in 4.6 seconds using 256 servers. As a comparison, the per-core throughput of MilliSort at 10 ms is almost 2x higher than CloudRAMSort despite operating at 1000x smaller timescales.

Another approach to speed up data-intensive computation is to exploit shared-memory parallelism on more powerful servers (i.e., scale-up). The major benefit of scale-up systems is efficiency because they can avoid the expensive network communication and overhead induced by fault tolerance. As shown in [13, 59], scale-up systems can be orders of magnitude faster than traditional systems like Spark [67]. However, there is a practical limit on the number of CPU cores available on a single server (a few hundred). Also, scale-up systems are much less flexible when the application requires fast data movement: if the input data need to be loaded over the network, the bandwidth of a single server will become a significant bottleneck. As a result, we expect both scale-up systems and flash burst to be valuable depending on the application.

10 Conclusion

MilliSort and MilliQuery demonstrate that several core patterns in data analytics can run efficiently at millisecond timescales. With a budget of 1 ms, most of the patterns can harness at least 100 servers; with a budget of 10 ms, our results suggest that most of the patterns can harness at least 1000 servers. Furthermore, our results indicate that there is only a small efficiency penalty for running at millisecond timescales. We identified two related problems that currently limit scalability: per-message costs and shuffle overheads. If future systems can improve on our implementation in these areas, it should be possible to execute large-scale computations even more efficiently, and at timescales even less than one millisecond. We do not yet know whether applications can take advantage of these small timescales, but we hope our results will encourage application developers to explore the potential benefits of running large-scale computations at millisecond granularity.

Acknowledgements

We thank our shepherd, Haryadi Gunawi, and our anonymous NSDI and OSDI reviewers for their feedback. We also thank Collin Lee for the early discussions that helped develop many of the ideas. This work was supported by the industrial affiliates of the Stanford Platform Lab, a Samsung Scholarship, and NSF grant CNS-191067.

References

[1] Agache, A., Brooker, M., Iordache, A., Liguori, A., Neugebauer, R., Piwonka, P., and Popa, D.-M. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)* (2020), pp. 419–434.

[2] Anon et al, Bitton, D., Brown, M., Catell, R., Ceri, S., Chou, T., DeWitt, D., Gawlick, D., Garcia-Molina, H., Good, B., Gray, J., Homan, P., Jolls, B., Lukes, T., Lazowska, E., Nauman, J., Pong, M., Spector, A., Trieber, K., Sammer, H., Serlin, O., Stonebraker, M., Reuter, A., and Weinberger, P. A Measure of Transaction Processing Power. *Datamation 31*, 7 (Apr. 1985), 112–118.

[3] Ao, L., Izhikevich, L., Voelker, G. M., and Porter, G. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing* (New York, NY, USA, 2018), SoCC 18, Association for Computing Machinery, pp. 263–274.

[4] Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D., and Yelick, K. A View of the Parallel Computing Landscape. *Commun. ACM 52*, 10 (Oct. 2009), 56–67.

[5] AWS Lambda. `https://aws.amazon.com/lambda`.

[6] AWS for the Edge: Bringing data processing and analysis closer to end-points. `https://aws.amazon.com/edge/`.

[7] AXTMANN, M., WITT, S., FERIZOVIC, D., AND SANDERS, P. In-place Parallel Super Scalar Samplesort (IPSSSSo). *CoRR abs/1705.02257* (2017).

[8] BELAY, A., PREKAS, G., KLIMOVIC, A., GROSSMAN, S., KOZYRAKIS, C., AND BUGNION, E. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proc. 11th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2014), OSDI'14, USENIX Association, pp. 49–65.

[9] BLOG, G. C. How distributed shuffle improves scalability and performance in cloud dataflow pipelines. `https://cloud.google.com/blog/products/data-analytics/how-distributed-shuffle-improves-scalability-and-performance-cloud-dataflow-pipelines`, 2018.

[10] Data Plane Development Kit. `http://dpdk.org/`.

[11] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM 51* (January 2008), 107–113.

[12] DRAGOJEVIĆ, A., NARAYANAN, D., CASTRO, M., AND HODSON, O. Farm: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Seattle, WA, Apr. 2014), USENIX Association, pp. 401–414.

[13] ESSERTEL, G., TAHBOUB, R., DECKER, J., BROWN, K., OLUKOTUN, K., AND ROMPF, T. Flare: Optimizing apache spark with native compilation for scale-up architectures and medium-size data. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)* (2018), pp. 799–815.

[14] FIRESTONE, D., PUTNAM, A., MUNDKUR, S., CHIOU, D., DABAGH, A., ANDREWARTHA, M., ANGEPAT, H., BHANU, V., CAULFIELD, A., CHUNG, E., ET AL. Azure accelerated networking: Smartnics in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)* (2018), pp. 51–66.

[15] Apache Flink - Stateful Computations over Data Streams, Jan 2021. `https://flink.apache.org`.

[16] FOULADI, S., ROMERO, F., ITER, D., LI, Q., CHATTERJEE, S., KOZYRAKIS, C., ZAHARIA, M., AND WINSTEIN, K. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (Renton, WA, July 2019), USENIX Association, pp. 475–488.

[17] FOULADI, S., WAHBY, R. S., SHACKLETT, B., BALASUBRAMANIAM, K. V., ZENG, W., BHALERAO, R., SIVARAMAN, A., PORTER, G., AND WINSTEIN, K. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)* (Boston, Massachusetts, USA, 2017), USENIX.

[18] FRIED, J., RUAN, Z., OUSTERHOUT, A., AND BELAY, A. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (Nov. 2020), USENIX Association.

[19] GAO, W., ZHAN, J., WANG, L., LUO, C., ZHENG, D., TANG, F., XIE, B., ZHENG, C., WEN, X., HE, X., ET AL. Data Motifs: A Lens Towards Fully Understanding Big Data and AI Workloads. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques* (2018), pp. 1–14.

[20] Google Cloud Functions. `https://cloud.google.com/functions/`.

[21] HARSH, V., KALE, L., AND SOLOMONIK, E. Histogram sort with sampling. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures* (New York, NY, USA, 2019), SPAA '19, Association for Computing Machinery, p. 201212.

[22] HASBROUCK, J., AND SAAR, G. Low-latency trading. *Journal of Financial Markets 16*, 4 (2013), 646–679.

[23] HOFMANN, M., RÜNGER, G., GIBBON, P., AND SPECK, R. Parallel sorting algorithms for optimizing particle simulations. In *2010 IEEE International Conference On Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS)* (2010), IEEE, pp. 1–8.

[24] HU, J., BRUNO, A., ZAGIEBOYLO, D., ZHAO, M., RITCHKEN, B., JACKSON, B., CHAE, J. Y., MERTIL, F., ESPINOSA, M., AND DELIMITROU, C. To centralize or not to centralize: A tale of swarm coordination. *arXiv preprint arXiv:1805.01786* (2018).

[25] Daos: Distributed asynchronous object storage, Aug. 2020. `https://github.com/daos-stack/daos`.

[26] Intel OPA-PSM2 Git Repository, Feb. 2020. `https://github.com/intel/opa-psm2.git`.

[27] IZRAELEVITZ, J., YANG, J., ZHANG, L., KIM, J., LIU, X., MEMARIPOUR, A., SOH, Y. J., WANG, Z., XU, Y., DULLOOR, S. R., ZHAO, J., AND SWANSON,

S. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module, 2019.

[28] JIANG, J., ZHENG, L., PU, J., CHENG, X., ZHAO, C., NUTTER, M. R., AND SCHAUB, J. D. Tencent sort.

[29] JONAS, E., PU, Q., VENKATARAMAN, S., STOICA, I., AND RECHT, B. Occupy the cloud: Distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing* (New York, NY, USA, 2017), SoCC 17, Association for Computing Machinery, pp. 445–451.

[30] KAFFES, K., CHONG, T., HUMPHRIES, J. T., BELAY, A., MAZIÈRES, D., AND KOZYRAKIS, C. Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (Boston, MA, 2019), USENIX Association, pp. 345–360.

[31] KAFFES, K., YADWADKAR, N. J., AND KOZYRAKIS, C. Centralized core-granular scheduling for serverless functions. In *Proceedings of the ACM Symposium on Cloud Computing* (2019), pp. 158–164.

[32] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (Boston, MA, 2019), USENIX Association, pp. 1–16.

[33] KALTOFEN, E. L. The seven dwarfs of symbolic computation. In *Numerical and symbolic scientific computing*. Springer, 2012, pp. 95–104.

[34] KIM, C., PARK, J., SATISH, N., LEE, H., DUBEY, P., AND CHHUGANI, J. CloudRAMSort: Fast and Efficient Large-Scale Distributed RAM Sort on Shared-Nothing Cluster. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012* (2012), pp. 841–850.

[35] KOWALEWSKI, R., JUNGBLUT, P., AND FRLINGER, K. Engineering a distributed histogram sort. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)* (2019), pp. 1–11.

[36] KULKARNI, C., MOORE, S., NAQVI, M., ZHANG, T., RICCI, R., AND STUTSMAN, R. Splinter: Bare-metal extensions for multi-tenant low-latency storage. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (2018), pp. 627–643.

[37] KUMAR, G., DUKKIPATI, N., JANG, K., WASSEL, H. M., WU, X., MONTAZERI, B., WANG, Y., SPRINGBORN, K., ALFELD, C., RYAN, M., ET AL. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication* (2020), pp. 514–528.

[38] LAKSHMANAN, V., AND TIGANI, J. *Google BigQuery: the Definitive Guide: Data Warehousing, Analytics, and Machine Learning at Scale*. O'Reilly Media, 2019.

[39] LEE, C., AND OUSTERHOUT, J. Granular computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (2019), pp. 149–154.

[40] LI, X., LU, P., SCHAEFFER, J., SHILLINGTON, J., WONG, P. S., AND SHI, H. On the versatility of parallel sorting by regular sampling. *Parallel Computing 19*, 10 (1993), 1079 – 1103.

[41] LIM, H., HAN, D., ANDERSEN, D. G., AND KAMINSKY, M. Mica: A holistic approach to fast in-memory key-value storage. In *Proc. 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Seattle, WA, Apr. 2014), USENIX Association, pp. 429–444.

[42] LIN, S.-C., ZHANG, Y., HSU, C.-H., SKACH, M., HAQUE, M. E., TANG, L., AND MARS, J. The architectural implications of autonomous driving: Constraints and acceleration. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (2018), pp. 751–766.

[43] MARTY, M., DE KRUIJF, M., ADRIAENS, J., ALFELD, C., BAUER, S., CONTAVALLI, C., DALTON, M., DUKKIPATI, N., EVANS, W. C., GRIBBLE, S., ET AL. Snap: a microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (2019), pp. 399–413.

[44] MCSHERRY, F. Graph analysis and hilbert space-filling curves. `https://bigdataatsvc.wordpress.com/2013/07/02/graph-analysis-and-hilbert-space-filling-curves/`, Jul 2013. Accessed: 2020-01-30.

[45] Microsoft Azure Functions. `https://azure.microsoft.com/en-us/services/functions/`.

[46] MONTAZERI, B., LI, Y., ALIZADEH, M., AND OUSTERHOUT, J. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. In *Proc. 2018 ACM SIGCOMM Conference* (New York, NY, USA, 2018), SIGCOMM '18, ACM.

[47] Dispersive Routing - Intel Omni-Path Fabric Performance Tuning, Aug 2020. `https://edc.intel.com/content/www/xl/es/design/products-and-solutions/networking-and-io/fabric-products/omni-path/intel-omni-path-performance-tuning-user-guide/dispersive-routing/`.

[48] OUSTERHOUT, A., FRIED, J., BEHRENS, J., BELAY, A., AND BALAKRISHNAN, H. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (Boston, MA, 2019), USENIX Association, pp. 361–378.

[49] 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. *Commun. ACM 54* (July 2011), 121–130.

[50] OUSTERHOUT, J., GOPALAN, A., GUPTA, A., KEJRIWAL, A., LEE, C., MONTAZERI, B., ONGARO, D., PARK, S. J., QIN, H., ROSENBLUM, M., RUMBLE, S., STUTSMAN, R., AND YANG, S. The RAMCloud Storage System. *ACM Trans. Comput. Syst. 33*, 3 (Aug. 2015), 7:1–7:55.

[51] PRABHAKAR, B. Time perimeters and stock exchanges in the cloud. `https://conferences.sigcomm.org/sigcomm/2020/tutorial-netfinance.html`, Aug 2020. Accessed: 2020-08-30.

[52] PREKAS, G., KOGIAS, M., AND BUGNION, E. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 325–341.

[53] PU, Q., VENKATARAMAN, S., AND STOICA, I. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (Boston, MA, Feb. 2019), USENIX Association, pp. 193–206.

[54] QIN, H., LI, Q., SPEISER, J., KRAFT, P., AND OUSTERHOUT, J. Arachne: Core-aware Thread Management. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2018), OSDI'18, USENIX Association, pp. 145–160.

[55] RASMUSSEN, A., PORTER, G., CONLEY, M., MADHYASTHA, H. V., MYSORE, R. N., PUCHER, A., AND VAHDAT, A. TritonSort: A Balanced Large-scale Sorting System. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2011), NSDI'11, USENIX Association, pp. 29–42.

[56] RUEFENACHT, M., BULL, M., AND BOOTH, S. Generalisation of recursive doubling for allreduce: Now with simulation. *Parallel Computing 69* (2017), 24–44.

[57] SHEN, M., ZHOU, Y., AND SINGH, C. Magnet: Push-based shuffle service for large-scale data processing. *Proc. VLDB Endow. 13*, 12 (2020), 3382–3395.

[58] SHI, H., AND SCHAEFFER, J. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing 14*, 4 (1992), 361 – 372.

[59] SHUN, J. *Shared-memory parallelism can be simple, fast, and scalable*. PUB7255 Association for Computing Machinery and Morgan & Claypool, 2017.

[60] SNIR, M., OTTO, S., HUSS-LEDERMAN, S., WALKER, D., AND DONGARRA, J. *MPI-The Complete Reference, Volume 1: The MPI Core*, 2nd. (revised) ed. MIT Press, Cambridge, MA, USA, 1998.

[61] Sort Benchmark Home Page. `http://sortbenchmark.org`.

[62] THAKUR, R., RABENSEIFNER, R., AND GROPP, W. Optimization of collective communication operations in mpich. *The International Journal of High Performance Computing Applications 19*, 1 (2005), 49–66.

[63] TIGANI, J., AND NAIDU, S. *Google BigQuery Analytics*. John Wiley & Sons, 2014.

[64] Broadcom's Trident 3 enhances ECMP with Dynamic Load Balancing, Sept 2017. `https://www.broadcom.com/blog/broadcom-s-trident-3-enhances-ecmp-with-dynamic-load-balancing`.

[65] WICKRAMASINGHE, U., AND LUMSDAINE, A. A Survey of Methods for Collective Communication Optimization and Tuning. *CoRR abs/1611.06334* (2016).

[66] ZAHARIA, M., DAS, T., LI, H., HUNTER, T., SHENKER, S., AND STOICA, I. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 423–438.

[67] ZAHARIA, M., XIN, R. S., WENDELL, P., DAS, T., ARMBRUST, M., DAVE, A., MENG, X., ROSEN, J., VENKATARAMAN, S., FRANKLIN, M. J., GHODSI, A., GONZALEZ, J., SHENKER, S., AND STOICA, I. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM 59*, 11 (Oct. 2016), 56–65.

[68] ZHANG, H., CHO, B., SEYFE, E., CHING, A., AND FREEDMAN, M. J. Riffle: optimized shuffle service for large-scale data analytics. In *Proceedings of the Thirteenth EuroSys Conference* (2018), pp. 1–15.

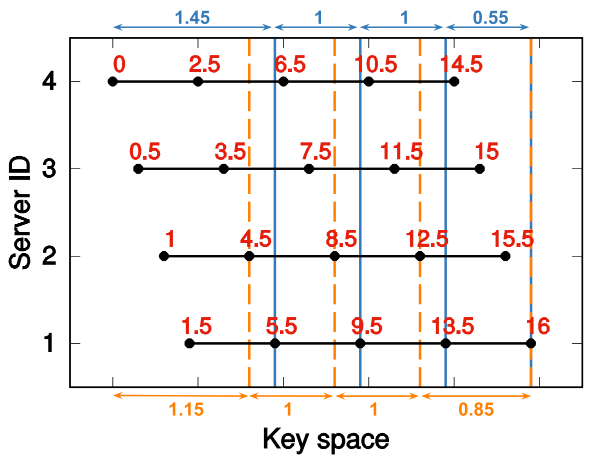


Figure 9: A comparison of the splitter selection algorithms on an example scenario. The keys of each server's data are uniformly distributed in a range indicated by a horizontal line, where dots are pivots (starting and ending pivots are not considered by the original algorithm). The splitters selected by the original algorithm are indicated with solid blue vertical lines. The splitters selected by the weighted algorithm are indicated with dashed vertical lines. The numbers on pivots are the cumulative pivot weights used by the weighted algorithm. The numbers on arrows above and below the diagram indicate the relative sizes of the buckets for the original and weighted algorithms, respectively.

A Appendix

A.1 Improved splitter selection

The splitter selection algorithm described in Section 4.2 tends to produce unbalanced partitions where the first server contains considerably more records than the last server. This imbalance can be explained by considering the groups of keys delimited by the pivots from each server; all of the groups contain about the same number of records. If we choose the Mth smallest pivot as the first splitter (assuming $s = 1$), then the first server will contain M full groups of records. In addition, it will contain some records from up to $M - 1$ additional groups, whose contents are divided between the first two servers (see Figure 9). Thus, the first server is likely to contain about $1.5M$ groups worth of records. In contrast, the last server will contain records from at most M groups, of which all but one are partial (some of their records have keys smaller than the last splitter). Thus, the last server will probably hold only about $0.5M$ groups of data.

To mitigate the data imbalance, we developed a new weighted approach to selecting splitters. The original approach behaved as if all of the keys in each group had the same value as the pivot at the end of the group. The new approach changes the weighting, so that half of the keys in each group are attributed to the beginning of the group and half to the end. Specifically, in the new approach, each server also includes its smallest and largest keys as pivots, and each pivot is annotated with a weight. The first and last pivots have a weight of 0.5 (half a group), and middle keys have

	120 servers (7000 records per server)							
	Uniform				Gaussian			
	Naive		Improved		Naive		Improved	
Pivots	P50	P90	P50	P90	P50	P90	P50	P90
120	33.6%	45.5%	6.9%	8.6%	37.6%	47.7%	6.9%	8.6%
240	23.5%	25.4%	4.4%	5.4%	23.4%	25.4%	4.4%	5.5%
360	15.9%	16.9%	3.1%	3.8%	15.9%	17.0%	3.2%	3.9%
480	11.5%	12.3%	2.5%	3.0%	11.6%	12.4%	2.5%	3.0%
600	9.6%	10.3%	2.0%	2.4%	9.7%	10.4%	2.0%	2.5%

Table 7: Excess records in the largest partition, relative to the average partition size, when using the improved splitter selection algorithm, vs. the naive algorithm. Input keys were drawn from two random distributions, and rows corresopnd to different numbers of pivots per server. "P50" and "P90" represent the median and 90th percentile over 1000 runs, respectively.

a weight of 1.0 (half of the preceding group and half of the following group). The annotated pivots from all servers are collected and sorted as usual. Then the pivots are scanned from smallest to largest, adding up the pivot weights; when a given pivot is reached, the cumulative weight is an estimate of how many groups worth of data have keys less than or equal to the pivot. The ith splitter will be the first pivot encountered where the cumulative weight is at least $i \times sM$.

We evaluated the improved mechanism for splitter selection by running stand-alone simulations with keys drawn from random distributions. Table 7 shows that the improved mechanism ensures that the largest partition is within 10% of the ideal size. Without the improvement, the largest partition will be 30-50% larger than ideal if the number of pivots per server equals the number of servers. Said another way, using the new splitter selection mechanism provides a greater benefit than increasing the pivots per server by 5x.

EPaxos Revisited

Sarah Tollman
Stanford University

Seo Jin Park
MIT CSAIL

John Ousterhout
Stanford University

Abstract

This paper re-evaluates the performance of the EPaxos consensus protocol for geo-replication and proposes an enhancement that uses synchronized clocks to reduce operation latency. The benchmarking approach used for the original EPaxos evaluation does not trigger or measure the full impact of conflict behavior on system performance. Our re-evaluation confirms the original claim that EPaxos provides optimal median commit latency in a WAN, but it shows much worse tail latency than previously reported (more than 4x worse than Multi-Paxos). Furthermore, performance is highly sensitive to application workloads, particularly at the tail.

In addition, we show how synchronized clocks can be used to reduce conflicts in geo-replication. By imposing intentional delays on message processing, we can achieve roughly in-order deliveries to multiple replicas. When applied to EPaxos, this technique reduced conflicts by at least 50% without introducing additional overhead, decreasing mean latency by up to 7.5%.

1 Introduction

Consensus-based replication protocols such as Raft [23] and Multi-Paxos [15] play an important role in large-scale datacenter applications. These protocols have traditionally required two round-trip times (RTTs) between machines in order to ensure durability and consistency before returning results to clients. The latency impact of these RTTs is particularly severe in geo-replicated applications, which have replicas in geographically distributed datacenters with inter-datacenter RTTs of 100ms or more.

In recent years there have been numerous proposals for new consensus protocols built around a *fast path* that can complete many operations with a single RTT [16, 21, 24, 30]. Operations that meet certain conditions, such as being commutative with other pending operations, can proceed along the fast path, while operations not meeting the conditions require at least two RTTs.

Unfortunately, most of these proposals, particularly those that exploit commutativity, work best within a datacenter, where RTTs are low [16, 24, 30]. High network delays in a wide-area network (WAN) make it more difficult to exploit commutativity because they increase the window of time during which operations can conflict. As conflicts increase, fewer operations can complete in 1 RTT, which reduces the latency benefit of the protocol.

Egalitarian Paxos (EPaxos) [21] is a commutativity-based consensus protocol that claims to have low latency in a WAN. As with other fast-path protocols, the performance of EPaxos is heavily dependent on workload, as the workload determines the proportion of operations that are able to benefit from the fast path. The original EPaxos evaluation included multiple scenarios for command interference, which plausibly measured best and worse case performance.

However, while replicating the EPaxos paper results, we discovered that the original benchmarking methodology did not trigger, and the metrics chosen did not capture, the full impact of conflicts on EPaxos performance. We reproduced the original EPaxos results, then broadened the evaluation to cover a wider range of potential conflict behavior. We also collected different metrics, which we believe more holistically reflect protocol performance.

Our re-evaluation draws different conclusions about EPaxos than the original evaluation. The original evaluation concluded that EPaxos has optimal commit latency in a WAN, where "optimal" is defined as 1 RTT to a simple majority of replicas. We agree that many operations can achieve this optimal latency, but we show that the percentage of operations completing in 1 RTT varies greatly depending on the workload. In one of our experiments, this percentage was as low as 65%. We also demonstrate that the latency increase for conflicting operations is substantial; even with a safe upper bound on execution latency many conflicting operations experience latency that is more than 4x worse than that of Multi-Paxos.

In the process of evaluating EPaxos more deeply, we discovered two opportunities for improving EPaxos' performance. When EPaxos runs on highly skewed workloads, dependency chains can grow without bound, resulting in tail latencies above 5 seconds or even livelock. We found a simple modification to EPaxos that prevents unbounded dependency growth and ensures a tight upper bound on latency. Second, we improved EPaxos performance using synchronized clocks. We noticed that conflicts occur primarily because different replicas process operations at different times. We modified EPaxos to use synchronized clocks to ensure that all quorum replicas process a given operation at the same time. This reduces conflict rates by at least 50% without introducing any additional latency.

This paper makes the following contributions:

- We show how that the original EPaxos evaluation does not fully capture the impact of conflicts on system performance. The EPaxos benchmarking framework has been used by subsequent papers such as Bipartisan Paxos [30], so we think it is important to publicize these limitations to prevent them from propagating further.
- We re-evaluate EPaxos with an improved benchmarking framework that captures a wider range of possible conflict behaviors. Our evaluation shows that the benefits of EPaxos are narrower and the bad behaviors significantly worse than previously reported. It is not safe to conclude that EPaxos will improve performance for a given applica-

tion without knowing details of the application's behavior and whether tail latency matters.

- We show how synchronized clocks can be used to reduce EPaxos conflict rate and increase its feasibility for a wider range of workloads.

The remainder of this paper is organized as follows: Section 2 provides an overview of the EPaxos protocol. Section 3 describes how we were able to provide an upper bound on EPaxos execution latency. Section 4 describes how we improved EPaxos by taking advantage of synchronized clocks. Section 5 details the differences between our EPaxos evaluation methodology and the original. Section 6 evaluates EPaxos and our improvements. Section 7 discusses related work, and Section 8 concludes.

2 Egalitarian Paxos Overview

EPaxos claims three benefits: low latency for geo-replication, high throughput, and graceful performance degradation in the face of failures. In this paper we focus on the latency benefit: EPaxos can complete most operations in a single wide-area RTT. Consensus protocols typically require two RTTs to complete an operation: one for the client to communicate with one of the replicas, and another to communicate information among the replicas. Most consensus protocols, such as Paxos [15] and Raft [23], have a single leader through which all operations must pass, so for most clients both of the RTTs for an operation will require wide-area communication. In contrast, EPaxos uses a leaderless approach, so clients can communicate with the nearest replica, typically in the same data center. As a result, only the second RTT requires wide-area communication. The cost of the RTT from the client to the local replica is negligible in terms of overall operation latency.

The challenge for leaderless protocols such as EPaxos is to establish a consistent ordering between operations submitted to different replicas. When *interfering* operations (those that are not commutative) arrive concurrently at different replicas, the replicas must somehow agree on an ordering, so that all replicas execute the operations in the same order. In the worst case, reaching agreement will require at least two WAN RTTs among the replicas. However, in many cases a single WAN RTT provides enough information to allow safe execution. For example, one RTT is sufficient to determine whether concurrent operations are commutative; if they are, then the order of execution does not matter. The performance benefit of leaderless approaches depends on how many operations can take the one-RTT *fast path* as opposed to the two-RTT *slow path*.

EPaxos uses *dependencies* to implement its fast path. When an *originating* replica receives an operation from a client, it issues a WAN RTT to collect dependency information from other replicas. The dependencies describe operations that interfere with the new one. If a quorum of replicas returns the same dependencies, then the operation will take the fast path: the originator will be able to analyze the dependencies, determine the ordering, and execute the operation without initiating any more RPCs (in the absence of crashes). If different replicas return different dependencies, then a second WAN RTT is required to propagate the union of the dependencies to the quorum. After the second RTT, the quorum is now in agreement, so the operation is durable.

In EPaxos, durability is defined in terms of dependencies: once a quorum of nodes has persisted an operation and its dependencies, the operation is guaranteed to execute eventually, even in the face of crashes. EPaxos uses the term *committed* to refer to an operation that is durable in this way. At the time an EPaxos operation commits, it has not been executed and its execution order may not even be known. Nonetheless, enough information has been persisted so that the operation can eventually be ordered and executed.

In the remainder of this section we will separately describe the EPaxos commit and execution protocols.

2.1 Commit Protocol

When a replica receives an operation from a client, it assigns that operation to an *instance*. Instances typically contain a single operation, but a replica may choose to batch multiple operations into a single instance. When batching occurs, the dependencies for the instance include all the dependencies for its constituent operations.

Each instance is assigned a globally unique *instance number* consisting of a unique identifier for the originating replica and a sequence number within that replica. Instance numbers provide an ordering among all the instances from a given originator.

In addition to assigning an instance number to each instance, the originating replica also proposes a *dependency set* and *sequence number* for the instance. The dependency set consists of the highest conflicting instance number from each replica (in addition, all preceding instances for the replica are also treated as dependencies). The initial sequence number is chosen as one greater than the largest sequence number in the dependency set. As will be seen below, the sequence number is used to order operations that are mutually dependent. The originating replica derives the dependency set and sequence number for a new instance from all other instances it has knowledge of. The durable storage for each object in the system must include the highest instance number (and sequence number) for each replica that has modified the object.

The first WAN RTT for an instance, called the *PreAccept* phase, begins with these suggestions from the originator. The originator sends the instance, along with its dependencies and sequence number, to a quorum of other replicas. Each replica confers with its state to provide its own suggestion for dependency set and sequence number for the instance, based on all of the instances it knows about. If the peer disagrees on the dependency set or sequence number for the instance, it will merge its determinations with those of the originator. The peer will reply with a dependency set that is the union of the one it received and the one it determined from its state, and a sequence number that is the maximum of the two. The peer persists this information locally and will consider it when

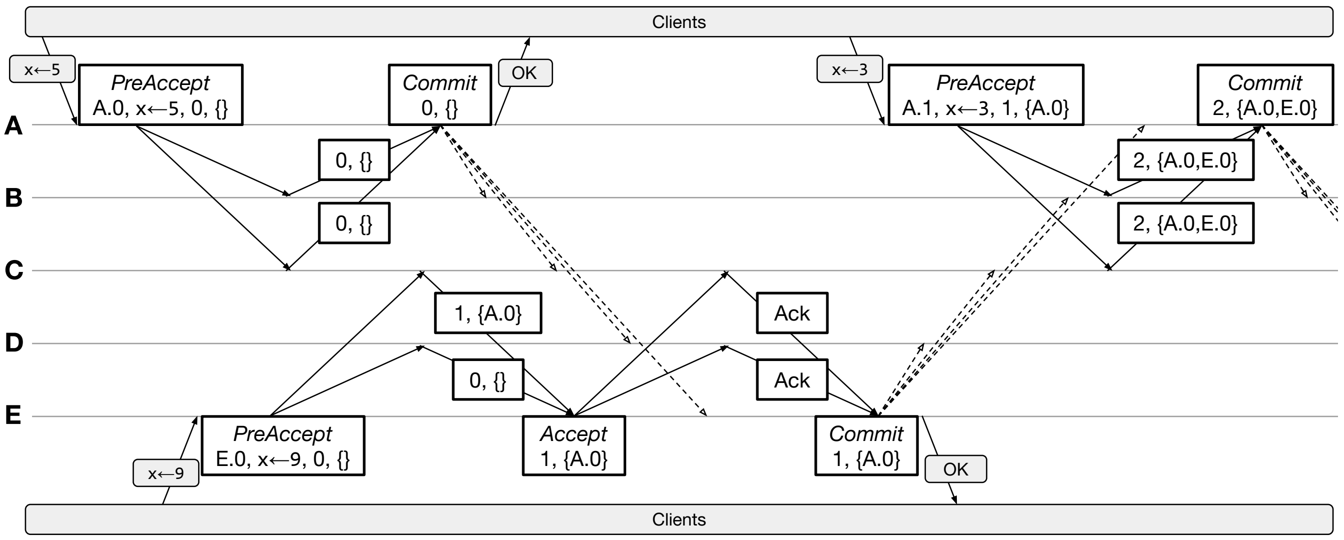


Figure 1: EPaxos commit protocol examples. A, B, C, D, and E are replicas in a key-value store. PreAccept boxes contain the instance number, operation, sequence number, and dependency set for the instance. Subsequent inter-replica messages contain only the sequence number and dependencies. For simplicity, we only show PreAccept and Accept messages to the necessary quorum, and commit messages are indicated by dotted arrows. This figure is adapted from the EPaxos SOSP presentation [18].

processing future PreAccept requests; in replying, the replica promises to reserve that sequence number and dependency set for that instance, until it hears otherwise from the originator.

If the originator receives identical responses from a quorum of peers, it can commit on the fast path. When this happens, it considers the instance committed; it sends Commit messages asynchronously to all other replicas and proceeds to the execution phase discussed in Section 2.2 below. For example, in Figure 1, a quorum of replicas A, B, and C agree on the sequence number and dependencies for instance A.0, so A.0 can commit on the fast path.

If the originator receives conflicting responses, it must issue a second round of RPCs for the instance, called the *Accept* phase. In the Accept phase, the originator merges the sequence numbers and dependency sets from the fast-path quorum's PreAccept replies and sends those to other peers. Because sequence numbers and dependency sets are strictly increasing, the Accept phase will always succeed; each of the recipients of an Accept request has already reserved a lower sequence number and smaller dependency set for the instance, corresponding to an earlier execution order, so it can surely accept higher ones, corresponding to a later execution order. In Figure 1, the PreAccept replies from C and D for E.0 differ, so E merges the responses and commits E.0 after the dependencies and sequence numbers have been replicated to the quorum.

An instance can still commit on the fast path if its originator disagrees with the PreAccept replies, so long as the replies agree with one another. Because sequence numbers and dependencies can only increase, the originator can increase its values for those properties to match the responses, achieving quorum agreement without further communication with other replicas. In Figure 1, instance A.1 can commit on the fast path because

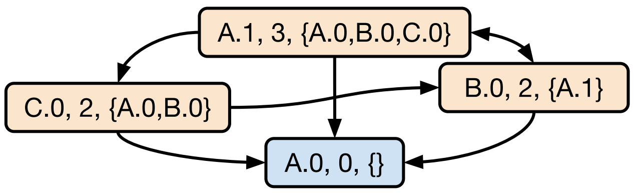


Figure 2: EPaxos execution algorithm example. Each node represents an instance, annotated with its instance number, sequence number, and dependency set. There are directed edges from each node to its dependencies. The nodes are colored by strongly connected component. Traversing the graph yields the execution order A.0, B.0, C.0, A.1.

B and C agree on its sequence number and dependencies. Although A did not initially agree with B and C, A increases its sequence number and dependencies to those agreed on by B and C in order to commit on the fast path.

EPaxos' fast-path quorum size is optimal for common 3- and 5-node cluster sizes. The EPaxos fast-path quorum size is $F + \lfloor \frac{F+1}{2} \rfloor$ replicas, where F is the number of tolerated failures and there are $N = 2F+1$ total replicas. For common 3 and 5 node cluster sizes, EPaxos' fast-path quorum is a simple majority. This is an improvement over the fast-path quorum of prior fast-path consensus protocols, such as Generalized Paxos [16], which has a fast-path quorum size that is always one node larger than that of EPaxos. The EPaxos slow-path quorum size is always a simple majority. EPaxos' quorum sizes are made possible by its crash recovery mechanism. The EPaxos crash recovery protocol is not relevant to the performance-oriented discussions in this paper, and is detailed in the EPaxos paper.

2.2 Execution Protocol

An instance's execution order is determined by creating and traversing a graph of its dependencies. This implies that an in-

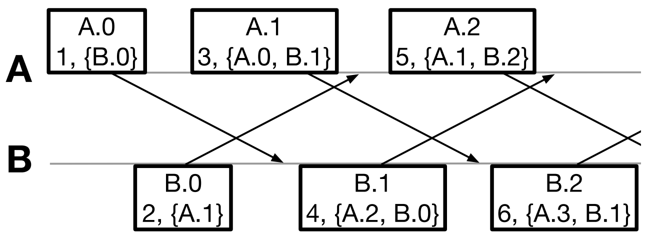


Figure 3: Example of EPaxos infinite dependency chains. Replicas A and B originate instances on the same hot key. In this example, there are three replicas in the system, but A and B are closer together and use each other as their quorum. The arrows indicate PreAccept messages and their timing. Each box contains the instance number, final sequence number, and final dependency set for the instance.

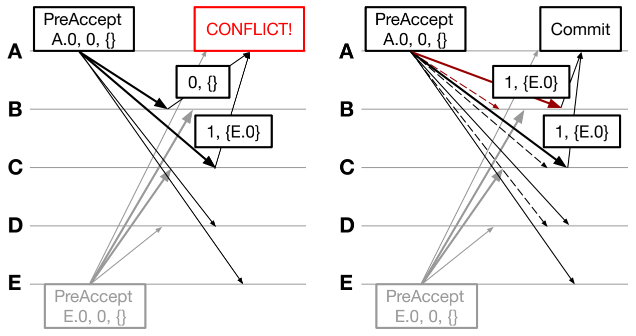
stance must wait for all of its transitive dependencies to commit before executing. In the dependency graph, nodes are instances and directed edges point from instances to their dependencies. As discussed in Section 2.1, if an instance is dependent on R.*n*, it is implicitly dependent on R.0 through R.(*n*-1), and R.0 through R.(*n*-1) would be added as nodes in the graph. The graph can be pruned to contain only those instances that have not executed and whose operations actually interfere with those of the current instance. The graph contains not just an instance's direct dependencies, but also the transitive closure of dependencies. In order to create this graph, and before it can be traversed, all transitive dependencies must have committed so that their sequence numbers and dependency sets are final.

Once an instance's dependency graph is final, it can be traversed to determine execution order. First, the graph is separated into strongly connected components – groups of nodes within which there is a path from each node to every other node. The strongly connected components are executed in reverse topological order. Within a component, instances are executed in order of increasing sequence number, with ties broken in a deterministic manner (e.g. by replica id). Figure 2 provides an example of the execution algorithm. In the example, A.0 forms its own strongly connected component, since it does not have any dependencies. Instances A.1, B.0, and C.0 form another strongly connected component. By reverse topological order, the strongly connected component containing A.0 will be executed first. Within the remaining strongly connected component, the nodes are executed in order of increasing sequence number, breaking the tie between B.0 and C.0 by replica id.

3 Bounding Dependency Chains

Execution latency is unbounded in the original EPaxos protocol because dependencies may chain recursively and an instance cannot execute until all of its transitive dependencies have committed. The EPaxos paper discussed this problem, but the problem did not occur with the workloads used for the original evaluation.

An instance's transitive dependency set can continually grow if new interfering instances are originated before all replicas become aware of prior instances. If this happens, the instance will wait on more and more recently added transitive dependencies to commit, and will be unable to execute. Figure 3 provides an example of this. In the example, a pair of replicas, each of which is in the other's quorum, continually originates new instances on some hot key. Although A.0 is originated first, B originates B.0 before B becomes aware of A.0. For this reason, A.0 becomes dependent on B.0. Similarly, B.0 becomes dependent on A.1, A.1 becomes dependent on B.1, B.1 becomes dependent on A.2, and A.2 becomes dependent on B.2. A.0 is now transitively dependent on B.2. With this behavior, this pattern could continue infinitely, causing execution livelock in the worst case. In practice, this problem causes high execution latency for skewed workloads, as demonstrated in Section 6.4.



(a) Unmodified EPaxos **(b)** With TOQ

Figure 4: EPaxos PreAccept phase example, with and without TOQ. Replicas A and E originate interfering instances A.0 and E.0 concurrently. Without TOQ, the responses from B and C differ for A.0. With TOQ, replica C delays processing the PreAccept for A.0 until the time that C processes it, avoiding conflict.

We are able to bound execution latency by pruning those transitive dependencies that will necessarily be executed after the current instance. Given two instances A and B, the EPaxos execution algorithm states that if B has A as a dependency and B has a higher sequence number than A, then B is serialized after A. The proof of Theorem 5 in the EPaxos proof of correctness [19] proves that, given this serialization, B will necessarily execute after A. Therefore, B can be pruned from A's execution graph. Any new interfering transitive dependency C of B will also necessarily execute after A, because C has a higher sequence number than A and has A in its dependency set, so it can be pruned as well. For example, in Figure 3, B.0 has a higher sequence number than A.0 and has A.0 as a dependency, so B.0 and its yet unknown transitive dependencies (A.1, and A.1's transitive dependencies) can be pruned from A.0's execution graph. With our modification, execution delay is bounded by approximately 3 WAN RTTs. Appendix A.1 describes the bound in more detail.

4 Using Synchronized Clocks to Reduce Conflict Rate

In this section we present a novel approach that reduces conflicts in EPaxos by taking advantage of synchronized clocks. We call this approach Timestamp-Ordered Queueing, or TOQ.

The benefits of EPaxos, and any commutativity-exploiting consensus protocol, depend on minimizing interference between non-commutative operations. By synchronizing the physical clocks of all replicas in the cluster, we are able to order instances so that all replicas process them in the same order, which can eliminate interference even for non-commutative operations. Our approach to ordering is an enhancement; out of order messages or poorly synchronized clocks may result in less conflict mitigation, but do not hinder correctness. We implemented our approach on EPaxos, but we believe it can be generally applied to any fast-path consensus protocol. (We leave this for future work.)

Conflicts occur when replicas learn about instances in different orders. In EPaxos, replicas first learn about an instance they did not originate through its PreAccept message. When the order in which a fast-path quorum processes interfering PreAccept messages differs, the quorum disagrees on the dependency information for those instances, causing them to take the slow path. Figure 4a illustrates an example of this. In the figure, replica A sends a PreAccept message for A.0 at the same time as replica E sends a PreAccept message for E.0. In the example, the fast-path quorum for instances originated from A consists of B and C. Replica B processes the PreAccept messages in the order (A.0, E.0), while replica C processes them in the order (E.0, A.0). Because of this difference in ordering, B and C respond to A's PreAccept with different sequence numbers and dependency sets, causing A.0 to take the slow path.

In the original EPaxos protocol, the order in which a replica learns about instances is determined by the one-way delay between the replica and the originators of the instances. In Figure 4a, the delay between A and B is shorter than that between E and B, and the delay between E and C is shorter than that between A and C. The difference in the one-way delays from the originator to each replica in the fast-path quorum determines whether an instance is ordered before or after instances that interfere with it. By the time a PreAccept message is processed by the furthest replica from its originator, the responses from the closer replicas are stale.

To reduce conflict rate, closer replicas can delay the time at which they process a PreAccept message until the time at which the furthest replica would process it. In Figure 4a, by the time C processes A.0, B and C have both processed E.0. If B were to respond at the same time as C, B and C would respond with the same dependency information. In Figure 4b, replicas delay processing PreAccepts until the time at which the furthest replica processes them. If B delays processing A.0 until the time that C processes it, both B and C have seen E.0 before A.0, and A.0 can commit on the fast path.

4.1 Design

Because replicas cannot intuit the time at which the furthest replica from the originator will process a PreAccept message, each PreAccept message includes a *ProcessAt* timestamp. Each replica will estimate the one-way delay from itself to all other replicas. The ProcessAt time will be the sum of the time at which the PreAccept is sent and the one-way delay from the originator to the furthest replica from the originator.

Each replica sorts PreAccepts in a priority queue based on the ProcessAt time of the PreAccept. Highest priority is given to lower ProcessAt times. An element is dequeued and processed once its ProcessAt time has passed. Because each replica receives the same ProcessAt time for a given instance, if PreAccepts are processed in order of their ProcessAt times then conflicts should not occur.

In order for instances to be totally ordered, a replica must also delay assigning dependency information to instances that it originates. To illustrate why this is necessary, we will look at an example of how conflict can occur when replicas do not delay their own instances. Hypothetically, if replicas A and B originate interfering instances A.0 and B.0 concurrently. A would process the instances in the order (A.0, B.0), and B would process the instances in the order (B.0, A.0). Instances will be processed out of order if a replica does not delay those instances it originates.

To delay the time at which a replica assigns dependency information to its own instances, we leverage the fact that the originator of an instance need not agree with the replies to its PreAccept messages, so long as the replies are unanimous. With TOQ, all PreAccept messages are sent out with sequence number 0 and empty dependencies. The originator adds its own PreAccept messages to its PreAccept priority queue and processes them at their ProcessAt time, consistent with the time they should be processed by the other replicas in the cluster. This means that an originator will not add its instances as dependencies of other instances until other replicas would do the same.

To maintain correctness, we needed to slightly modify the commit protocol for EPaxos. In unmodified EPaxos, an instance can commit on the fast path if the PreAccept replies are unanimous, even if the originator disagrees. With TOQ, since the originator modifies the dependency information for its own instances after it sends the PreAccept messages for them, it is possible that the originator would assign a greater sequence number or dependencies than would its quorum, even if the quorum agrees with one another. Instead of allowing the originator to disagree with the rest of the quorum no matter what, the replica can now disagree with the quorum only if the dependency information of the quorum is greater than that of the originator. Lemma 1 of the EPaxos proof of correctness [19], "EPaxos replicas commit only safe tuples," depends on a fast-path quorum recording the same tuple for any instance that commits in the PreAccept phase. With this modification, we ensure that this holds true.

4.2 Sync Groups

In practice, there is a tradeoff between reducing conflict rate and increasing the minimum latency for each instance. So far, we have described TOQ with the approach of synchronizing all PreAccept messages to all replicas in the cluster. To do this, the ProcessAt time for each PreAccept is the estimated time the PreAccept would be processed at the furthest replica. However,

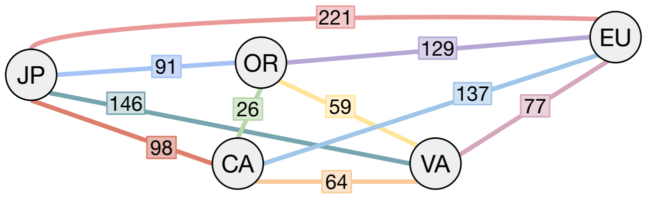


Figure 5: Round-trip times (in ms) between replicas in different locations in our test topology, as measured by the `ping` application. Replicas are located in Virginia (VA), California (CA), Oregon (OR), Japan (JP), and London (EU).

this increases the minimum possible latency for an instance. To illustrate this, we can use back-of-the-envelope calculations based on the RTTs from our test cluster, shown in Figure 5. The figure contains RTTs, and we estimate one-way delay to be half of RTT. Without PreAccept ordering, the minimum latency for an instance that originated in Japan would be 98ms, which is the RTT to the furthest replica in its quorum, California. With PreAccept ordering to the whole cluster, California would delay processing a PreAccept from Japan by the difference in the one-way delay between Japan and California and Japan and London, which is 61.5ms. This increases the minimum latency for an instance originated in Japan by more than 50%.

Another option would be to synchronize message ordering to the furthest replica in each originator's quorum. This approach has the same minimum latency as does unmodified EPaxos. However, syncing to just the quorum is less effective when it comes to reducing conflict rate. Each originator's quorum handles all of the originator's messages in the same order, but may handle messages originating from other replicas in different orders, leading to discrepencies in the dependency information in PreAccept replies.

A hybrid approach is to synchronize message ordering among a quorum *union*. This quorum union contains all replicas included in another node's fast-path quorum. With this sync group, conflicts should occur only when the originator assigned greater dependency information to an instance than did the remainder of its quorum. The benefits of this approach differ greatly depending on the topology; for some topologies, the quorum union might contain the entire cluster.

5 Capturing the Impact of Conflicts

The evaluation approach in the EPaxos paper used workloads that did not trigger the full range of conflict behaviors, and it chose metrics that did not measure the full impact of conflicts. As a result, its conclusions about performance are over-optimistic. In this section we discuss the implications of the original EPaxos evaluation and introduce the changes we made in our evaluation.

5.1 Commit vs. Execution

The EPaxos paper measures latency in terms of commit time, not execution time. Specifically, latency is measured as the time from initiation of an operation on a client until receipt of a commit acknowledgment from the local replica. This approach was chosen because a commit guarantees that an operation is durable and will eventually execute; thus clients need not wait for execution. However, this argument applies only to operations that return no results or errors, such as blind writes (blind writes are the only operation implemented in the EPaxos benchmarks). Commit latency is not globally sufficient; if a client depends on information returned by an operation, then it must wait for execution. For example, the write operations in both Zookeeper [1] and Redis [2] return results. Furthermore, if an operation can result in an error (such as attempting a bank account withdrawal without sufficient funds, or attempting to set a value in a table that has been dropped) then the client must wait for execution to complete. Furthermore, all reads must wait for preceding conflicting writes to execute. Thus, we think it is likely that most operations will require clients to wait for execution, and we argue that EPaxos latency should be measured with execution time, not commit time.

5.2 Access Patterns

One of the challenges in evaluating a protocol like EPaxos is that its performance depends on the application workload. The EPaxos benchmarks use a key-value store with an access pattern based on a single hot key. Each reference is made either to the hot key, or to a unique key that is never referenced by any other operation. Conflict rates are controlled by adjusting the fraction of references to the hot key (either 0%, 2%, or 100% in the EPaxos paper latency evaluation). This approach has two limitations. First, having all interference on a single key understates the impact of conflicts, since there can never be more than a single strongly-connected component in the dependency graph. Second, the EPaxos evaluation conflated the fraction of references to the hot key with the rate of conflicts in the EPaxos commit protocol; because conflict rate is a product of many additional factors, such as system load, the actual conflict rate is considerably less than the fraction of references to the hot key. For example, the workload labeled '100% interference', which was claimed to be worst case, produced actual conflict rates as low as 0%.

For our evaluation we used a Zipfian distribution [11], which is generally accepted as an approximation of real workloads. For example, both Memcached [6] and Facebook's social graph database [4] experience highly skewed workloads that approximate Zipfian. No single value of the Zipfian skew parameter is representative of all applications, so we made measurements with multiple values ranging from lightly skewed (.6) to highly skewed (.99). This covers the ranges used by other benchmarks such as Linkbench [3] and YCSB [12] and allows us to observe how application characteristics affect EPaxos performance.

We also used a mix of read and write operations in our evaluation, whereas the EPaxos paper used only writes. Read operations have different behavior from writes: they do not interfere with other reads, but they do interfere with writes. Thus the mix of reads and writes affects the conflict rate.

5.3 Load

The approach to load generation in the EPaxos paper minimizes opportunities for command interference, which results in over-optimistic conclusions about latency. First, the paper measures latency only at low system load; higher loads will result in more conflicts, which will increase latency. Furthermore, load is generated with a collection of closed-loop clients, each of which issues a continuous stream of back-to-back operations, with exactly one operation outstanding at all times. Load is controlled by varying the number of clients. We found that this produced a caravan effect, where large groups of operations passed through a replica at once, followed by gaps with no operations. This behavior is unlikely to occur in real applications, and results in an unvarying system load, while real systems are likely to experience bursts. Even systems that use server proxies for admission control still experience variation in load when they are not at maximum capacity. Finally, back-to-back operations result in uneven loading across clients: clients that are more centrally located have higher throughput than those further away, since throughput is inversely proportional to RTT with this approach.

For our measurements we used higher loads (typically 80% of the maximum sustainable load) and we used a Poisson process to schedule new operations, which results in uneven loading that better approximates real workloads. A Poisson arrival process is open loop, which means that care must be taken to choose an arrival rate that does not exceed the system's capacity. If a Poisson arrival process results in system overload, queues will grow without bound, the system does not reach a steady state, and performance measurements are unlikely to be meaningful. We monitored queue lengths in all of our experiments and chose arrival rates for which queue lengths remained bounded. In addition, we added a cap on the total number of outstanding operations for each client (the cap was never reached in our experiments).

5.4 Throughput

There are several confounding factors in the EPaxos throughput measurements, which make it difficult to determine whether the results reflect fundamental properties of the protocols or artifacts of the experimental setup:

- The throughput experiments were run in a LAN, not a WAN. WAN configurations are likely to have higher conflict rates because longer network delays increase the window for conflicts, leading to lower throughput.
- There were different approaches to generating load for different experiments, which led to inconsistent results. With one method, EPaxos '100%' has 12% higher throughput than Multi-Paxos, and with the other it has 27% higher throughput.
- The evaluation of batching (many operations per instance) does not consider the impact of batching on WAN latency.
- The evaluation only considers perfectly even load balancing across clients. Throughput would likely decrease if clients in different locations issued operations at different rates. Correcting for this with load balancing amongst the replicas would negate the latency benefits of EPaxos.
- Optimizations to Multi-Paxos were not considered. For example, the Multi-Paxos leader can hold a read lease for all objects.

5.5 Summary

Evaluation of EPaxos in Section 6 differs from the original EPaxos evaluation in the following respects:

- We measure execution latency, instead of commit latency.
- We use a Zipfian distribution for choosing keys to access, instead of a single hot key.
- We use a mixture of read and write operations, instead of all writes.
- We measure latency at about 80% of maximum throughput, instead of a lightly-loaded system.
- We use Poisson arrival rates, instead of back-to-back operations.
- We measure throughput in the same configuration used for latency measurements, instead of using a LAN configuration with batching.
- We measure a large number of different configurations with different write rates and Zipfian skew factors, instead of 3 hot fractions.

6 Evaluation

We had two goals in our EPaxos evaluation. First, we wanted to see whether our changes to the EPaxos evaluation would change the conclusions about EPaxos performance relative to Multi-Paxos. Our new evaluation confirms the original conclusion that many operations can achieve optimal latency. However, only operations that experience no interference can achieve optimal latency, and we demonstrate that conflict rate varies greatly by application (in our experiments, we measured conflict rates as high as 35%). Additionally, we show that conflicting operations experience significantly higher than optimal latency (more than 4x that of Multi-Paxos, even with our improvements).

Our second goal was to evaluate the effectiveness of TOQ. We found that TOQ is effective at reducing conflict rate, reducing EPaxos latency.

6.1 Experimental Setup

When evaluating EPaxos, we aimed to use an experimental setup as similar to that of the authors as possible. Our analysis is a fork of the EPaxos paper evaluation [20] and is available on GitHub [28]. We used Google Cloud servers in Virginia (VA), California (CA), Oregon (OR), Japan (JP), and London (EU). We used the `n1-standard-8` machine type with the Ubuntu 18.04.5 LTS operating system.

Unless otherwise specified, all experiments mitigate dependency chaining using the approach described in Section 3. EPaxos and Multi-Paxos replicate all operations, including reads. Workload does not impact Multi-Paxos, so we used a workload with a Zipfian skew factor of .9, 50% writes, and 1 million unique keys. Since any reference point is sufficient to see how conclusions change, we compare EPaxos against only Multi-Paxos.

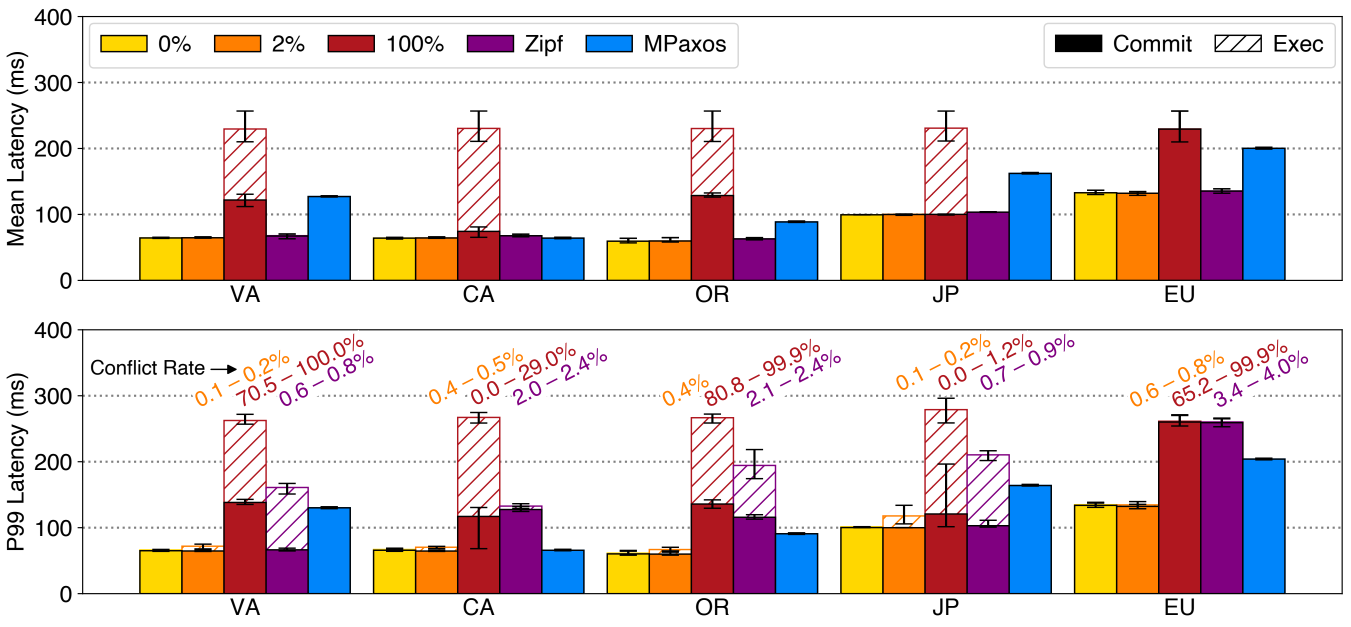


Figure 6: Comparing client-perceived commit and execution latency for EPaxos across a variety of workloads at each of the five sites. 10 clients at each site issue operations, all of which are writes, with outstanding operation at a time. This leads to a throughput of 500 operations per second for Multi-Paxos, 230 operations per second for '100%' and 650 operations per second for the '0%', '2%', and Zipfian workloads. All Multi-Paxos clients issue operations to the leader, which is in California; EPaxos clients issue operations to the replica in the same datacenter. The Zipfian distribution has a skew parameter of .9, 50% writes, and 1 million unique keys. We ran each experiment 5 times, capturing metrics for 60 seconds at steady-state. We show the average latency across the 5 trials, with error bars indicating the minimum and maximum latency of the trials. The range of conflict rates from the trials, measured on the servers, appears above the bar corresponding to the experiment.

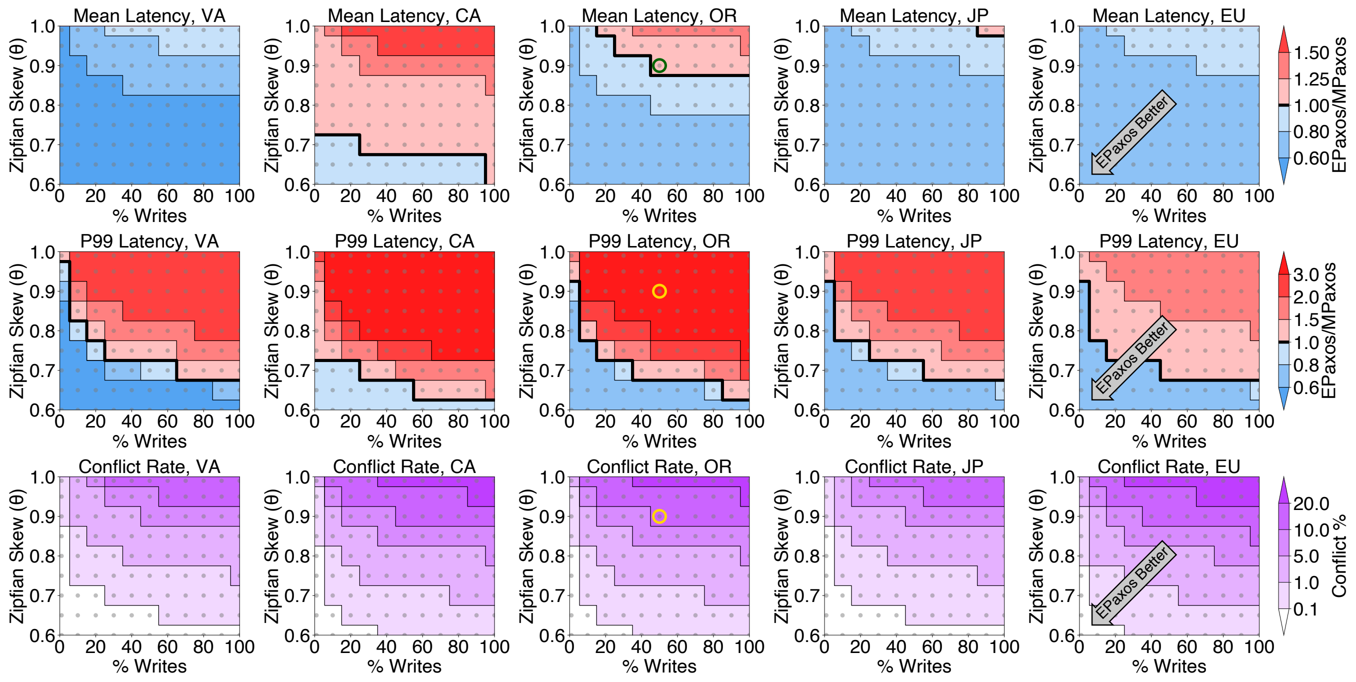


Figure 7: Comparing EPaxos and Multi-Paxos execution latency across a spectrum of Zipfian skew parameters and read-write ratios. Each grey dot corresponds to a workload on which EPaxos was tested. We divide the latency for each EPaxos experiment by Multi-Paxos' latency. Different fill colors represent different ranges of performance; for example, the darkest red color in the top graphs indicates that EPaxos mean latency was at least 1.5x that of Multi-Paxos. All workloads have 1 million unique keys, and all experiments were run at a throughput of 11,000 operations per second, which is 80% of the maximum throughput of EPaxos with a skew parameter of .99 and 100% writes. The circled point indicates the workload used for Figure 8.

We initially ran our experiments for both 3 and 5 replicas, but decided to only include measurements for 5 replicas because performance degraded similarly for both cluster sizes as we modified the experimental methodology. TOQ does not provide a benefit for 3 replicas because conflicts do not occur with a quorum size of 2 (the originator can simply agree with the first replica to respond).

We do not enable *thrifty* – a mode of operation in which PreAccept and Accept messages are only sent to the necessary quorum. We chose to disable thrifty because we wanted our results to be as favorable to EPaxos as possible. As detailed in Appendix A.2, thrifty increases EPaxos conflict rates, which makes latency worse. Thrifty provides the same throughput benefit for EPaxos and Multi-Paxos (about 15%), so enabling thrifty does not impact conclusions about throughput. Furthermore, TOQ is not compatible with thrifty. While decreases in conflict rate due to TOQ are much larger than those from disabling thrifty, we wanted it to be clear in our comparison that any decrease in conflict rate could be attributed to TOQ rather than from disabling thrifty.

6.2 Execution Latency

Our first experiment reproduces the basic commit latency results from the EPaxos paper (see Figure 6); we used the configuration from the EPaxos paper, without any of the changes described in Section 5. We report mean latency instead of median for reasons that will be discussed below. The results are very similar to those in Figure 4 of the EPaxos paper, except that we measured a lower 99th percentile latency for the ‘2%’ workload. This difference does not impact any of our conclusions.

Figure 6 has three additional measurements not present in the EPaxos paper. First, it includes execution latency as well as commit latency. When conflict rates are low, execution latency is indistinguishable from commit latency. However, with high conflict rates, the execution latency can be more than 2x the commit latency. For the pathological ‘100%’ workload, EPaxos execution latency is considerably worse than Multi-Paxos at every site, both at the mean and the 99th percentile. Second, we also measured a Zipfian access pattern; even at low load (about 4% of maximum possible throughput for this experiment), 99th percentile latencies for the Zipfian workload are significantly higher than for the ‘2%’ workload. 99th percentile execution latency under the Zipfian workload is worse than Multi-Paxos at every site. Third, we measured conflict rate on the servers. Primarily due to low load, conflict rates for the ‘2%’ workload were as low as 0.1% and conflict rates for the ‘100%’ workload were as low as 0%.

For the remaining experiments, we applied the evaluation changes discussed in Section 5. One of the challenges in benchmarking EPaxos is that performance varies drastically depending on the workload. It is not difficult to find arguably realistic workloads indicating that EPaxos is either very good or very bad. Thus, we don’t believe it is possible to evaluate EPaxos fairly with a few discrete workloads. Instead, we ran each of the remaining experiments over a large range of workloads, typically a full grid with 9 choices of Zipfian skew and 10 choices of read-write ratio. We then plot the data to show how certain system properties vary across the workloads. A potential EPaxos user can use these plots to determine whether EPaxos would provide performance benefits for their particular application. We tried to vary the workload enough to cover the full range of conflict behaviors likely to be experienced by real applications. The experiments were all run at 80% of the maximum throughput of the workload with the lowest maximum throughput. A consequence of keeping throughput consistent for all experiments is that the system is underloaded for many of the workloads. Appendix A.3 contains measurements showing that conflict rate and latency both increase as throughput increases.

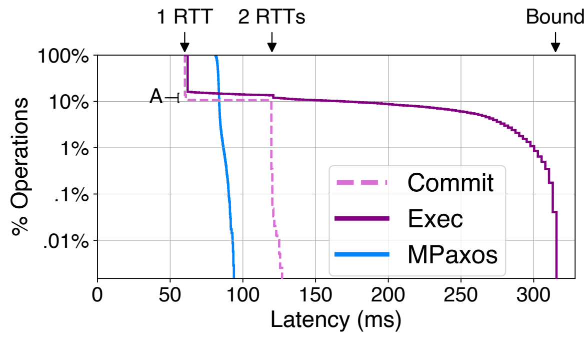


Figure 8: Complementary CDF of execution and commit latency. A point (*x*,*y*) indicates that a fraction *y* of instances had latency greater than *x*. The data was measured on clients at Oregon on a workload with 50% writes, Zipfian skew 0.9, and a load of 11,000 operations/sec. ‘A’ indicates the area of instances that experience execution delays despite taking the fast path.

Figure 7 compares the execution latency of EPaxos and Multi-Paxos across the grid of workloads. EPaxos has better mean latency for almost all of the workloads (except for clients in California, which can reach the Multi-Paxos leader without a WAN RTT). On the other hand, EPaxos has worse 99th percentile latency for almost all workloads. EPaxos cannot offer lower latency than Multi-Paxos if there is a conflict in the PreAccept phase, since this results in a second WAN RTT. For EPaxos P99 latency to be better than Multi-Paxos, less than 1% of operations must encounter conflicts; the bottom graphs in Figure 7 show that few workloads have this behavior. In addition to extra RTTs caused by conflicts, EPaxos can also incur execution delays while waiting for dependent operations to commit; Muilti-Paxos does not incur these delays.

Figure 8 exposes several interesting aspects of commit and execution latency:

- Most instances finish execution in one WAN RTT.
- The median latency for both commit and execution will be one RTT unless conflicts approach 50%, which never happened in our experiments. Thus, median latency reflects only network topology, not protocol response to workload.
- Instances that take more than one RTT for execution trend rapidly towards the worst-case delay for dependencies to commit. (‘Bound’ in the figure.)
- Execution latency is rarely in the vicinity of two RTTs:

if there is a conflict, there will probably be a significant dependency delay.

- A significant fraction of instances that take the fast path experience dependency delay. (This is indicated by the area between the lines at point 'A' in the figure.)
- The worst case performance penalty from EPaxos is greater than the best case performance gain; EPaxos fast path latency is 30% better than Multi-Paxos latency, while the slow path ranges from 40% to 240% worse.

Conclusions about EPaxos latency are almost entirely determined by the choice of metric. The original evaluation used median commit latency as the primary metric. This is the most favorable choice for EPaxos: median latency for EPaxos is independent of application behavior, identical for commit and execution, and uniformly better than Multi-Paxos. Tail latency is often used as a metric in both industry and academia; 99th percentile tail latency is also almost independent of application behavior for EPaxos, and it is significantly worse than Multi-Paxos, especially for execution. Workload differences tend to manifest themselves primarily in the range between the 50th and 99th percentiles, where most metrics will overlook them. We chose to use mean latency instead of median in our graphs, since it reflects some of these workload differences.

6.3 Benefits of TOQ

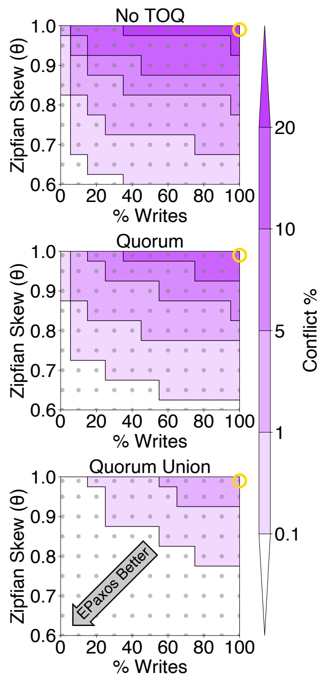
In order to evaluate TOQ we installed the Huygens [10] clock synchronization software to synchronize the clocks of all replicas in the cluster. This resulted in server clock offsets between 20µs and 1ms. The original Huygens evaluation measured clock offsets within 10µs; we believe that we experienced higher clock offsets because of differences in our environments (WAN vs. LAN and virtual machines vs. bare metal).

Figure 9 shows the benefits of TOQ. Quorum sync reduces conflict rates by at least 50% across locations. This reduces mean latencies, which improves the benefits of EPaxos relative to Multi-Paxos. Quorum sync has the advantage of not impacting minimum latency (latency is already limited by the most distant peer in the quorum; quorum sync delays the other peers in the quorum so they process PreAccept messages at the same time as the most distant peer).

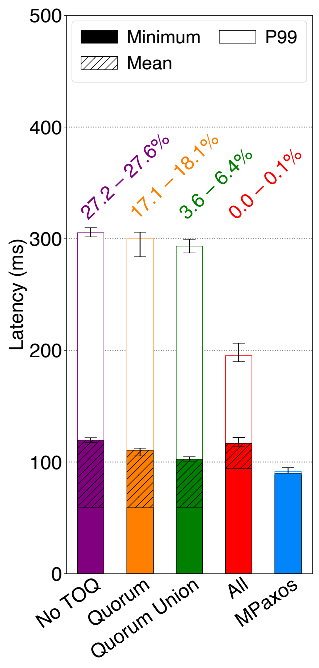
Quorum union sync introduces a performance tradeoff. It reduces conflicts by at least 85%, which improves worst-case latency. However, it increases minimum latency. For example, quorum union sync delays Japan's PreAccept messages until they would reach Virginia, which adds about 25ms to the minimum latency for operations issued by clients in Japan. This does not impact the minimum latency for Oregon in Figure 9b, but the impact on Japan can be seen in Figure 20 in the Appendix. Syncing to all replicas completely eliminates conflict, but increases minimum latencies so much that it makes this option unattractive.

6.4 Unbounded Dependency Chains

Our focus on execution latency, combined with Poisson arrivals, exposed the problem of unbounded dependency chains



(a) Conflict Rate



(b) Latency

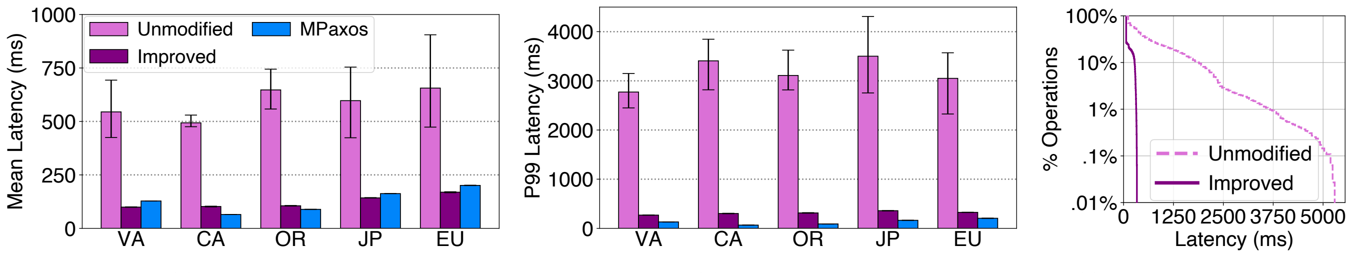
Figure 9: Conflict rate and execution latency for EPaxos with and without TOQ. Figure (a) compares conflict rates for different TOQ sync groups. Figure (b) compares Mean and P99 execution latency for the circled workload (Zipfian skew of .99 and 100% writes). 'Minimum' indicates the minimum possible latency for an operation given the protocol and sync group. Other than TOQ, the experiments for this graph were run with the same methodology as Figure 7. Error bars and conflict rates are shown as in Figure 6. The quorum union consists of Virginia, California, and Oregon. For measurements of sites other than OR, as well as the latency graph for a workload with Zipfian Skew .8 and 50% writes, see Appendix A.4.

discussed in Section 3. Figure 10 shows that EPaxos execution latency can be much higher than Multi-Paxos when it does not protect against infinite dependency chains. Before we implemented our optimization, we observed execution latencies as high as 5 seconds under high-skew workloads. Livelock occurred with the pathological '100% conflict' workload when we removed the cap on outstanding operations from our Poisson scheduler (the cap eventually prevents new operations from entering the system, so the existing operations can complete).

Figure 10c shows that our fix reduces tail latency significantly. The benefits are most noticeable for write-heavy workloads that are highly skewed. However, workloads that generally experience low conflict can still experience bursts of chained dependencies, leading to latency spikes that are surprising and difficult to diagnose. Section 6.6 below shows that that dependency chain optimization also has a significant impact on throughput.

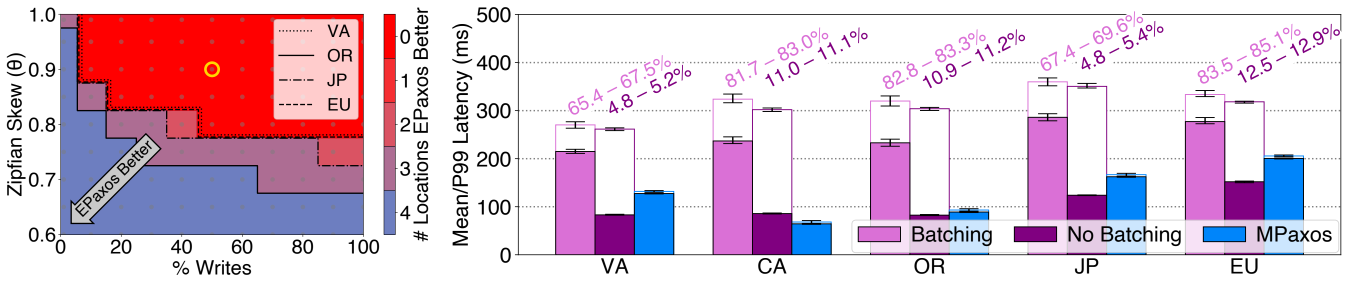
6.5 Latency Cost of Batching

The EPaxos paper demonstrated that batching operations into instances increases the throughput of EPaxos 9x and Multi-



(a) Mean Latency **(b)** P99 Latency **(c)** Complementary CDF

Figure 10: Latency impact of unbounded dependency chains for skewed, write-heaving workloads. All operations are writes, and the Zipfian experiments used a skew parameter of .99. These experiments were run at a throughput of 4000 operations per second, which is 80% of the maximum possible throughput of unmodified EPaxos. Figures (a) and (b) compare Mean and 99th percentile execution latency for EPaxos with out without an upper bound on execution latency. Error bars are shown as in Figure 6. Figure (c) is a complementary CDF showing the fraction of EPaxos operations with latency higher than a given value, with and without bounding dependency chains. Data is shown only for clients in OR; other sites are similar.



(a) Mean Latency **(b)** Circled Workload

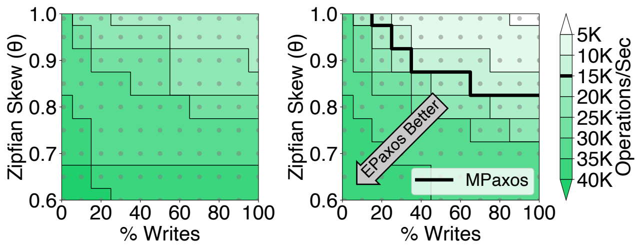
Figure 11: EPaxos execution latency and conflict rate with batching enabled. Figure (a) compares mean latency with batching enabled between EPaxos and Multi-Paxos for a variety of workloads. The lines indicate the workloads for which EPaxos mean latency in that client location is roughly equivalent to Multi-Paxos mean latency; above the line EPaxos latency is worse, and below it Multi-Paxos mean latency is worse. The fill colors indicate the number of client locations that experience a lower latency with EPaxos than Multi-Paxos for the given workload. The circled point indicates the workload in Figure (b): a Zipfian distribution with a skew parameter of .9, where 50% of operations issued by clients are writes. Figure (b) error bars and conflict rates are shown as in Figure 6.

Paxos 5x, but EPaxos batching significantly increases latency because it causes nearly all operations to take the slow path. This is because, with batching, an operation conflicts if any operation in its batch has a conflict.

The EPaxos paper measurements did not capture the drawback of batching, as they measured batching only with the '0%' and '100%' loads, and batching has no effect on conflict rate for either of those workloads. Figure 11a shows that the mean latency for EPaxos is worse than Multi-Paxos for most of our workloads when batching is enabled (compare with Figure 7, where EPaxos mean latency is almost always lower than Multi-Paxos). Figure 11b shows more detail for a particular workload. With batching, the conflict rates are 6-16x higher than without batching. As a consequence of high conflict rates, batching causes EPaxos mean latency to be more than double that of both Multi-Paxos and EPaxos without batching.

6.6 Wide-Area Throughput

EPaxos throughput, like latency, is heavily impacted by operation interference. Figure 12 shows the maximum throughput EPaxos can sustain for each workload point, both with and without the optimization to bound dependency chains. Even with bounded dependency chains, throughput varies by more than a factor of two across the range of workloads. Without the optimization it varies by a factor of 10. Throughput for the improved version of EPaxos is at least as good as Multi-Paxos for all workloads, and more than double Multi-Paxos when there is little interference. Throughput for the unimproved version of EPaxos can be as much as 3x lower than Multi-Paxos.



(a) Improved **(b)** Unmodified

Figure 12: EPaxos throughput for a variety of workloads, both with and without bounded dependency chains. Throughput was determined by varying the Poisson rate parameter until latency increased without an increase in throughput. Multi-Paxos maximum throughput is 15,000 operations per second. For the improved version of EPaxos, Multi-Paxos throughput is equal to that of EPaxos with a Zipfian skew of .99 and 100% writes, which is why there is no 'MPaxos' line on Figure (a).

Although our results show significantly worse throughput for EPaxos than the original evaluation, we are not confident that our throughput measurements (or those in the EPaxos paper) are a pure reflection of protocol differences rather than of confounding factors. When investigating the factors that limit server throughput, we observed several anomalies as servers neared saturation. Seemingly innocuous changes could have a large impact on throughput. For example, increasing the number of available cores caused throughput to drop. We believe that some of the issues may come from unexpected behavior of Go's thread scheduler. For example, for maximum throughput it is important that the thread implementing the EPaxos or Multi-Paxos protocol have exclusive access to a core. We suspect that Go occasionally deschedules that thread in order to run less-critical threads that listen for incoming messages.

Further, there are particular implementation details that may artificially limit throughput for both EPaxos and Multi-Paxos. For example, EPaxos has a thread that cycles through unexecuted instances, checking if they can be executed, which becomes increasingly expensive as the number of unexecuted instances increase, and could likely be optimized. In addition, we measured that Multi-Paxos disproportionately suffers from imbalance in the program's threading structure.

We experimented with wildly modified versions of the code that eliminated many of these compounding factors, but concluded that measuring the best possible throughput that is achievable with either protocol was tangential to the goal of our work. This would have required effectively rewriting the code, and we wanted to keep constant with the original EPaxos evaluation as many factors as possible.

Nonetheless, we felt that it was important to disclose our findings, as our results differ from those of the original EPaxos evaluation, and the compounding factors that we encountered apply to the original EPaxos evaluation as well as ours.

7 Related Work

A number of existing protocols exploit commutativity to provide fast replication. Other commutativity-based protocols may suffer from similar performance degradation to EPaxos in a WAN, but the EPaxos slow path can incur higher latency than that of most other protocols. We leave analysis of other protocols to future work. *Generalized Paxos* [16] allows clients to complete commutative operations in 1 WAN RTT. Generalized Paxos has a 3 RTT slow path because it reverts to a stable leader when operations interfere, and it also has a larger quorum size than EPaxos. *CURP* [24] completes commutative operations in 1 RTT by separating durability from ordering. CURP uses a stable leader, but has clients also issue their operations to 'witnesses', which keep track of a set of operations that have been seen but not yet ordered by the leader. If a client issues an operation that interferes with another that is unordered, witnesses reject that operation and it must take the slow path. Unlike EPaxos, CURP commutativity does not use ordering; even if all witnesses process an interfering operation in the same order, it will still conflict based on the presence of interference.

Bipartisan Paxos [30] (BPaxos) uses modularity to address scaling issues of optimized consensus protocols. BPaxos allows bottleneck modules to be scaled independently from the rest of the system, increasing throughput. The BPaxos evaluation has similar limitations to that of EPaxos; a fixed 'conflict percentage' is chosen instead of using a representative distribution, and they increase throughput by adding client threads. The protocol is evaluated in a local-area network (LAN), and the authors list poor performance in a WAN as a limitation.

TAPIR [31], *Janus* [22], and *MDCC* [14] combine concurrency control and consensus to commit commutative distributed transactions in 1 WAN RTT. Previously, replication and transaction protocols would each separately, and redundantly, ensure consistency. By only ensuring consistency once, these protocols decrease latency by 1 WAN RTT.

Speculative Paxos [25] and *Network-Ordered Paxos* [17] are consensus protocols that use ordering to complete operations in 1 RTT. Both use specialized networking hardware to serialize instances within the network, which is infeasible in a WAN.

Speculative PBFT [29] and and *Eve* [13] are replication protocols that optimistically execute operations before their ordering is confirmed, rolling back those that conflict. Speculative PBFT allows clients to speculatively execute based on a prelimary result while the leader replicates the command. However, the result cannot be externally visible before the client learns that it is committed. Eve allows for concurrent execution within a multi-core system.

PLATO [7] orders multi-cast messages in a LAN. A replica detects out-of-order packets by observing packet inter-arrival times. PLATO tags each message with a timestamp before which it should not be processed. This allows the packet to be revoked if another arrives that PLATO believes should have an earlier ordering. In addition to being insuitable for a leaderless protocol, PLATO's approach is speculative on the receiver, while our timestamps are determined by the sender and will be the same for each receiver.

CAESAR [5] and *Clock-RSM* [9] are consensus protocols that use synchronized clocks. CAESAR is a leaderless protocol that is similar to EPaxos, and uses synchronized clocks to allow instances to take the fast path even if the quorum disagrees on an instance's dependencies. In order to do this, CAESAR increases the quorum size by one node, which increases the minimum latency of operations. Clock-RSM uses timestamps from synchronized clocks to order instances. Instances are sorted by timestamp, and can complete when no unknown instance could have had an earlier timestamp. Because of this, a given instance depends on all other replicas communicating that they have no prior instances before it can complete. Much of the latency benefit of Clock-RSM is due to replicas broadcasting replies to all other replicas, which would also be possible for EPaxos.

Spanner [8], Google's globally distributed database, uses synchronized clocks to linearize transactions. This is at a higher level than the consensus layer, which uses Multi-Paxos.

On the correctness of Egalitarian Paxos [26] identifies an error in the EPaxos recovery procedure that breaks correctness. A potential fix for this issue is implemented in [27]. Our evaluation is concerned with the performance of normal operation.

8 Conclusion

In this paper we have shown how conclusions about a system can be biased by elements of the experimental methodology such as workload and metrics. Although we agree with the original evaluation that EPaxos reduces median latency compared to Multi-Paxos, we show that the percentage of operations that acheives this low latency varies greatly by application. Additionally, there is high variance in operation latency; for many workloads, a roughly 30% reduction in median latency can be at the expense of a more than 4x increase in 99th percentile latency. By covering a larger range of workloads, we also encountered issues that did not manifest in the original EPaxos paper, such as unbounded dependency chains, the cost of dependency delays (which can affect even the fast path), the latency penalty from batching, and the impact of conflicts on throughput.

We also presented TOQ, a novel technique that uses synchronized clocks to improve the performance of consensus protocols for geo-replication. We evaluated TOQ on EPaxos and demonstrated that it can reduce the conflict rate and mean latency of EPaxos without introducing additional latency.

9 Acknowledgements

Our work would not have been possible without the Egalitarian Paxos authors' transparency about their evaluation, both from the details in the body of their paper and the availability of their source code. We thank the authors of EPaxos, particularly Iulian Moraru, for their responsiveness to questions we had about the paper, protocol, their experiments, and our findings.

This work was supported by the industrial affiliates of the Stanford Platform Lab. Our experiments were run on Google Cloud with the support of a Google Cloud Platform Education Grant. We thank our shepherd, Shuai Mu, and the anonymous reviewers for their helpful comments on the paper.

References

[1] Zookeeper api. https://zookeeper.apache.org/doc/r3.4.6/api/org/apache/zookeeper/ZooKeeper.html, January 2016.

[2] Redis api. https://redis.io/commands, May 2020.

[3] T. Armstrong. Linkbench code base. https://github.com/facebookarchive/linkbench, November 2012.

[4] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. Linkbench: A database benchmark based on the facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, page 1185–1196, New York, NY, USA, 2013. Association for Computing Machinery.

[5] B. Arun, S. Peluso, R. Palmieri, G. Losa, and B. Ravindran. Speeding up consensus by chasing fast decisions. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 49–60, 2017.

[6] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, page 53–64, New York, NY, USA, 2012. Association for Computing Machinery.

[7] M. Balakrishnan, K. Birman, and A. Phanishayee. Plato: Predictive latency-aware total ordering. In *2006 25th IEEE Symposium on Reliable Distributed Systems (SRDS'06)*, pages 175–188, 2006.

[8] 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 *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 261–264, Hollywood, CA, 2012. USENIX Association.

[9] J. Du, D. Sciascia, S. Elnikety, W. Zwaenepoel, and F. Pedone. Clock-rsm: Low-latency inter-datacenter state machine replication using loosely synchronized physical clocks. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 343–354, 2014.

[10] Y. Geng, S. Liu, Z. Yin, A. Naik, B. Prabhakar, M. Rosunblum, and A. Vahdat. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI'18, page 81–94, USA, 2018. USENIX Association.

[11] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, SIGMOD '94, page 243–252, New York, NY, USA, 1994. Association for Computing Machinery.

[12] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. *SIGMOD Rec.*, 23(2):243–252, May 1994.

[13] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin. All about eve: Execute-verify replication for multi-core servers. In *10th USENIX Symposium*

on Operating Systems Design and Implementation (OSDI 12), pages 237–250, Hollywood, CA, Oct. 2012. USENIX Association.

[14] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. Mdcc: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, page 113–126, New York, NY, USA, 2013. Association for Computing Machinery.

[15] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[16] L. Lamport. Generalized consensus and paxos. Technical Report MSR-TR-2005-33, March 2005.

[17] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports. Just say NO to paxos overhead: Replacing consensus with network ordering. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 467–483, Savannah, GA, Nov. 2016. USENIX Association.

[18] I. Moraru. There is more consensus in egalitarian parliaments. `https://www.youtube.com/watch?v=KxoWlUZNKn8`.

[19] I. Moraru, D. G. Andersen, and M. Kaminsky. A proof of correctness for egalitarian paxos. 2012.

[20] I. Moraru, D. G. Andersen, and M. Kaminsky. Epaxos code base. `https://github.com/efficient/epaxos`, August 2013.

[21] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 358–372, New York, NY, USA, 2013. Association for Computing Machinery.

[22] S. Mu, L. Nelson, W. Lloyd, and J. Li. Consolidating concurrency control and consensus for commits under conflicts. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 517–532, USA, 2016. USENIX Association.

[23] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association.

[24] S. J. Park and J. Ousterhout. Exploiting commutativity for practical fast replication. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 47–64, Boston, MA, Feb. 2019. USENIX Association.

[25] D. R. K. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 43–57, Oakland, CA, May 2015. USENIX Association.

[26] P. Sutra. On the correctness of egalitarian paxos. *CoRR*, abs/1906.10917, 2019.

[27] P. Sutra. On the correctness of egalitarian paxos code base. `https://github.com/otrack/epaxos`, October 2019.

[28] S. Tollman. Epaxos revisited code base. `https://github.com/platformlab/epaxos`, March 2021.

[29] B. Wester, J. Cowling, E. B. Nightingale, P. M. Chen, J. Flinn, and B. Liskov. Tolerating latency in replicated state machines through client speculation. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, page 245–260, USA, 2009. USENIX Association.

[30] M. Whittaker, Giridharanil, A. Szekeres, J. M. Hellerstein, and I. Stoica. Bipartisan paxos: A modular state machine replication protocol. *arXiv preprint arXiv:2003.00331*, 2020.

[31] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 263–278, New York, NY, USA, 2015. Association for Computing Machinery.

A Appendix

A.1 Upper Bound on Execution Latency

With the improvement to EPaxos described in Section 3, execution latency is bounded at roughly 3 WAN RTTs. Let's say that replica A originates instance A.*i* and has transitive dependency B.*i* from replica B. In order for B.*i* to have an equal or lower sequence number than A.*i*, or not have A.*i* as a dependency, some other replica C in B's fast-path quorum must learn about B.*i* before A.*i*.

In the worst case, C will learn about B.*i* an instant before C learns about A.*i*, B.*i* will take the slow path, and B.*i*'s commit must propagate to A. Thus, the latency bound is the sum of:

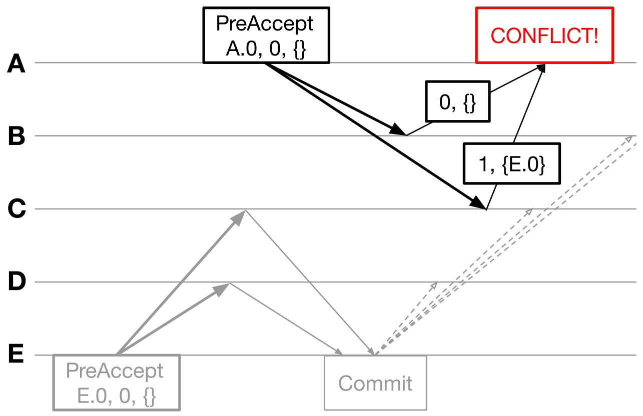
- 1 message delay: the one-way delay from A to C, which is A.*i*'s PreAccept message.
- <2 message delays: the remainder of the one-way delay from B to the other replicas in its quorum + the one-way delay for their responses. (This will be 1 message delay if C is the furthest replica in B's quorum, and always less than 2 message delays. For our topology, this is always 1 message delay.)
- 2 message delays: an RTT between B and its quorum, for the Accept phase.
- 1 message delay: the one-way delay from B to A, for B.*i*'s commit to propagate to A.

As an example, we can compute the upper bound on execution latency for Japan in our test topology. Figure 5 contains the RTTs between replicas in the topology, and we estimate that the one-way delay is half of RTT. The worst case for Japan (JP) is that an instance from London (EU) arrives at Oregon (OR) an instant before Japan's instance and takes the slow path. The maxmimum execution latency for the instance from Japan will be 45.5ms (one-way delay from JP to OR, for JP's PreAccept) + 64.5ms (one-way delay from OR to EU for OR's PreAccept reply) + 129ms (RTT between EU and OR, for the Accept phase) + 110.5ms (EU's commit propagation to JP) = 349.5ms. This is consistent with the maximum execution latency we measured for clients in Japan in our experiments, which was 349.9ms.

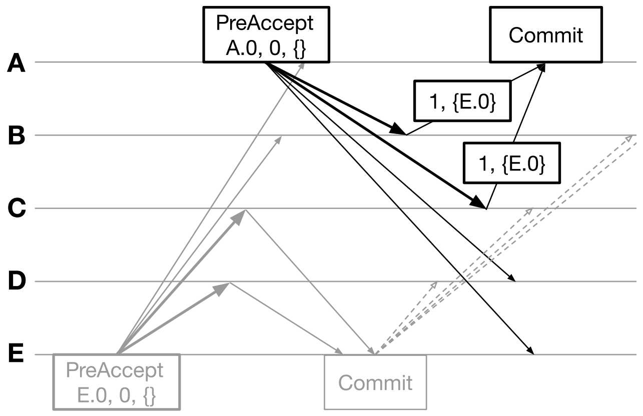
A.2 Impact of Thrifty on Conflict Rate

The original EPaxos evaluation used an optimization called *thrifty*, in which an originator only sends PreAccept and Accept messages to the necessary quorum. Although Enabling thrifty increases throughput for both EPaxos and Multi-Paxos by about 15%, enabling thrifty increases EPaxos conflict rates. With the thrifty optimization, some nodes do not become aware of an instance until it has committed. When some nodes know about an instance but others don't, there is increased potential for discrepancies in PreAcceptReplies, leading to increased conflicts.

Figure 13 provides an example of this. In the figure, replica E sends a PreAccept message for instance E.0 before replica A sends a PreAccept message for A.0. With thrifty enabled, only one of the two replicas in A's fast-path quorum receives the PreAccept for E.0 before the PreAccept for A.0. This causes A.0 to take the slow path. With thrifty disabled, both replicas in A's fast-path quorum receive the PreAccept for E.0 before that for A.0, so A.0 can commit on the fast path.



(a) Thrifty Enabled



(b) Thrifty Disabled

Figure 13: EPaxos PreAccept phase example, with and without *thrifty*. Replicas A and E originate interfering instances A.0 and E.0 concurrently. A's fast-path quorum consists of replicas B and C, and E's fast-path quorum consists of replicas C and D. With thrifty enabled, the responses from B and C differ for A.0. With thrifty disabled, replica B receives the PreAccept for E.0 before A.0, avoiding conflict.

In Figure 14, we demonstrate experimentally that thrifty increases EPaxos conflict rates. Conflict rates are higher with thrifty enabled for all three workloads in the figure, which generally leads to higher latency.

A.3 Impact of Throughput and Workload Size on Conflict Rate

When comparing EPaxos to a variety of skews and write ratios in Section 6, we held throughput and the number of unique keys in the workload constant. However, both of these variables also impact operation interference.

Figure 15 shows that EPaxos conflict rate, mean latency, and 99th percentile latency increase as throughput increases. Higher throughput corresponds to having a higher number of concurrent operations, creating more opportunities for interference. EPaxos 99th percentile latency plateaus once more

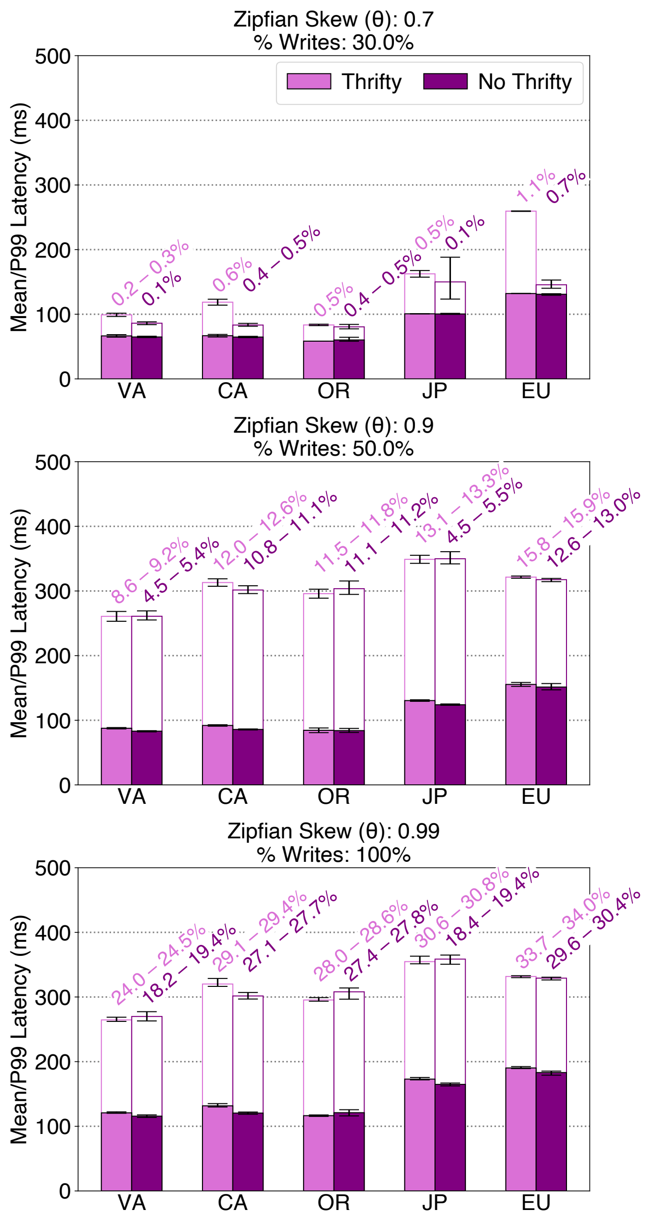


Figure 14: Comparing EPaxos Mean and 99th percentile execution latency with and without *thrifty* enabled for three different workloads. The experiments for this graph where run with the same methodology as Figure 7. Error bars and conflict rates are shown as in Figure 6.

than 1% of operations hit the upper bound on execution latency described in Appendix A.1.

Figure 16 shows that EPaxos conflict rate, mean latency, and 99th percentile latency decrease as the number of unique keys in the workload increases. Although our latency analyses showcase a variety of skew factors, our analyses only display measurements for a single number of unique keys. The trends would be the same with a different number of unique keys, but the measurements for each data point would be different.

A.4 Additional TOQ Graphs

In Section 6.3 and with Figure 9, we demonstrated that TOQ reduced conflict rate and, consequently, latency for clients in Oregon (OR). Figure 17 shows that TOQ reduces conflict rate for clients in all locations. Figure 18 shows that TOQ reduces Mean latency for all client locations, except for clients in Japan with quorum union sync. Figure 19 shows that TOQ reduces 99th percentile latency for all client locations. Mean latency increases for Japan with quorum union sync because the quorum union contains VA, which is significantly further from JP than its quorum, adding about 24ms to the minimum operation latency for clients in Japan. Although Mean latency increases for clients in Japan with quorum union sync, 99th percentile latency still decreases, due to decreased conflict rate.

Figure 20 demonstrates the benefits and tradeoffs of TOQ for two specific workloads. For both workloads, TOQ with a sync group of all replicas completely eliminates conflict. However, the mean latency for the 'all' sync group is higher than without TOQ because its minimum latency is higher. The quorum union sync group decreases conflict rate for all locations, and latency for most locations, but latency for JP is higher because the minimum latency is higher. The quorum sync group does not decrease conflict rate to the same extent as the 'all' or quorum union sync groups, but it purely decreases latency because its minimum latency is the same as without TOQ.

For the top workload in Figure 20, TOQ primarily decreases 99th percentile latency. Without TOQ, more than 1% of operations are at the latency bound described in Section A.1. TOQ reduces command interference enough that fewer than 1% of operations are at the bound. Because conflict rates without TOQ are fairly low, the impact of TOQ on mean latency for the top workload is minimal.

For the bottom workload in Figure 20, conflict rates are high enough that TOQ primarily decreases mean latency. There is a minimal impact on 99th percentile latency for all sync groups other than the 'all' sync group because, even with the quorum union sync group, more than 1% of operations are at the latency bound.

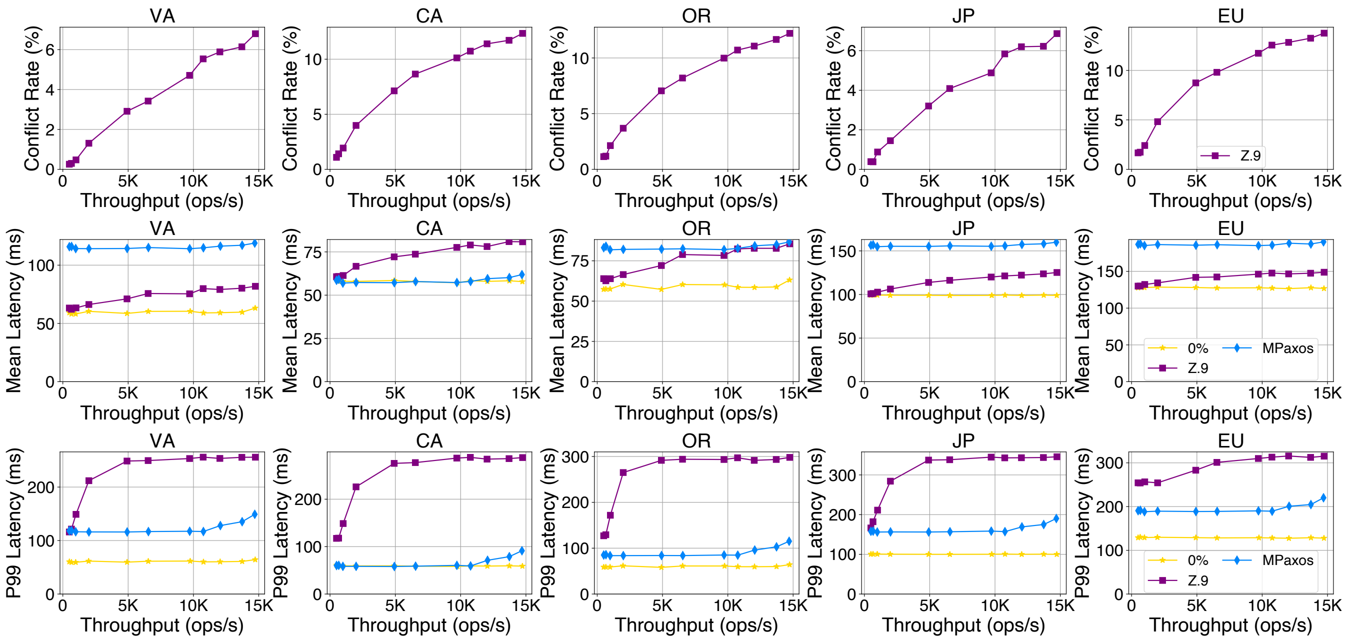


Figure 15: Conflict rate, mean latency, and 99th percentile latency as a function of throughput. To increase throughput, we varied the Poisson rate parameter. Maximum throughput for this the graph is the maximum that produces reasonable behavior for all protocols and experiments. 50% of client operations are writes, and the Zipfian distribution has 1 million unique keys.

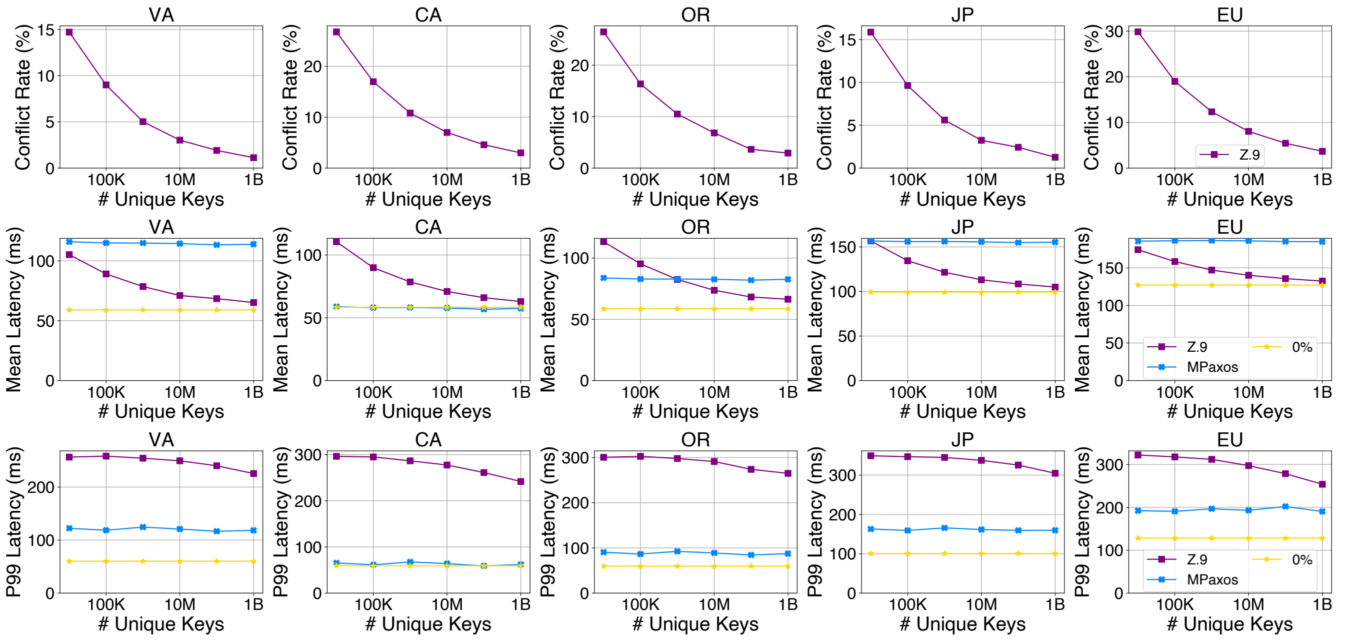


Figure 16: Conflict rate, mean latency, and 99th percentile latency as a function of the number of unique keys in the workload. All experiments were run at a throughput of 11,000 operations per second, which is 80% of maximum throughput for Multi-Paxos. 50% of client operations are writes.

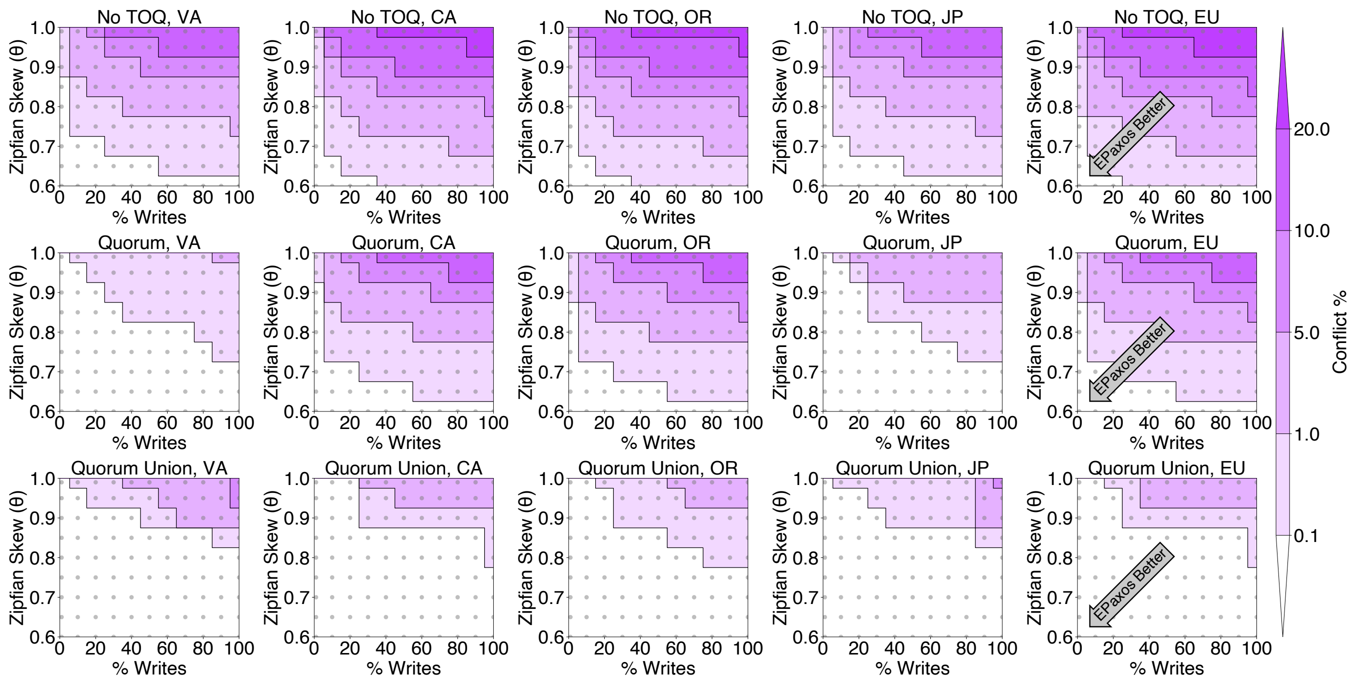


Figure 17: Comparing conflict rate for EPaxos without TOQ, EPaxos with TOQ to a quorum sync group, and EPaxos with TOQ to a quorum union sync group. The quorum union consists of Virginia, California, and Oregon. Other than TOQ, the experiments for this graph were run with the same methodology as Figure 7.

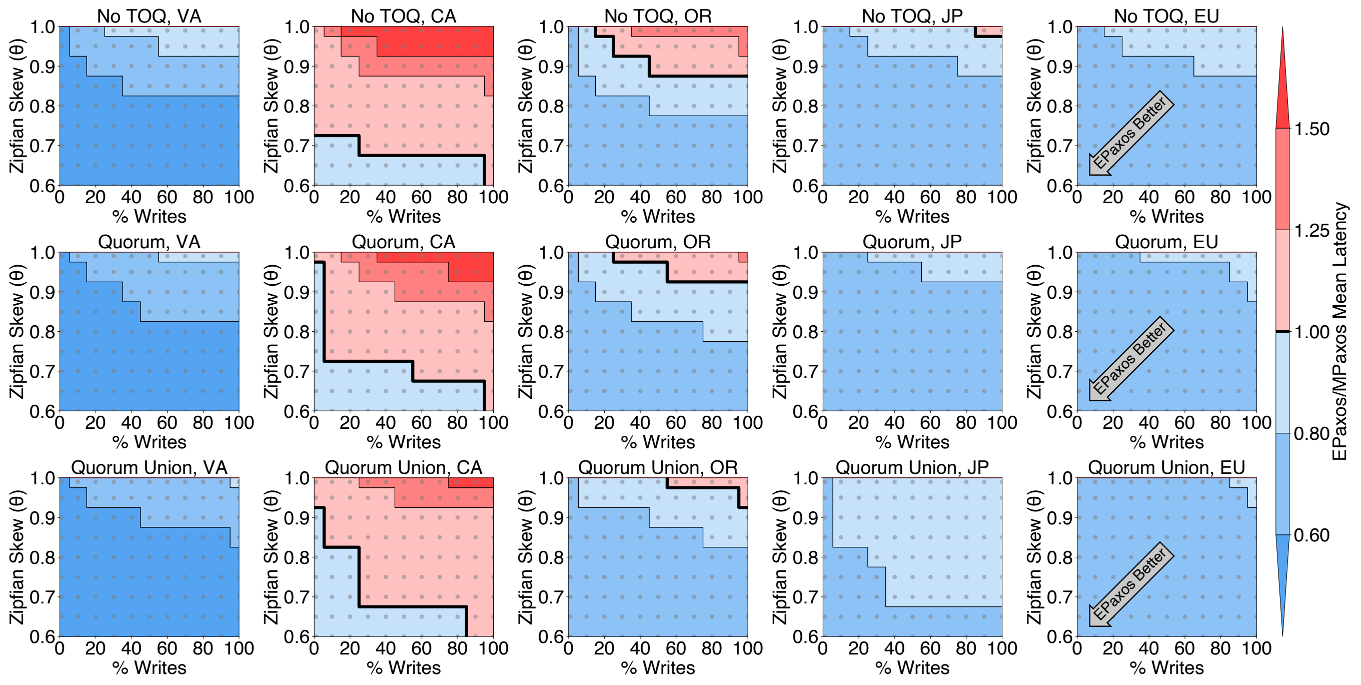


Figure 18: Comparing mean latency for EPaxos without TOQ, EPaxos with TOQ to a quorum sync group, and EPaxos with TOQ to a quorum union sync group. The quorum union consists of Virginia, California, and Oregon. Other than TOQ, the experiments for this graph where run with the same methodology as Figure 7.

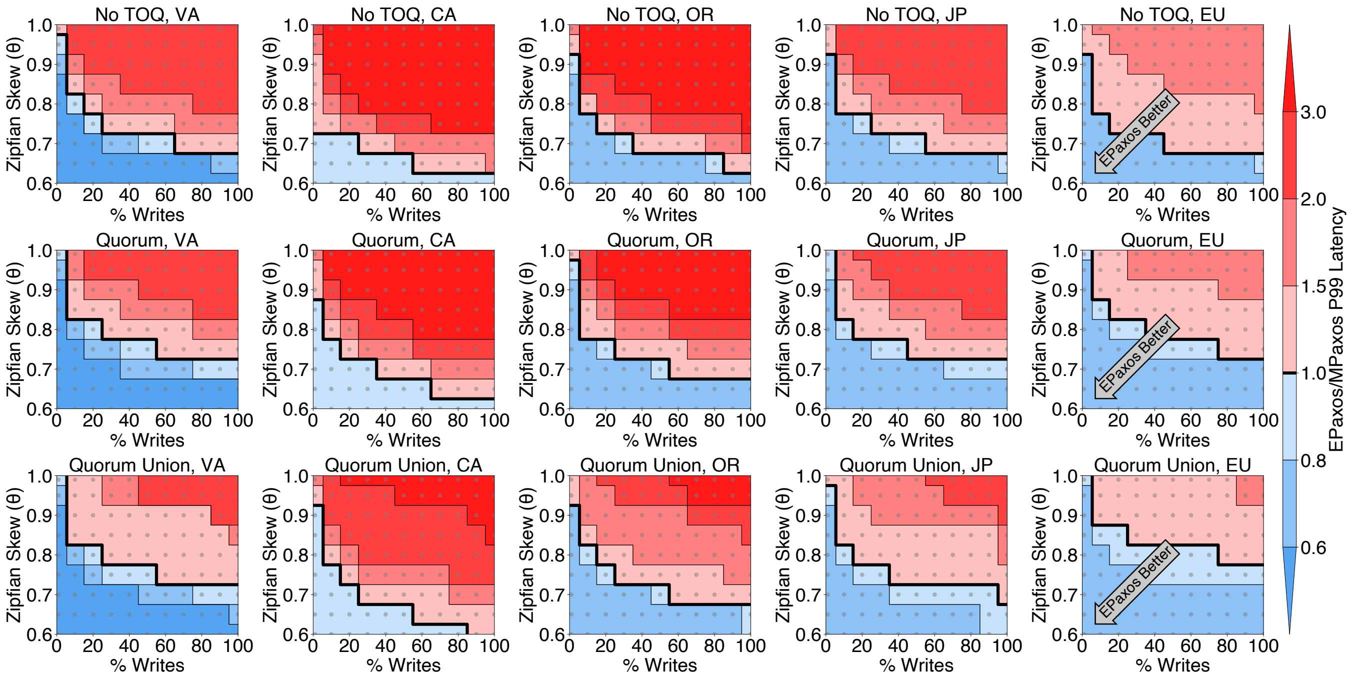


Figure 19: Comparing 99th percentile latency for EPaxos without TOQ, EPaxos with TOQ to a quorum sync group, and EPaxos with TOQ to a quorum union sync group. The quorum union consists of Virginia, California, and Oregon. Other than TOQ, the experiments for this graph where run with the same methodology as Figure 7.

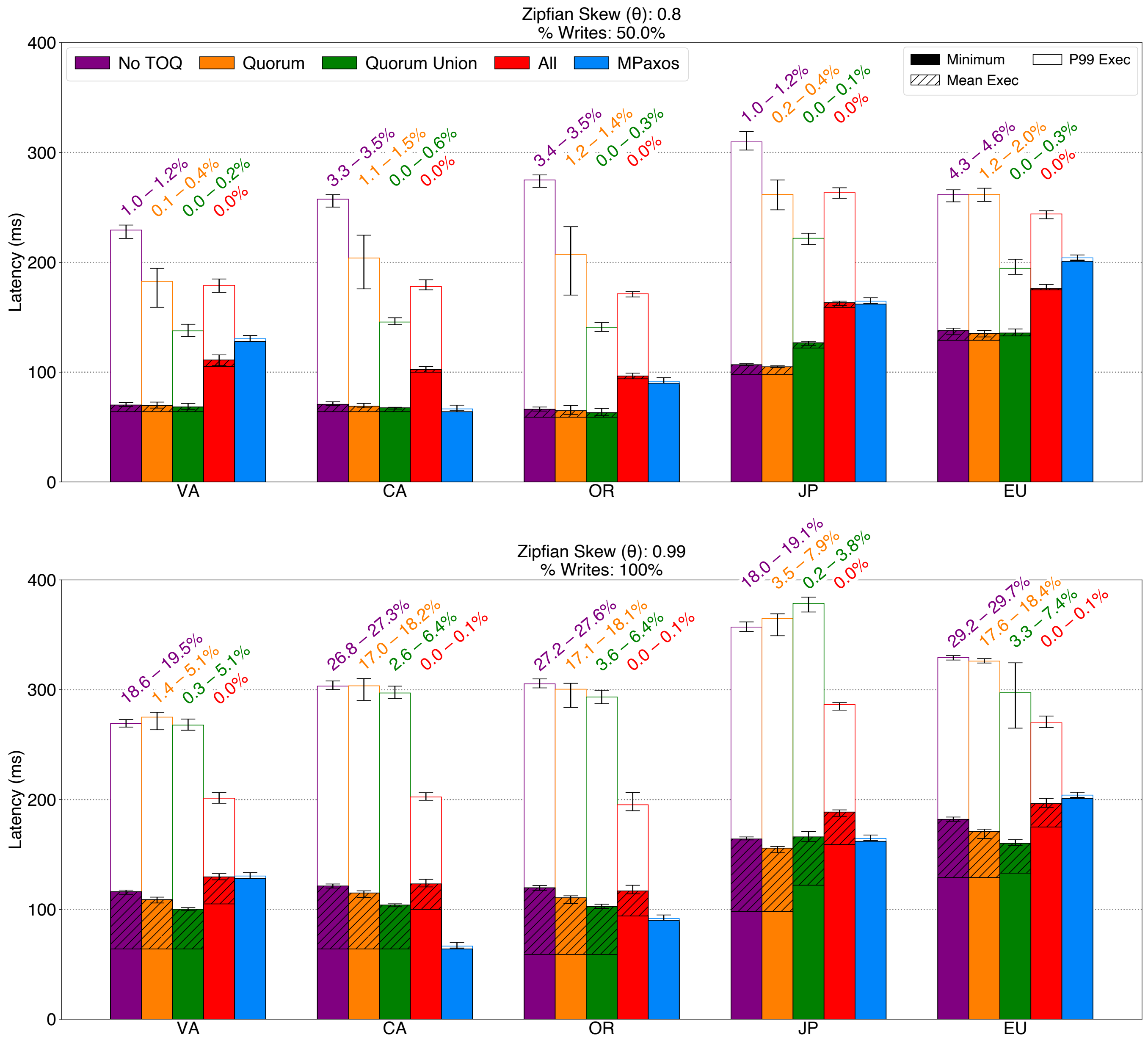


Figure 20: Comparing Mean and 99th percentile latency for EPaxos with three different TOQ sync groups. The top graph is a medium-conflict workload, and the bottom is a high-conflict workload. 'Minimum' refers to the minimum possible execution latency for the protocol and sync group. The quorum union consists of Virginia, California, and Oregon. Other than TOQ, the experiments for this graph where run with the same methodology as Figure 7. Error bars and conflict rates are shown as in Figure 6.

Ship Compute or Ship Data? Why Not Both?

Jie You*, Jingfeng Wu◇, Xin Jin†, and Mosharaf Chowdhury*
*University of Michigan, ◇Johns Hopkins University, †Peking University

Abstract

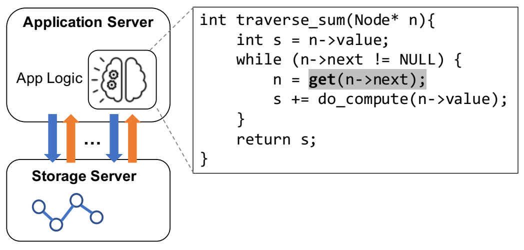
How cloud applications should interact with their data remains an active area of research. Over the last decade, many have suggested relying on a key-value (KV) interface to interact with data stored in remote storage servers, while others have vouched for the benefits of using remote procedure call (RPC). Instead of choosing one over the other, in this paper, we observe that an ideal solution must adaptively combine both of them in order to maximize throughput while meeting application latency requirements. To this end, we propose a new system called Kayak that proactively adjusts the rate of requests and the fraction of requests to be executed using RPC or KV, all in a fully decentralized and self-regulated manner. We theoretically prove that Kayak can quickly converge to the optimal parameters. We implement a system prototype of Kayak. Our evaluations show that Kayak achieves sub-second convergence and improves overall throughput by 32.5%-63.4% for compute-intensive workloads and up to 12.2% for non-compute-intensive and transactional workloads over the state-of-the-art.

1 Introduction

Two trends stand out amid the rapid changes in the landscape of cloud infrastructure in recent years:

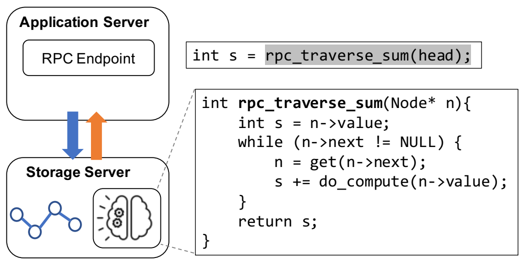
- First, with cloud networks moving from 1Gbps and a few hundred μs to 100Gbps and single-digit μs [11, 20, 25], disaggregated storage has become the norm [4, 17, 21, 28, 31, 40, 44]. It decouples compute from storage, enabling flexible provisioning, elastic scaling, and higher utilization. As a result, increasingly more applications now access their storage servers over the network using a key-value (KV) interface.
- Second, the steady increase in compute granularity, from virtual machines, to containers, to microservices and serverless functions, is popularizing storage-side computation using remote procedure calls (RPCs) [23]. Many databases allow stored procedures and user-defined functions [5–8, 26, 41, 42], and some KV stores allow just-in-time or pre-compiled runtime extensions [9, 18, 29, 39].

The confluence of these two contradicting trends – the former moves data to compute, while the latter does the opposite – highlights a long-standing challenge in distributed systems: *should we ship compute to data, or ship data to compute?*



```
int traverse_sum(Node* n){
    int s = n->value;
    while (n->next != NULL) {
        n = get(n->next);
        s += do_compute(n->value);
    }
    return s;
}
```

(a) *Ship data to compute.*



```
int s = rpc_traverse_sum(head);
```

```
int rpc_traverse_sum(Node* n){
    int s = n->value;
    while (n->next != NULL) {
        n = get(n->next);
        s += do_compute(n->value);
    }
    return s;
}
```

(b) *Ship compute to data.*

Figure 1: ***Graph traversal implemented with (a) disaggregated storage and (b) storage-side compute. The latter (1b) results in less network round-trips but exert more load on the storage server.***

The answer, in broad strokes, boils down to the ratio of computation and communication. The benefits of storage disaggregation, i.e., shipping data to compute, typically holds when most of the time of a function invocation (hereafter referred to as a request) is spent in computation. However, when a single request triggers multiple dependent accesses to the disaggregated storage, time spent in network traversals and data (un)marshalling starts to dominate [10]. Figure 1a shows an example of a simple graph traversal algorithm implemented on top of disaggregated storage, where "pointer chasing" can make network traversals the bottleneck.

In contrast, storage-side computation enables applications to offload part of their application logic to storage servers. The storage layer is customized to support application-specific RPC-style APIs [10, 12, 29], which shave off network round-trips. Figure 1b shows the previous example implemented with storage-side computing, where only one network round-trip is sufficient. However, this is not universally viable either; for compute-intensive workloads, the compute capacity of the storage servers can become the bottleneck when too much computation is offloaded to the storage.

In short, there is no one-size-fits-all solution. Existing

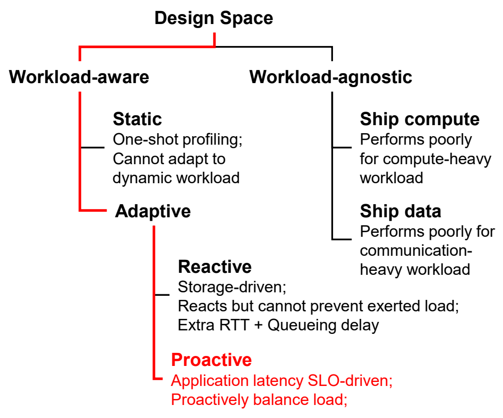

Design Space
Workload-aware
Workload-agnostic
Static
One-shot profiling; Cannot adapt to dynamic workload
Adaptive
Ship compute
Performs poorly for compute-heavy workload
Ship data
Performs poorly for communication-heavy workload
Reactive
Storage-driven; Reacts but cannot prevent exerted load; Extra RTT + Queueing delay
Proactive
Application latency SLO-driven; Proactively balance load;

Figure 2: ***Design space of Kayak.***

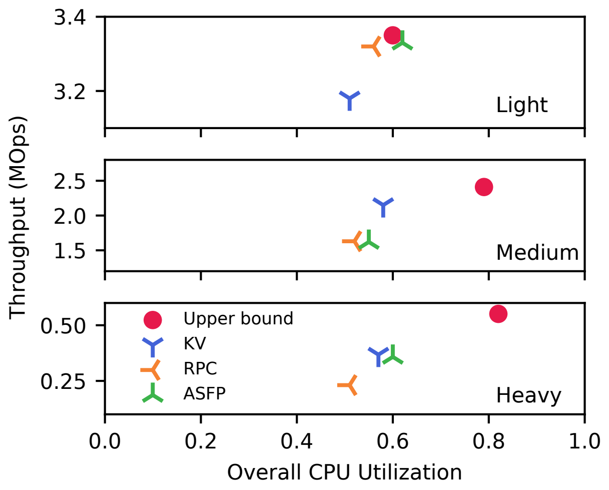


Figure 3: ***Throughput and overall request-level CPU utilization across both application and storage servers for three different workloads under different execution schemes, with 200µs SLO.***

works have explored different points in the design space (Figure 2). Arbitrarily forcing a disaggregated storage (i.e., ship data) or a storage-side computation architecture (i.e., ship compute) in a *workload-agnostic* manner leads to poor utilization and low throughput. Our measurements show that *workload-agnostic* solution leads to up to 58% lower throughput and 37% lower utilization when compared to the optimal. For *workload-aware* alternatives, one choice is taking a one-shot *static* approach, whereby a workload is profiled once at the beginning. However, statically choosing either KV- or RPC-based approach falls short even for a single tenant (§2.2).

The alternative, therefore, is taking an *adaptive* approach that can dynamically choose between shipping data and shipping compute. Existing adaptive solutions such as ASFP [12] take a *reactive* approach: all requests are forwarded using RPC to the storage server, which can then react by pushing some of them back. While this provides a centralized point of control, each request experiences non-zero server-side queueing delay, and more importantly, requests that are pushed back suffer from one *extra* round-trip time (RTT), which is detrimental to low-latency applications with strict latency SLOs. Moreover, throughput-driven designs cannot proactively throttle exerted load on the storage server w.r.t. tail latency service-level objectives (SLOs).

We observe that an ideal solution fundamentally calls for a *balanced* architecture that can effectively utilize the available resources and increase overall throughput while satisfying tail latency SLOs. In this paper, we present Kayak that takes a *proactive* adaptive approach to achieve these goals (§3). In order to maximize throughput without SLO violations, Kayak proactively decides between shipping compute and shipping data when executing incoming requests, and it throttles request rate in order to meet SLO requirements. Specifically, Kayak takes a latency-driven approach to optimize two parameters simultaneously: (1) the *request rate*, and (2) the *RPC fraction*, which denotes the proportion of the incoming requests to be executed using RPC.

Unfortunately, the optimal RPC fraction varies for different workloads and their SLO requirements. There is no closed-form expression to precisely capture the relationship between RPC fraction, request rate, and tail latency either. Finally, we show that the order in which we optimize request rate and RPC fraction affects convergence of the optimization algorithm. We address these challenges by designing a dynamic optimization method using a dual loop control (§4). Kayak employs a faster control loop to optimize request rate and a slower one to optimize RPC fraction. Combined together, Kayak iteratively searches for the optimal parameters, with a provable convergence guarantee. In addition to increasing throughput in the single-tenant scenario, Kayak must also ensure fairness and work conservation of shared server resources in multi-tenant settings. Kayak pins tenants to CPU cores in a fair manner and employs work stealing to achieve work conservation.

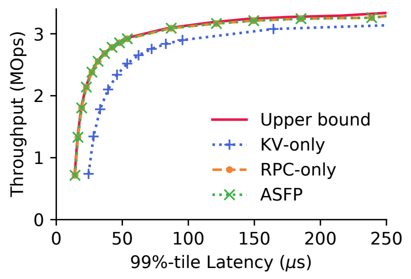
Our evaluation on a prototype of Kayak shows that: (1) Kayak achieves sub-second convergence to optimal throughput and RPC fraction regardless of workloads; (2) Kayak improves overall throughput by 32.5%-63.4% for compute-intensive workloads and up to 12.2% for non-compute-intensive and transactional workloads; and (3) in a multi-tenant setup, Kayak approximates max-min fair sharing and scales without sacrificing fairness.

2 Motivation

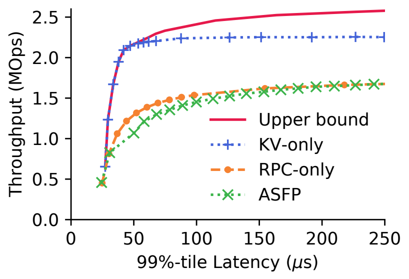
2.1 Limitations of Existing Designs

Existing solutions either fail to efficiently utilize the available CPU cores for a large variety of workloads or introduce additional overhead to reactively adapt to workload variations, both of which lead to lower throughput.

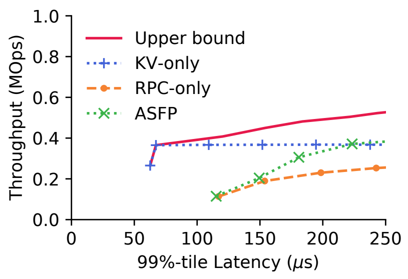
Workload-agnostic approaches either use KV-only design or RPC-only design. The former results in excessive network round-trips of storage access during execution, while the latter overloads the storage server CPU and leaves the CPU on the



(a) *Light (100ns/RTT)*



(b) *Medium (1μs/RTT)*



(c) *Heavy (10μs/RTT)*

Figure 4: ***Throughput w.r.t. SLO (99%-tile latency) for 3 different workloads under different execution schemes.***

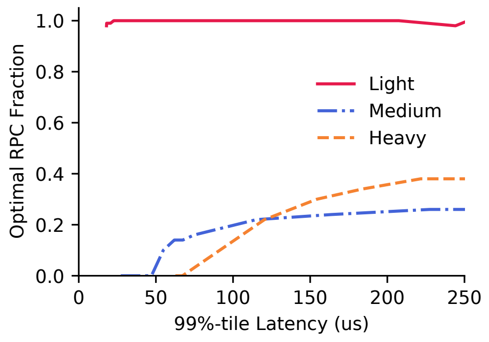


Figure 5: ***Optimal RPC fraction w.r.t. SLO (99%-tile latency) for 3 different workloads.***

application server underutilized. In either case, the overall CPU utilization is low which hinders the performance.

ASFP [12] presents an alternative to workload-agnostic solutions by taking a workload-aware approach with runtime adaptation. In that design, the requests are executed using RPC by default, and if the storage server gets overloaded by the exerted computation, a pushback mechanism is triggered to push the exerted computation back to the application server side. However, the excessive queueing on the storage server still cannot be prevented, although it can be alleviated by the pushback mechanism. Furthermore, the execution of pushed-back requests needs to restart on the application server, wasting CPU cycles on both servers.

To illustrate these issues, we perform an experiment with the graph traversal application shown in Figure 1. We configure the workload so that the each request triggers two storage accesses (i.e., two network round-trips) when executed using the KV scheme. We vary the computation after each access and refer to them as Light (100ns computation time per access), Medium (1μs per access) and Heavy (10μs per access). For reference, adding one network round-trip incurs 9.2μs more latency for each request in our testing environment. We measure the maximum achievable throughput and the CPU utilization (defined as CPU cycles spent only in executing the requests). As shown in Figure 3, using only KV or RPC leads to lower overall CPU utilization and lower throughput. Although the reactive adaptive design utilizes a higher amount of CPU on both application and storage servers, its overheads add up quickly, and its CPU usage for request-level computation does not increase significantly.

To summarize, existing designs do not efficiently utilize the CPU resource on both application and storage servers, which limits their performance. There exists an optimal RPC fraction that maximizes the throughput (calculated by a comprehensive sweep as explained below).

2.2 Need for Dynamically Finding the Optimal Fraction

A key challenge here is that this optimal fraction varies for different workloads. To highlight this, we perform another experiment with the same graph traversal workload as before. We configure the application server to handle a fraction of the requests using RPC and the rest using the KV approach. We vary the RPC fraction from 0 to 1 and measure the overall throughput and end-to-end latency of all requests. In doing so, we obtain the throughput-latency measurements for all possible execution configurations, as shown in Figure 4 and Figure 5. The *upper bound* in Figure 4 is defined by selecting the best RPC fraction for each latency SLO to maximize the throughput, and the selected fraction is plotted in Figure 5. ASFP is configured as using RPC by default with pushback enabled.

As shown in Figure 4, workload with less computation favors RPC over KV, and vice versa. For Medium and Heavy workloads, the highest throughput is achieved by combining RPC and KV together. Moreover, we observe that, (1) for different workloads, the highest throughput is achieved with different RPC fraction; and (2) for different SLO constraints, the optimal fraction for highest throughput is also different. This calls for a proactive design to search for the optimal RPC fraction. Note that we repeated the same parameter sweep with the number of storage accesses per request set to 4 and 8, and observe similar trends (please refer to Appendix A).

3 Kayak Overview

Kayak proactively decides between *shipping compute* and *shipping data* in a workload-adaptive manner. It arbitrates incoming requests and proactively decides the optimal RPC fraction (i.e., what fraction of the compute to ship to storage servers) of executing the requests while meeting end-to-end tail latency SLO constraints.

3.1 Design Goals

Kayak aims to meet the following design goals.

- *Maximize throughput without SLO violations:* Applications have stringent tail latency SLO constraints to ensure low user-perceived latency. Kayak should maximize the throughput while not violating these SLO constraints.
- *High CPU utilization across all servers:* Kayak should balance the computation imposed by application logic across the application and storage servers.
- *Fair sharing of storage server resources:* In a multi-tenant cloud, multiple applications may be sending requests to the same storage server, which should be fairly shared.
- *Ease of deployment:* Applications should be able to use Kayak with minimal code modification.

3.2 Architectural Overview

At a high level, Kayak adaptively runs application logic on both the application and storage servers (Figure 6). However, even though it ships some computation to the storage server, Kayak's core control logic runs only on the application server and the storage server acts as an extended executor. Unlike existing reactive solutions, Kayak proactively decides the amount of computation to ship to the storage side.

Key ideas. Essentially, Kayak finds the optimal RPC fraction and maximizes throughput at runtime while meeting tail latency SLO constraints. The main design challenge is that the optimal fraction varies in accordance with different workloads and their SLO requirements (Figure 5), as well as the amount of load exerted on the storage server. In a multi-tenant cloud, all of these change dynamically. In order to keep up with the changing environment, Kayak proactively adjust the RPC fraction and the request rate according to realtime tail latency measurements.

However, the relationship between the request rate, RPC fraction and latency cannot be easily captured with a closed-form expression. Thus Kayak adopts a numerical optimization method based on a dual loop control algorithm (§4.3) to search for the optimal parameters iteratively. We notice that latency measurements in real systems exhibit large variance, which detrimentally impacts the performance of such iterative optimization algorithms. Kayak's algorithm accounts for such variance and has a provable convergence guarantee.

From an implementation perspective, Kayak faces another challenge as it has to make adjustments very quickly due to the high-throughput, low-latency nature of the environment. Kayak cannot afford to gather global information and make centralized decisions. Hence, the algorithm for Kayak is fully decentralized and runs only on the application servers.

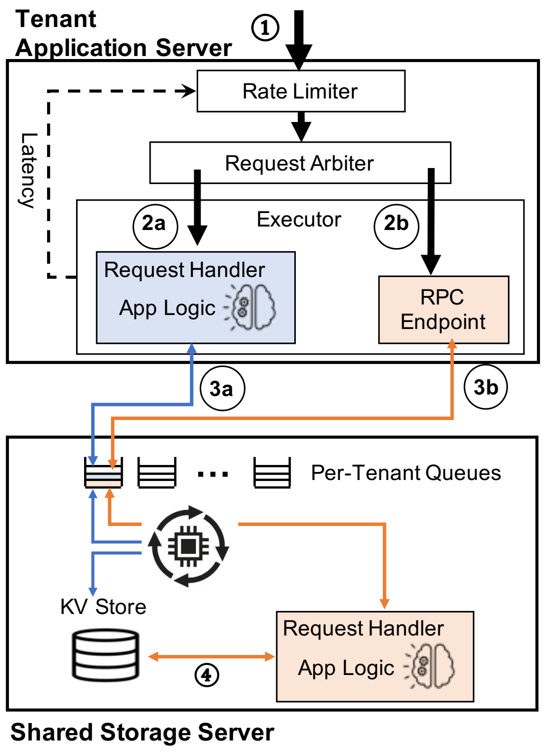


Figure 6: ***Kayak architecture. Tenant application servers interact with shared storage servers using Kayak, which proactively decides which requests to run on the application server using KV operations and which ones to ship to the storage server via RPC.***

Application server. The application server consists of three main components (Figure 6): (*i*) the Rate Limiter limits the rate of incoming requests from the tenant to satisfy SLO constraints; (*ii*) the Request Arbiter determines the optimal RPC fraction; and (*iii*) the Executor handles the requests according to the scheme decided by the Request Arbiter.

The Rate Limiter interacts with the tenants and ① receives incoming requests. It continuously monitors real-time tail latency of end-to-end request execution, and pushes back to signal the tenant to slow down. When the servers are overloaded and the SLO cannot be met, it drops overflowing requests. We assume that tenants implement mechanism to handle overflowing, such as resubmit dropped requests, and increase provisioning so that in the worst case scenario all requests can still be executed on all application servers within the SLO.

The Request Arbiter proactively determines the optimal RPC fraction. It selects the execution scheme for each request based on that fraction. For each request, the choice of execution scheme is determined by a Bernoulli distribution $B(1,X)$ where X is the proportion of requests processed using RPC.

The Executor handles the request in its entirety and reports the end-to-end completion time back to the Rate Limiter upon completion. It consists of two parts: (*i*) the Request Handler and (*ii*) the RPC Endpoint. If the request is to be executed using ②a the KV scheme, the Request Handler is triggered.

The Request Handler executes the application logic locally on the application server and keeps track of the states of the request. Whenever it needs to access data stored in the storage server, (3a) the KV API of the storage server is subsequently called. In contrast, if the request is to be executed on the storage side using the RPC scheme, then the request is simply forwarded to (2b) the RPC Endpoint on the application server. The RPC Endpoint issues an RPC request (3b) to the storage server for processing the request.

Storage server. The storage server includes an additional Request Handler to handle RPC requests in addition to a KV interface. Similar to allocating CPU cores in the application server to run application code, in Kayak, computation resources in the storage server are also allocated to specific tenants at the CPU core granularity.

Each tenant has a dedicated request queue, from which its core(s) polls KV and RPC requests. Handling an incoming KV request in Kayak is the same as what happens in a traditional KV store: the request is simply forwarded to the KV store. Upon receiving an RPC request, the Request Handler is triggered and executes the application logic on the storage server. The Request Handler calls (4) the local KV API whenever data access is needed, interacting with the stored data without crossing the network.

This static *pin-request-to-core* allocation scheme of Kayak makes it easier to enforce fair computation resource sharing between tenants. However, static allocation of CPU cores cannot guarantee work conservation of the CPU cores on the storage server. Kayak uses *work stealing* to mitigate this issue: whenever a tenant's dedicated queue is empty, the corresponding CPU core *steals* requests from other queues.

4 Kayak Design

Our primary objective is to maximize the total throughput without violating the tail latency SLO. However, higher throughput inevitably leads to higher latency in a finite system [27], and there exists a fundamental tradeoff between throughput and latency. Unfortunately, the precise relationship between latency and throughput of a real system, however, is notoriously difficult to be captured by a closed-form expression. In this paper, we use an analytical model to highlight our insights and take a tail latency measurement-driven approach to design a pragmatic solution.

At the same time, as illustrated in Section 2, a reactive approach to achieve this can lead to CPU wastage. Hence, Kayak proactively decides what fraction of the requests to offload vs. which ones to run in the application server, while maximizing the total throughput within the SLO constraint. The need for optimizing both raises a natural question: *which one to optimize first?* In this section, we analyze both optimization orders and design a dual loop control algorithm with provable convergence guarantees. Detailed proofs can be found in the appendix.

Sym.	Description
R	Total request rate
X	Proportion of requests processed using RPC
τ	Random variable of request latency
t_o	Latency SLO target
$T(X,R)$	Latency SLO as a function of X, R
$R(X)$	Function implicitly defined by $T(X,R(X)) = t_0$
k	Index of iterations

Table 1: ***Key notations in problem formulation.***

4.1 Problem Formulation

We denote the proportion of requests to be executed using RPC by X, the total incoming request rate by R, and we define τ as the random variable of request latency, thus we have:

$$\tau \sim P(R,X),$$

where R and X are the parameters of distribution P. Table 1 includes the key notations used in this paper.

We denote $T(X,R)$ as our SLO statistics metric, which takes a specific statistical interpretation for the particular SLO metric. For instance, if the SLO is defined as the 99%-tile latency then T is the 99%-tile for τ. We denote t_0 as the SLO target under the same statistic metric. Thus the problem can be formulated as:

$$\max_X \quad R \tag{1}$$
$$\text{s.t.} \quad T(X,R) \leq t_0 \tag{2}$$
$$R > 0, \tag{3}$$
$$X \in [0,1]. \tag{4}$$

Here constraint (2) captures the latency SLO constraint, and constraints (3) and (4) represents the boundary of R and X, respectively.

We make the following observation when solving this optimization problem:

Observation 1. *Fixing X, $F_X(R) := T(X,R)$ is monotonic increasing.*

Observation 1 captures the relationship of throughput and latency from queueing theory [27] for finite systems like Kayak.

4.2 Strawman: X-R Dual Loop Control

Optimization (1) cannot be directly solved with a closed-form solution of R and X due to the intractability of the function $T(X,R)$. Therefore, we use a numeric optimization method and try to optimize R and X independently and iteratively. To put it into our context, we need to design an iterative algorithm such that in each iteration, we first optimize either R or X, and then optimize the other. We also have to prove that this algorithm would actually converge to ensure optimality and stability of the system.

Now we are facing a question: *which one to optimize first?* In our problem, there is an asymmetry for X and R: X is the

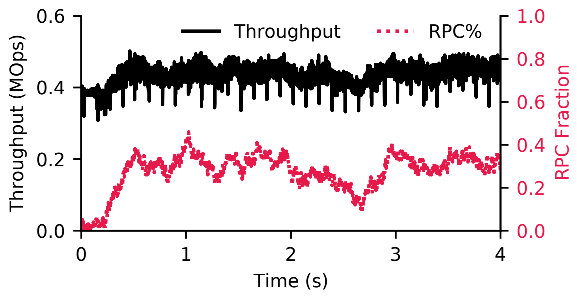


Figure 7: ***Instability of throughput and RPC fraction w.r.t time, with X-R Dual Loop Control.***

parameter in Optimization (1) where as R is the objective. A straightforward solution is to optimize X first, which leads to the following algorithm.

Algorithm I (X-R Dual Loop Control) For $k = 1, \ldots, K$, we alternatively update X_k and R_k by

1. Fix R_k, and find the RPC fraction X_k that minimizes latency, i.e., $X_k = \arg\min_X T(X, R_k)$;

2. Update R_k according to *gradient descent* so that $T \approx t_0$, i.e.,
$$R_{k+1} = R_k + \eta \left(t_0 - T(X_k, R_k)\right),$$
where $\eta > 0$ is the stepsize.

In this algorithm, the first step is to solve a convex optimization problem. Because our assumptions guarantee that X_k is unique and finite, this iteration is well defined for at least the first step (and we will show it is also good for the second step). Moreover, because $T_{R_k}(X) := T(X, R_k)$ is μ-strongly convex and L-smooth, X_k can be solved very quickly (or mathematically, in a linear rate) by iterative algorithms such as gradient descent. We have the following theorem that characterizes the convergence of this algorithm. The rigorous statements and proofs are deferred to Appendix B.2.

Theorem 1. *Fixing R, $F_R(X) := T(X,R)$ is strongly convex and smooth. Suppose for all X, $0 < \alpha \leq \frac{\partial T(X,R)}{\partial R} \leq \beta$. Let $0 < \eta < \frac{1}{\beta}$, then under mild additional assumptions*[1],
$$|R_K - R_*| \leq (1 - \eta\alpha)^K \cdot |R_0 - R_*|.$$

Here R_* denotes the optimal request throughput, and R_0 denotes the initialization. This result shows the iteration of X-R Dual Loop Control converges to the optimal requests exponentially fast, i.e., after at most $O(\log \frac{1}{\varepsilon})$ iterations, the algorithm outputs a solution that is ε-close to the optimal.

Instability of X-R dual loop control. However, while Algorithm I is theoretically sound, it is not practical to be implemented in a real system. The key obstacle is that the latency SLO metric cannot be *directly* obtained in practice. Instead, we can only measure a set of samples of latency τ, and then gather statistics to derive the SLO metric. Hence the derived SLO metric – be it average or 99%-tile – is only an *estimate* $\widehat{T}$ based on *sampling*. While sampling might not be a problem for many systems by using a high sampling rate, it is indeed a problem for Kayak. In particular, because of the microsecond-scale workload and the real-time requirement of Kayak, the sample size for each estimate is limited. This leads to *large variance* in the estimated $\hat{T}$, which results in *degraded* convergence speed and quality.

To quantify the impact of this variance and show the gap between theory and practice, we conduct a verification experiment. We run Algorithm I with the Heavy workload from Section 2 under an SLO constraint of 200μs 99%-tile latency. Our experiment confirms the aforementioned issue of variance in SLO estimates. Figure 7 shows poor convergence quality of both the throughput and the RPC fraction.

[1]For the sake of presentation, we omit the technical assumptions. For a complete description on the theorem, please refer to Appendix B.2

4.3 Our Solution: R-X Dual Loop Control

A naive mitigation to counter the SLO variance is to simply use a metric that is more robust, such as average latency. But this limits the operators to only one viable SLO metric and compromises the generality of the system.

In order to solve the challenge of unstable SLO estimates, we must design an algorithm that is not sensitive to the variance of $\hat{T}$. Compared with the RPC fraction X, the request rate R has a more intuitive and better-studied interaction with latency T from extensive study in queueing theory. Specifically, we take inspiration from recent works [19, 30] showing that even with variance in latency measurements, one can still achieve rate control (i.e., optimization of the throughput) in a *stable* manner. From the starting point of latency-driven rate control, we design a dual loop control algorithm that first optimizes R and then optimizes X to numerically solve the optimization problem. The algorithm is shown as follows. The first part is latency-driven rate control, and the second is gradient ascent.

Algorithm II (R-X Dual Loop Control) For $k = 1, \ldots, K$, we respectively update X_k and R_k by

1. Apply rate control so that the latency approximates SLO, i.e., *R_k be such that $T(X_k, R_k) \approx t_0$;*

2. Use gradient ascent to search for the optimal X_k, i.e., *$X_{k+1} = X_k + \eta \frac{\mathrm{d}R_k}{\mathrm{d}X}$, where $T(X, R_k) = t_0$, and η is a positive stepsize.*

R loop: rate control. In order to satisfy the SLO requirement ($T \leq t_0$), we need to carefully control the request rate R. Intuitively, too high an R leads to excessive queueing on the server side, causing SLO violations; at the same time, too low an R leads to low overall throughput and low resource utilization.

Input: Current throughput R, latency t, SLO target t_0
Output: Updated throughput R

```
/* Initialize global variables. */
1 $\mathbb{T} \leftarrow 0$; $\mathbb{R} \leftarrow 0$            ▷ Last involved latency and throughput.
/* Update $R$ for each round. */
2 Procedure UpdateR()
      /* Calculate $\Delta_R$ according to Newton's method. */
3     $\Delta_R \leftarrow \frac{(R-\mathbb{R})(t_0-T)}{T-\mathbb{T}}$
      /* Bounds checking, throughput should be positive. */
4     if $R + \Delta_R < 0$ then                 ▷ Unlikely. Violates (3).
5         $R \leftarrow \frac{R}{M}$               ▷ Discard $\Delta_R$ and divide $R$ by half.
6     else
7         $R \leftarrow \Delta_R + R$
```

Pseudocode. 1: Dynamic search of optimal R.

Let $R(X)$ be a function implicitly determined by the boundary constraint Eq. (2), i.e.,

$$T(X, R(X)) = t_0.$$

The implicit function $R(X)$ is indeed well defined, since for any X, $T(X,R)$ is monotonically increasing,[2] implying there exists an unique request throughput $R(X)$ that satisfies the boundary constraint, i.e., the maximum throughput is achieved when the latency is equal to the SLO target.

Essentially, we have to design a dynamic algorithm that actuates $R(X)$ in real-time (via $R_k(X)$ in step 1). This problem can be solved with a root-finding algorithm such as the classic Newton's method. However, if we apply this method directly, we may encounter situations where the updated throughput R is negative, which violates constraint (3). This happens when the throughput is too high and needs to be significantly reduced. In this case, we divide R by M instead of updating it using Newton's method. This ensures that (i) the updated throughput is positive; and (ii) the updated throughput is still significantly lower than before. We note that this out-of-bound scenario does not happen frequently. For simplicity, we choose $M = 2$. Our algorithm of searching for the optimal R is shown in Pseudocode 1.

X loop: RPC fraction control. For any given RPC fraction, the rate control of Kayak essentially maximizes throughput within the allowance of SLO requirement. With rate control, we effectively get the throughput as a function of the given RPC fraction ($R(X)$). In this part, we focus on the complementary and optimize the RPC fraction to maximize $R(X)$. We use a gradient ascent algorithm to achieve that. When the updated RPC fraction falls out of the range of $[0,1]$, we apply

[2]We assume that $T(X,R)$ is continuous, and for any X, there exist R_1 and R_2 such that $T(X,R_1) \le t_0 \le T(X,R_2)$. This assumption pluses monotonicity yields the existence and uniqueness of the implicit function $R(X)$.

Input: Current throughput R, RPC propotion X,
Output: Updated RPC propotion X

```
/* Initialize global variables. */
1 $\mathbb{R} \leftarrow 0$; $\mathbb{X} \leftarrow 0$   ▷ Last involved throughput and RPC fraction.
/* Update $X$ for each round. */
2 Procedure UpdateX()
      /* Calculate $\Delta_X$ according to Gradient Ascent. */
3     $\Delta_X \leftarrow -\eta \frac{R-\mathbb{R}}{X-\mathbb{X}}$
      /* Bounds checking, $X$ should be within constraints. */
4     if $X + \Delta_X \notin [0,1]$ then          ▷ Unlikely. Violates (4).
5         $X \leftarrow \max\{\min\{X + \Delta_X, 1\}, 0\}$
6     else
7         $X \leftarrow \Delta_X + X$
```

Pseudocode. 2: Dynamic search of optimal X.

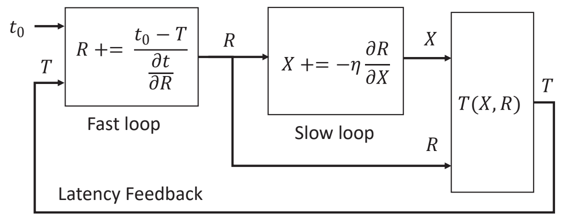

t_0
T
$R \mathrel{+}= \frac{t_0 - T}{\frac{\partial t}{\partial R}}$
Fast loop
R
$X \mathrel{+}= -\eta \frac{\partial R}{\partial X}$
Slow loop
X
R
$T(X,R)$
T
Latency Feedback

Figure 8: ***Nested control loops of Kayak.***

rounding to ensure it is within the boundary. Our algorithm of searching the RPC fraction is shown in Pseudocode 2.

Putting them together. Combing the rate control and the RPC fraction control, our algorithm (Algorithm II) naturally forms a bi-level (nested) control loops [15], with two actuators X and R and only one feedback signal t. We adopt a single control loop (the inner/fast loop), called R loop, to implement the rate control, i.e., finding the maximum throughput R while not violating the SLO t_0. The input of this control loop is the measured latency SLO metric $\widehat{T}$ and the output is request rate R which is the input for our request arbiter. We then adopt another control loop (the outer/slow loop), called X loop, to implement our request arbiter, i.e., choosing the best X that maximizes R_0.

Although this dual loop control design decouples the two actuators X and R, the resulting two feedback loops may be coupled. The coupling between two feedback loops may cause oscillation, which can be mitigated by choosing different sampling frequencies [15]. The exact two values can be tuned by the operator according to different workloads and system configurations. However, because the functioning of the second loop is dependent on the output of the first loop (R) to have converged to a stable point, it is best practice to choose a lower frequency for the second loop. Theoretically, we show that this dual loop control algorithm is guaranteed to converge in Section 4.4. Empirically, in our experiments, we let the sampling rates of the first and second loops to be 200Hz and 20Hz respectively, and we show that the system converges fast to near optimal throughput in Section 6. We evaluate the impact of frequency selection in detail in Section 6.5.

4.4 Perfomance Guarantee

From the R loop, we obtain an estimation $R_k(x)$ at each iteration k, which approximately satisfies $T(X, R_k(X)) \approx t_0$. In the X loop, we optimize X for our request arbiter such that R_0 is maximized. This is done by *stochastic gradient ascent* (SGA, or *online gradient ascent*) on X. There is a rich literature in online learning theory for SGA when $R_k(x)$ is concave, e.g., see [36]. Applying related theoretical results to our problem, we have the following performance guarantee for our system. The proof of Theorem 2 is deferred to Appendix B.3.

Theorem 2. *Suppose for all $k = 1, \dots, K$, $R_k(X)$ is concave, and $\|\nabla R_k(X)\|_2 \le L$. Consider the iterates of SGA, i.e.,*

$$X_{k+1} = X_k + \eta \nabla R_k(X_k).$$

Then we have the following regret bound

$$\sum_{k=1}^{K} (R_k(X_*) - R_k(X_k)) \le C \cdot \sqrt{K}, \qquad (5)$$

where $C := L\|X_1 - X_*\|_2$ is a constant depends on initialization and gradient bound, and X_* can be any fixed number. Note that the regret bound holds even $R_k(X)$ is chosen adversarially based on the algorithm history.

Inteperation of Theorem 2. The sublinear regret bound implies SAG behaviors nearly optimal on average: we see this by setting $X_* = \arg\max_X \sum_{k=1}^{K} R_k(X)$, and noticing that

$$\frac{1}{K}\sum_{k=1}^{K} R_k(X_*) - \frac{1}{K}\sum_{k=1}^{K} R_k(X_k) \le O\left(\frac{1}{\sqrt{K}}\right) \to 0.$$

More concisely, in our algorithm, $\{R_k(X)\}_{k=1}^{K}$ corresponds to a sequence of inaccurate estimations to the true implicit function $R(X)$ — even so the theorem guarantees a sublinear regret bound, which implies that our algorithm behaviors nearly as good as one can ever expect under the estimations, no matter how inaccurate they could be.

Furthermore, if for each k, $R_k(X)$ is an *unbiased estimator* to the true concave function $R(X)$, i.e., $\mathbb{E}R_k(X) = R(X)$, then $\bar{X} = \frac{1}{K}\sum_{k=1}^{K} X_k$ converges to the maximal of $R(X)$ in expectation: we see this by choosing $X_* = \arg\min_X R(X)$ and noticing that

$$\begin{aligned}\mathbb{E}\left[R(X_*) - R(\bar{X})\right] &\le \frac{1}{K}\sum_{k=1}^{K} \mathbb{E}\left[R(X_*) - R(X_k)\right] \\ &= \frac{1}{K}\sum_{k=1}^{K} \mathbb{E}\left[R_k(X_*) - R_k(X_k)\right] \\ &\le C \cdot \frac{1}{\sqrt{K}} \to 0.\end{aligned}$$

The above convergence result does not require any assumptions on the randomness of R_k, as long as $R_k(X)$ is an unbiased estimator of $R(X)$. This means our algorithm can *tolerate* variance in the measured latency which causes variance in estimated R_k. The convergence is empirically validated by our experiments in Section 6.1.

4.5 Scalability and Fault Tolerance

Scalability. Kayak is fully decentralized, and its control logic (e.g., rate and RPC fraction determination) is decoupled from the request execution in the dataplane. Throughput of a tenant is limited by its total available resources in application and storage servers; one can increase throughput by adding more application servers or by ensuring more resource share in the storage servers.

Fault tolerance. Kayak does not introduce additional systems components beyond what traditional KV- or RPC-based or hybrid systems do. As such, it does not introduce novel fault tolerance challenges. The consistency and fault tolerance of the KV store is orthogonal to our problem and out of the scope of this paper.

5 Implementation

We build a prototype of Kayak with about 1500 lines of code and integrate it with the in-memory kernel-bypassing key-value store Splinter [29]. The code is available at: `https://github.com/SymbioticLab/Kayak`

Kayak interface. Users of Kayak provide their custom defined storage functions (App Logic in Figure 6), which are compiled with Kayak and deployed onto both the application server and storage server. At runtime, users connect to Kayak and set the desired SLO target. Users then submit request in the format of storage function invocations to Kayak.

Application server. The core control logic of Kayak is implemented in the application server. One challenge we face during implementation is to optimize the code to reduce overhead, which is especially important because of the high throughput low latency requirement. For instance, the inner control loop constantly measures request latency and calculate the 99%-tile. One naive way is to measure the quantile is using selection algorithm to calculate the k-th order statistics of n samples, with has at least $O(n)$ complexity. Instead, we apply DDSketch [32] to estimate the quantile in real time with bounded error.

Storage server. The main challenge of implementing the storage server is supporting multi-tenancy and ensuring fairness and work conservation. We pin requests from different tenants to different CPU cores to ensure fairness. And we adopt work stealing to ensure work conservation: CPU cores with no requests to process steal requests from the queues of other cores. Specifically, similar to ZygOS [38], each CPU core of Kayak steals from all other CPU cores, which is different from Splinter's work stealing from only neighboring

CPU	Intel E5-2640v4 2.4 GHz
RAM	64GB ECC Memory DDR4 2400MHz
NIC	Mellanox ConnectX-4 25 GB NIC
OS	Ubuntu 16.04, Linux 4.4.0-142

Table 2: ***Server configurations for our testbed in CloudLab.***

cores. This further improves overall CPU utilization.

6 Evaluation

In this section we empirically evaluate Kayak with a focus on: (i) verification of convergence; (ii) performance improvement against state of the art [12]; and (iii) fairness and scalability with multiple tenants. Our key results are as follows.

- Kayak achieves sub-second convergence to optimal throughput and RPC fraction regardless of workloads. It can proactively adjust to dynamic workload change as well (§6.1).
- Kayak improves overall throughput by 32.5%-63.4% for compute-intensive workloads and up to 12.2% for non-compute-intensive and transactional workloads (§6.2).
- In a multi-tenant setup, Kayak approximates max-min fair sharing, with a Jain's Fairness Index [22] of 0.9996 (§6.3) and scales without sacrificing fairness (§6.4).
- We also evaluate Kayak's sensitivity to its different parameters (§6.5).

Methodology. We run our experiments on CloudLab [3] HPE ProLiant XL170r machines (Table 2). Unless specified otherwise, we configure Kayak to use 8 CPU cores across all servers. The fast control loop algorithm is configured to run every 5ms and the slow control loop runs every 50ms. The initial RPC fraction is set at 100%, and we define SLO as the 99%-tile latency.

Workloads. We use the workload described in Section 2. Unless otherwise specified, we configure the workload with a traversal depth of two so that each request issues two data accesses to the storage. We vary the amount of computation that takes place after each access and refer to them as Light (100ns computation time per access), Medium (1μs per access) and Heavy (10μs per access). This workload emulates a variety of workloads with different computational load in a non-transactional environment.

We extend this workload and create a Bimodal workload. We denote by `Bimodal(1us, 100ns, 50%, 5s)`, a workload that consists of 50% Medium (1μs/RTT) and 50% Light (100ns/RTT) with an interval of 5 seconds.

We also run YCSB-T [14] as a transactional workload. This workload is not computationally intensive.

Unless otherwise specified, for all workloads, we set our latency SLO target as: 99%-tile request latency lower than or equal to 200μs.

Baseline. Our primary baseline is ASFP [12], which is available at https://github.com/utah-scs/splinter/releases/tag/ATC'20 and also built on top of Splinter [29].

6.1 Convergence

In this section, we validate that Kayak's fast loop can converge to a stable throughput R while satisfying SLO constraint and when running together with fast loop, the slow loop can also converge to the optimal RPC fraction.

Fast loop only. We first disable the slow loop and run Kayak with a fixed RPC fraction (100%), to show that the fast loop (rate control) can converge to optimal throughput with different workloads.

We run Light, Medium and Heavy workloads with one application server and one storage server, and measure how the throughput and 99%-tile latency changes with time. As shown in Figure 9, Kayak ramps up the throughput quickly when the measured 99%-tile request latency is below the SLO threshold of 200μs. Along with the increase of throughput, the latency also increases, as observed from the rise of red line. The entire converging process happens within 0.2 seconds.

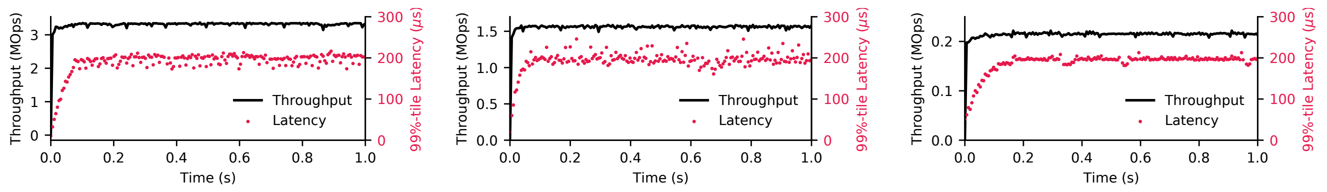
After approaching the SLO limit, both the throughput and latency remains stable with minor fluctuations, confirming the convergence of our fast loop. We note that the converged throughput are the same as the measurements of the RPC-only configuration in Figure 4. This means that our fast loop indeed converges to the optimal throughput.

Dual loop control. Now we move on to verifying the convergence of both loops combined. We repeat the previous experiments, but with both control loops enabled. Figure 10 shows the dynamics of throughput and RPC fraction and how they change with time. We highlight three observations.

- Similar to Figure 9, throughput increases rapidly within the first 0.2 seconds; this is due to the fast loop.
- With the Medium and Heavy workloads, the throughput increase slows down after 0.2 seconds. This increase comes from the slow loop, as we can see a change in RPC fraction. Note that the Light workload does not show this trend, because in this setup the initial RPC fraction (100%) is already the optimal for it.
- After 1 second since the start, the throughput converges to a stable value with only minor fluctuations.

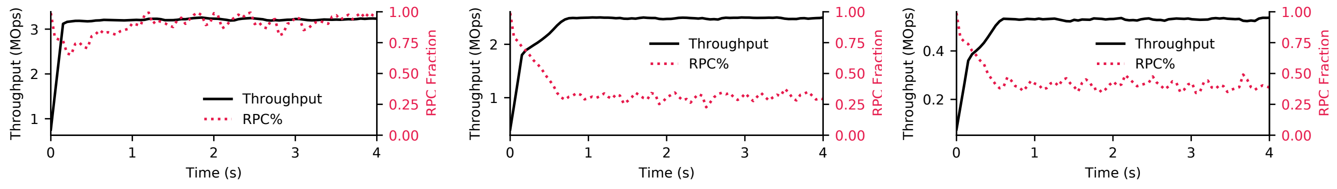
Comparing the RPC fraction in Figure 10 against Figure 5, we observe that our algorithm converges to the optimal RPC fraction. Comparing the throughput in Figure 10 against the Optimal configuration in Figure 4, we observe that the converged throughput is the optimal throughput.

Convergence under dynamic workloads. One advantage of Kayak is that it can proactively adjust to changing workload. To verify this, we run Kayak with the `Bimodal(1us, 100ns, 50%, 5s)` workload. Figure 11 shows the dynamics of throughput and RPC fraction. As we can see, Kayak adapts to the changing workload, and adjusts both the throughput and RPC fraction accordingly in a timely fashion.



(a) *Light (100ns/RTT)* (b) *Medium (1μs/RTT)* (c) *Heavy (10μs/RTT)*

Figure 9: ***Throughput and 99%-tile latency w.r.t time, with only the fast loop of Kayak and fixed RPC fraction of 100%.***



(a) *Light (100ns/RTT)* (b) *Medium (1μs/RTT)* (c) *Heavy (10μs/RTT)*

Figure 10: ***Dynamics of throughput and RPC fraction w.r.t time, with nested control loops of Kayak.***

SLO	YCSB-T		Light		Medium		Heavy		Bimodal	
	Kayak	ASFP	Kayak	ASFP	Kayak	ASFP	Kayak	ASFP	Kayak	ASFP
50μs	2.63	2.58	3.05	3.05	1.71	1.06	N/A	N/A	2.13	1.43
100μs	3.12	2.78	3.36	3.36	2.37	1.45	0.37	N/A	2.74	2.16
200μs	3.35	3.01	3.59	3.52	2.54	1.64	0.48	0.33	2.98	2.37
400μs	3.35	3.02	3.70	3.61	2.61	1.68	0.57	0.40	3.03	2.48

Table 3: ***Throughput (MOps) of Kayak and ASFP under different workloads and SLO targets. "N/A" means the SLO target is infeasible.***

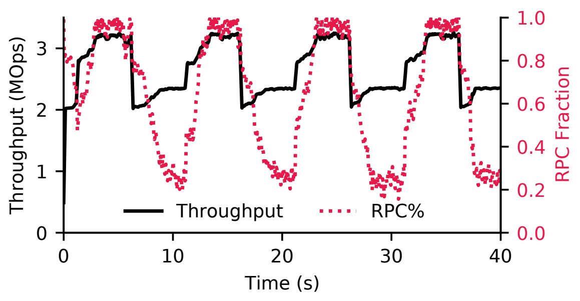


Figure 11: ***Dynamics of throughput and RPC fraction w.r.t time under Bimodal workload.***

6.2 Performance

Performance improvement. We compare the performance of Kayak against the state of the art ASFP [12]. We use all the workloads: (i) Light/Medium/Heavy computational workload; (ii) Bimodal workload; and (iii) YCSB-T workload. We use one application server and one storage server, and vary the SLO target from 50μs to 400μs. The results are shown in Table 3, which we summarize as follows.

- For non-compute-intensive workloads (Light and YCSB-T), Kayak achieves up to 12.2% throughput improvement. In this case, most of the requests are handled via RPC and Kayak's opportunity for improvement is lower.
- For compute-intensive workloads, Kayak achieves 32.5%-63.4% throughput improvement. In this case, the RPC fraction decreases, and ASFP's pushback mechanism kicks in; the overhead of pushing requests back in comparison to Kayak's proactive placement increases the gap.

Workload	YCSB-T	Light	Medium	Heavy	Bimodal
Gap	5.4%	11.8%	6.7%	3.5%	10.6%

Table 4: ***Kayak's performance gap from the upper bound for different workloads.***

Overall, Kayak outperforms ASFP because its proactive design is more efficient in using both application- and storage-side CPUs.

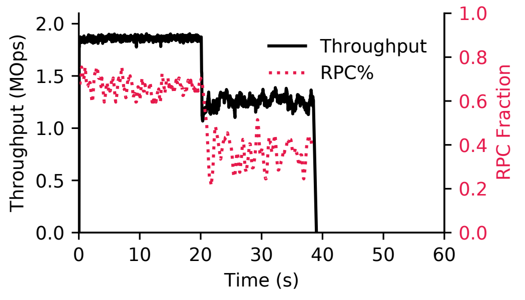
Gap between Kayak and upper bound. We set the SLO target to be 200μs and compare the achieved throughput between using Kayak and the upper bound obtained by the parameter sweep method (§2.2). We define the gap between Kayak and the upper bound as follows:

$$Gap = \frac{Throughput_{max}}{Throughput_{max,Kayak}} - 1$$

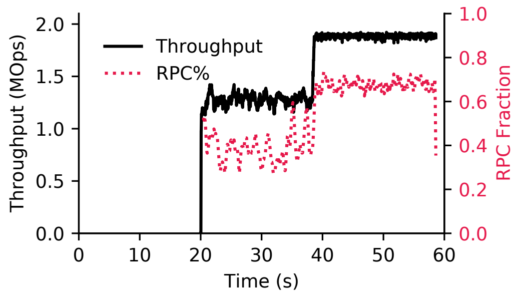
Intuitively, the lower the gap is, the closer Kayak is to the optimal. As shown in Table 4, Kayak has a slowdown of 11.8% for computationally light workload and 3.5% for computationally heavy workload.

6.3 Fairness

In this section, we show that Kayak can enforce max-min fairness when multiple tenants contend for server resources.



(a) *Tenant A*



(b) *Tenant B*

Figure 12: ***Dynamics of throughput and RPC fraction w.r.t time of tenant A and B in the multi-tenant experiment.***

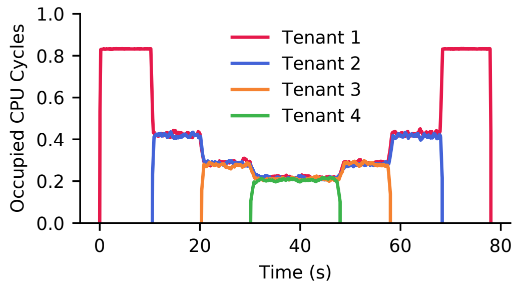
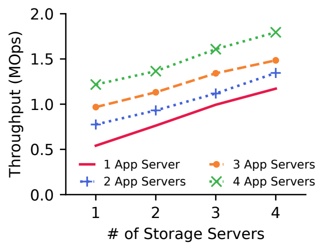
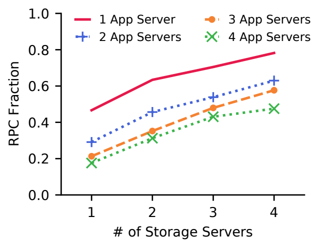


Figure 13: ***Fair-sharing of throughput for 4 applications sharing one storage server.***



(a) *Throughput*



(b) *RPC Fraction*

Figure 14: ***Throughput and RPC Fraction of Kayak with different server configurations.***

We run two tenants with the same setup and configure each to use four CPU cores instead of eight, while the server is still configured with eight CPU cores. Tenant A is started first, and after 20 seconds, tenant B is started. Figure 12a shows the dynamics of tenant A. During the first 20 seconds, tenant A quickly converges to the optimal throughput of around 1.82MOps. After 20 seconds when tenant B is started, the throughput achieved by tenant A drops to around 1.21MOps. Note that in this process, the optimal RPC fraction also shifts from 60% to 40%. This is because after tenant B joins, the server has less CPU resource to process tenant A's requests. After 40 seconds, tenant A stops sending requests.

Figure 12b shows the dynamics of tenant B, and we can see that when the two tenants are running together the achieved throughput is 1.21MOps and 1.24MOps, respectively. The gap between the two tenants is only 2.4%, which indicates that the server resources are fairly shared. After 40 seconds, the throughput and RPC fraction of tenant B increases because the storage server has more available CPU resources after tenant A stops.

Then we increase to four tenants and start the tenants one after one, with ten seconds in between. Each tenant is configured with one CPU core, and runs a different workload. Tenant {1, 2, 3, 4} runs {Light,Medium,Heavy,Bimodal}, respectively. We plot the occupied CPU cycles on the storage server for each tenant in Figure 13. When the 4 tenants are running together, we measure the Jain's Fairness Index [22] to be 0.9996.

6.4 Scalability

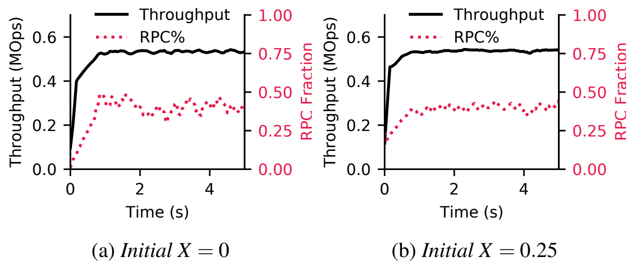
In this section, we verify that the Kayak control loops are decoupled from data plane and do not limit Kayak's scalability. To do so, we run experiments with Heavy workload, and vary the number of application and storage servers. For simplicity, we make sure a single request does not access data from more than one storage servers. We measure the total throughput and the converged average RPC fraction. As shown in Figure 14a, throughput increases with both adding a storage and an application server. This is because adding either essentially adds more CPU cores to the system. Note that the RPC fraction increases with the increment of the storage servers but decreases with the increment of the application servers (Figure 14b). This is because Kayak can judiciously arbitrate the requests and balance the load between the application and storage servers, by choosing the optimal RPC fraction.

6.5 Sensitivity Analysis

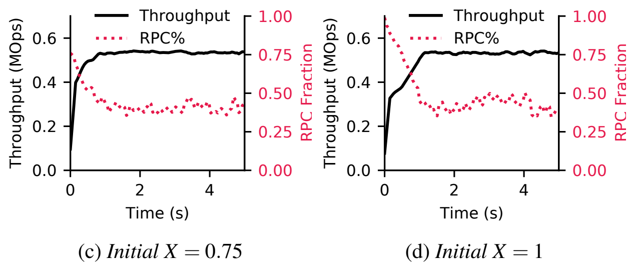
Initial state. First we evaluate whether Kayak is sensitive to its initial state. We run four experiments using the Heavy workload, and vary the starting value for RPC fraction X from $\{0, 0.25, 0.75, 0.100\}$. As shown in Figure 15, Kayak converges to the optimal request throughput and RPC fraction regardless of the initial RPC fraction in all four scenarios.

Choice of loop frequencies. In Section 4 we argue that in order for the dual loop control to work, we need to choose appropriate sampling frequencies for both loops. Here we analyze how different sampling frequencies affect the dynamics of our system.

We run two experiments using the Medium workload, and



(a) *Initial* $X = 0$ (b) *Initial* $X = 0.25$



(c) *Initial* $X = 0.75$ (d) *Initial* $X = 1$

Figure 15: *Throughput and RPC fraction during the converging process, with different starting RPC fraction.*

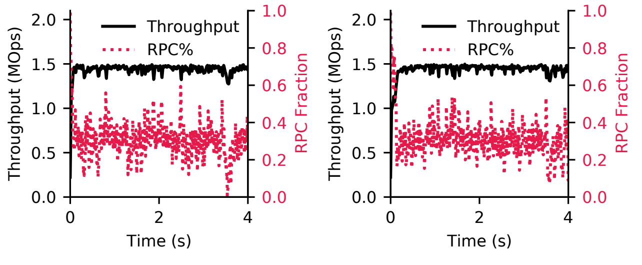
plot our results in Figure 16. In the first experiment we set the interval for both loop to be 200Hz; in the second experiment we invert the loop frequency so that the inner control loop runs slower than outer control loop. As shown in Figure 16a and 16b, the convergence quality degrades significantly in both cases. Hence, it is important to choose proper sampling frequencies to ensure that the inner control loop runs faster than the outer.

7 Discussion and Future Work

Rate Limiting. In Kayak, we consider rate limiting and admission control because simply moving up the latency curve cannot push the servers beyond their physical capacity, and we consider that if a tenant specifies a strict SLO target, requests that miss this target are essentially failed requests. As such, we assume that the tenants will implement a mechanism such as resubmitting requests which are dropped due to rate limiting, after adapting to a slower rate. At a slower timescale, tenants should also increase provisioning to avoid having to slow down, so that in the worst case scenario all requests can still be executed on all application servers within the SLO target.

Storage Function Differentiation. Kayak does not inspect individual storage functions, which reduces overhead and aligns with our design goal to be non-intrusive to applications. Therefore, Kayak can not distinguish between individual storage functions. We would like to explore how to differentiate individual storage functions without changing application code, which would allow us to implement more fine-grained control policy.

Developing Storage Functions. Currently, developers using Kayak and Splinter [29] have to code the same application logic again in the storage function which run inside the storage server. This duplication complicates the development process and increases the chance of human errors and bugs. Automatically generating server-side storage functions from code written using traditional KV API can alleviate this problem. This would be an interesting avenue for future work.



(a) *Equal loop frequency* (b) *Inverse loop frequency*

Figure 16: *Dynamics of throughput and RPC fraction w.r.t time, with different loop frequencies: (a) both loops run at 200Hz; (b) inner loop at 20Hz, outer loop at 200Hz.*

8 Related Work

Key-Value Stores. A recent line of research on key-value store has been focusing on utilizing RDMA to boost key-value store performance. Stuedi et al. [43] have achieved a 20% reduction in GET CPU load for Memcached using soft-iWARP without Infiniband hardware. Pilaf [33] uses only one-sided RDMA read to reduce CPU overhead of key-value store. In addition, it uses a verifiable data structure to detect read-write races. FaRM [16] proposes a new main memory key-value store built on top of RDMA. FaRM comes with transaction support but still support lock-free RDMA reads. HERD [24] further improves the performance of RDMA-based key-value store by focusing on reducing network round trips while using efficient RDMA primitives. FaSST [25] generalizes HERD and used two-sided RPC to reduce the number of QPs used in symmetric settings such as distributed transaction processing, improving scalability.

While there have been many research works focusing on improving raw performance of key-value stores, few investigates real performance implications on the application. TAO [13] is an application-aware key-value store by Facebook that optimizes for social graph processing. Kayak builds on top of this concept and focuses on application-level objectives such as SLO constraint.

Storage-side computation. Storage-side computation (i.e. *shipping compute to data*) has made its way from latency-insensitive big data systems such as MapReduce [1, 37, 45] and SQL databases [5–8, 26, 41, 42] into latency-critical KV stores [9, 18, 29, 39]. Comet [18] supports sandboxed Lua extensions to allow user-defined extensions to customize the storage by enabling application-specific operations. Malacology [39] utilizes Lua extensions contributed by users of the Ceph storage system [2], allowing installing and updating new object interfaces at runtime. Splinter [29] pushes bare-metal extension to storage server to allow RPC-like operations in addition to traditional key-value operations. These works breaks the assumption of dissagregated storage and necessitates the need for proactive arbitration provided by Kayak.

Adaptive compute placement. An emerging line of research aims at adaptively balancing between client-side processing and server-side processing. ASFP [12] extends Splinter by reactively pushing back requests to the client side if the server gets overloaded, but at the cost of wasting CPU and network resource.Instead, Kayak proactively balances the load exerted on both application and storage server. Cell [34] implements a B-tree store on RDMA supporting both client-side (RDMA-based) and server-side (RPC-based) search. Cell determines between these two schemes by tracking RDMA operation latency. This requires instrumentation into the application, which Kayak avoids by measuring end-to-end request latency instead. A recent work called Storm [35] uses a reactive-adaptive approach similar to that of ASFP [12] but with a different policy, where for each request it will try the traditional KV API first, and switch to RPC API if it detects that the application is trying to chase the pointers.

9 Conclusion

In this paper, we show that by proactively and adaptively combining RPC *and* KV together, overall throughput and CPU utilization can be improved. We propose an algorithm that dynamically adjusts the rate of requests and the RPC fraction to improve overall request throughput while meeting latency SLO requirements. We then prove that our algorithm can converge to the optimal parameters. We design and implement a system called Kayak. Our system implementation ensures work conservation and fairness across multiple tenants. Our evaluations show that Kayak achieves sub-second convergence and improves overall throughput by 32.5%-63.4% for compute-intensive workloads and up to 12.2% for non-compute-intensive and transactional workloads.

Acknowledgements

Special thanks go to the entire CloudLab team for making Kayak experiments possible. We would also like to thank the anonymous reviewers, our shepherd, Ryan Stutsman, and SymbioticLab members for their insightful feedback. This work is in part supported by NSF grants CNS-1813487, CNS-1845853, CNS-1900665, CNS-1909067, and CCF-1918757.

References

[1] Apache Hadoop. https://hadoop.apache.org/.

[2] Ceph. https://ceph.io/.

[3] CloudLab. https://cloudlab.us/.

[4] Intel Rack Scale Design. https://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design-overview.html.

[5] Microsoft SQL Server Stored Procedures. https://docs.microsoft.com/en-us/sql/relational-databases/stored-procedures/stored-procedures-database-engine.

[6] Microsoft SQL Server User-Defined Functions. https://docs.microsoft.com/en-us/sql/relational-databases/user-defined-functions/user-defined-functions.

[7] MySQL Stored Procedures Tutorial. https://www.mysqltutorial.org/mysql-stored-procedure-tutorial.aspx/.

[8] Oracle PL/SQL. https://www.oracle.com/database/technologies/application-development-PL/SQL.html.

[9] Redis. https://redis.io/.

[10] Atul Adya, Daniel Myers, Henry Qin, and Robert Grandl. Fast key-value stores: An idea whose time has come and gone. In *HotOS*, 2019.

[11] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Communications of the ACM*, 60(4):48–54, 2017.

[12] Ankit Bhardwaj, Chinmay Kulkarni, and Ryan Stutsman. Adaptive placement for in-memory storage functions. In *ATC*, 2020.

[13] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. Tao: Facebook's distributed data store for the social graph. In *ATC*, 2013.

[14] Akon Dey, Alan Fekete, Raghunath Nambiar, and Uwe Röhm. Ycsb+ t: Benchmarking web-scale transactional databases. In *2014 IEEE 30th International Conference on Data Engineering Workshops*, pages 223–230. IEEE, 2014.

[15] John C Doyle, Bruce A Francis, and Allen R Tannenbaum. *Feedback Control Theory*. Courier Corporation, 2013.

[16] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *NSDI*, 2014.

[17] Peter X Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *OSDI*, 2016.

[18] Roxana Geambasu, Amit A Levy, Tadayoshi Kohno, Arvind Krishnamurthy, and Henry M Levy. Comet: An active distributed key-value store. In *OSDI*, 2010.

[19] Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. In *SIGCOMM*, 2015.

[20] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. Rdma over commodity ethernet at scale. In *SIGCOMM*, 2016.

[21] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. Tcp≈rdma: Cpu-efficient remote storage access with i10. In *NSDI*, 2020.

[22] Rajendra K Jain, Dah-Ming W Chiu, William R Hawe, et al. A quantitative measure of fairness and discrimination. *Eastern Research Laboratory, Digital Equipment Corporation, Hudson, MA*, 1984.

[23] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter rpcs can be general and fast. In *NSDI*, 2019.

[24] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using rdma efficiently for key-value services. In *SIGCOMM*, 2014.

[25] Anuj Kalia, Michael Kaminsky, and David G Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided rdma datagram rpcs. In *OSDI*, 2016.

[26] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan PC Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.

[27] Leonard Kleinrock. Queueing systems. volume i: theory. 1975.

[28] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. Flash storage disaggregation. In *EuroSys*, 2016.

[29] Chinmay Kulkarni, Sara Moore, Mazhar Naqvi, Tian Zhang, Robert Ricci, and Ryan Stutsman. Splinter: Bare-metal extensions for multi-tenant low-latency storage. In *NSDI*, 2018.

[30] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan MG Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, et al. Swift: Delay is simple and effective for congestion control in the datacenter. In *SIGCOMM*, 2020.

[31] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. In *NSDI*, 2014.

[32] Charles Masson, Jee E Rim, and Homin K Lee. Ddsketch: A fast and fully-mergeable quantile sketch with relative-error guarantees. *Proceedings of the VLDB Endowment*, 12(12).

[33] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *ATC*, 2013.

[34] Christopher Mitchell, Kate Montgomery, Lamont Nelson, Siddhartha Sen, and Jinyang Li. Balancing cpu and network in the cell distributed b-tree store. In *ATC*, 2016.

[35] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pismenny, Liran Liss, Michael Wei, Dan Tsafrir, et al. Storm: a fast transactional dataplane for remote data structures. In *SYSTOR*, 2019.

[36] Francesco Orabona. A modern introduction to online learning. *arXiv preprint arXiv:1912.13213*, 2019.

[37] Dongchul Park, Jianguo Wang, and Yang-Suk Kee. In-storage computing for hadoop mapreduce framework: Challenges and possibilities. *IEEE Transactions on Computers*, 2016.

[38] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 325–341, 2017.

[39] Michael A Sevilla, Noah Watkins, Ivo Jimenez, Peter Alvaro, Shel Finkelstein, Jeff LeFevre, and Carlos Maltzahn. Malacology: A programmable storage system. In *EuroSys*, 2017.

[40] Vishal Shrivastav, Asaf Valadarsky, Hitesh Ballani, Paolo Costa, Ki Suh Lee, Han Wang, Rachit Agarwal, and Hakim Weatherspoon. Shoal: A network architecture for disaggregated racks. In *NSDI*, 2019.

[41] Michael Stonebraker and Greg Kemnitz. The postgres next generation database management system. *Communications of the ACM*, 34(10):78–92, 1991.

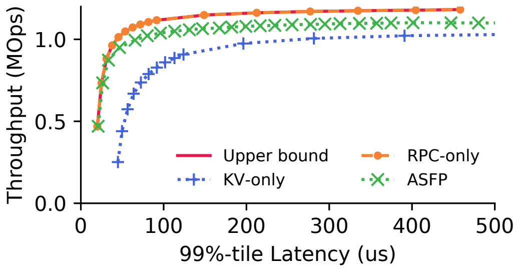
[42] Michael Stonebraker and Ariel Weisberg. The voltdb main memory dbms. *IEEE Data Eng. Bull.*, 36(2):21–27, 2013.

[43] Patrick Stuedi, Animesh Trivedi, and Bernard Metzler. Wimpy nodes with 10gbe: Leveraging one-sided operations in soft-rdma to boost memcached. In *ATC*, 2012.

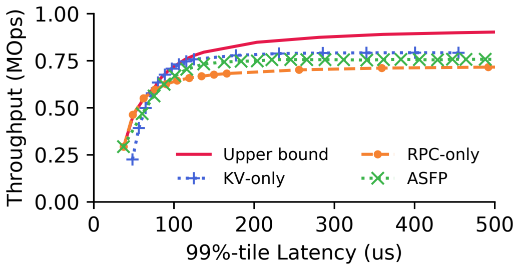
[44] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building an elastic query engine on disaggregated storage. In *NSDI*, 2020.

[45] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

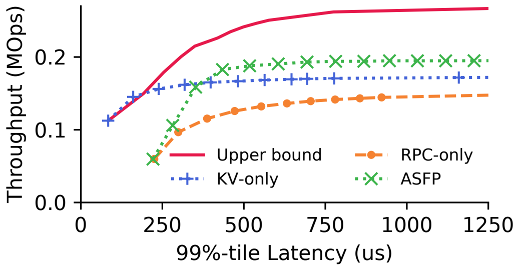
A Supplemental Measurements for Graph Traversal Workload



(a) *Light (100ns/RTT)*



(b) *Medium (1μs/RTT)*



(c) *Heavy (10μs/RTT)*

Figure 17: ***Throughput w.r.t. SLO (99%-tile latency) for graph traversal with four storage accesses per request.***

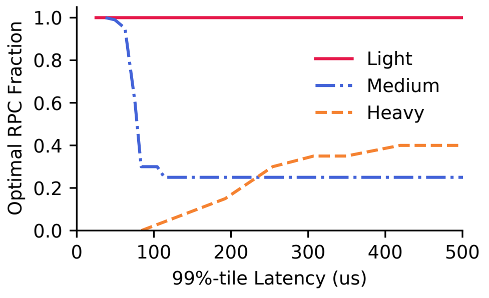
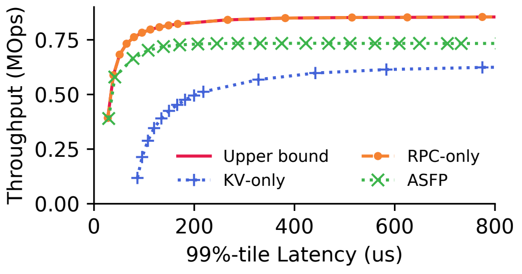
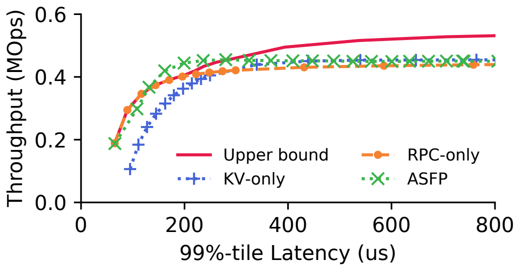


Figure 18: ***Optimal RPC fraction w.r.t. SLO (99%-tile latency) for graph traversal with four storage accesses per request.***

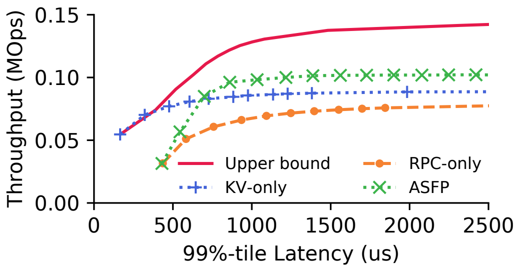
We repeat the same parameter sweep as in §2.2 but with the number of storage accesses per request set to 4 and 8. We vary the RPC fraction from 0 to 1 and measure the overall throughput and end-to-end latency of all requests. In doing so, we obtain the throughput-latency measurements for all possible execution configurations, as shown in Figure 17 and Figure 19. The optimal RPC fraction during these sweeps are shown in Figure 18 and Figure 20. We note that the observation we make in §2.2 still holds in these measurements.



(a) *Light (100ns/RTT)*



(b) *Medium (1μs/RTT)*



(c) *Heavy (10μs/RTT)*

Figure 19: ***Throughput w.r.t. SLO (99%-tile latency) for graph traversal with eight storage accesses per request.***

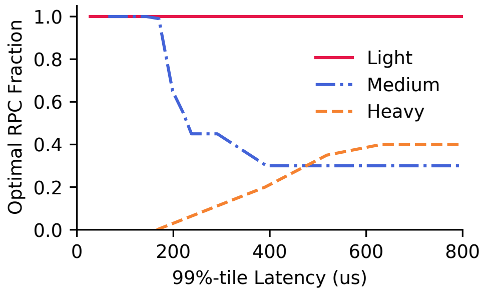


Figure 20: ***Optimal RPC fraction w.r.t. SLO (99%-tile latency) for for graph traversal with eight storage accesses per request.***

B Analysis of Algorithms

B.1 Problem Formulation

We aim to solve the following optimization problem

$$\begin{aligned} \max_X \quad & R \\ \text{s.t.} \quad & T(X,R) \le t_0 \end{aligned} \tag{P}$$

where we assume:

Assumption 1. *Fix X, $F_X(R) := T(X,R)$ is monotonic increasing and twice differentiable.*

Assumption 2. *For any X, there exist R_1 and R_2 such that $\min_X T(X,R_1) \le t_0 \le \min_X T(X,R_2)$.*

B.2 Algorithm I: X-R Dual Loop Control

We adopt the following iterative algorithm to solve the problem:

Algorithm I (X-R Dual Loop Control) For $k = 1, \dots, K$, we alternatively update X_k and R_k by

- $X_k = \arg\min_X T(X,R_k)$;
- $R_{k+1} = R_k + \eta\,(t_0 - T(X_k,R_k))$.

We have the following theorem to characterize the convergence of the above algorithm.

Theorem 1. *Suppose in addition we have,*

1. *$T(X,R)$ is twice differentiable and lower bounded.*
2. *Fix R, $F_R(X) := T(X,R)$ is μ-strongly convex[3], L-smooth[4] and coercive.[5]*
3. *For all X, we have $0 < \alpha \le \frac{\partial T(X,R)}{\partial R} \le \beta$.*

If we set $0 < \eta < \frac{1}{\beta}$, then

$$|R_K - R_*| \le (1-\eta\alpha)^K \cdot |R_0 - R_*|\,.$$

In the following we elaborate the proof for Theorem 1. We begin with introducing a series of lemmas. Let us denote

$$H(R) = \min_X T(X,R).$$

Lemma 1. *For all $R \ne S$,*

$$0 < \alpha \le \frac{H(R)-H(S)}{R-S} \le \beta.$$

Moreover, the above inequality implies $H(R)$ is monotonic increasing and continuous.

Proof. Without loss of generality let $R > S$. Let $X = \arg\min_X T(X,R)$ and $Y = \arg\min_X T(X,S)$. Then

$$\frac{H(R)-H(S)}{R-S} = \frac{T(X,R)-T(Y,S)}{R-S} \begin{cases} \le \frac{T(Y,R)-T(Y,S)}{R-S} = \frac{\partial T(Y,P)}{\partial R} \le \beta, \\ \ge \frac{T(X,R)-T(X,S)}{R-S} = \frac{\partial T(X,Q)}{\partial R} \ge \alpha > 0. \end{cases}$$

Here $P, Q \in (S,R)$ are given by mean-value theorem. $\square$

Lemma 2. *There exists an unique R_* such that $H(R_*) = t_0$. Moreover, this R_* gives the maximum of the original optimization problem.*

Proof. We have already shown that $H(R)$ is monotonic and continuous. Recall that there exists R_1 and R_2 such that $H(R_1) \le t_0 \le H(R_2)$, thus there exists an unique R_* such that $H(R_*) = t_0$.

For any R so that $R > R_*$, we have $H(R) > H(R_*) = t_0$ by monotonicity, thus R does not meet the constraint. Therefore R_* is the maximum of the optimization problem. $\square$

Proof of Theorem 1. With $H(R) := \min_X T(X,R)$, we can rephrase Algorithm I as

$$R_{k+1} = R_k + \eta\,(H(R_*) - H(R_k))\,.$$

Let R_0 be the initialization. We next show the convergence of this iteration.

Case I: $R_0 < R_*$. If $R_k < R_*$, then

$$R_* - R_{k+1} = R_* - R_k - \eta\,(H(R_*) - H(R_k)) \begin{cases} \le R_* - R_k - \eta\alpha\,(R_* - R_k) = (1-\eta\alpha)\,(R_* - R_k) \\ \ge R_* - R_k - \eta\beta\,(R_* - R_k) = (1-\eta\beta)\,(R_* - R_k) \end{cases}$$

that is $0 \le (1-\eta\beta)\,(R_* - R_k) \le R_* - R_{k+1} \le (1-\eta\alpha)\,(R_* - R_k)$. Using this recursion, if $R_0 < R_*$, we have

$$0 \le (1-\eta\beta)^K\,(R_* - R_0) \le R_* - R_K \le (1-\eta\alpha)^K\,(R_* - R_0).$$

Case II: $R_0 > R_*$. If $R_k > R_*$, then

$$\begin{aligned} & R_{k+1} - R_* \\ & = R_k + \eta\,(H(R_*) - H(R_k)) - R_* \\ & = R_k - R_* - \eta\,(H(R_k) - H(R_*)) \\ & \begin{cases} \le R_k - R_* - \eta\alpha\,(R_k - R_*) = (1-\eta\alpha)\,(R_k - R_*) \\ \ge R_k - R_* - \eta\beta\,(R_k - R_*) = (1-\eta\beta)\,(R_k - R_*) \end{cases} \end{aligned}$$

that is $0 \le (1-\eta\beta)\,(R_k - R_*) \le R_{k+1} - R_* \le (1-\eta\alpha)\,(R_k - R_*)$. Using this recursion, if $R_0 > R_*$, we have

$$0 \le (1-\eta\beta)^K\,(R_0 - R_*) \le R_K - R_* \le (1-\eta\alpha)^K\,(R_0 - R_*).$$

To sum up, when $\eta < \frac{1}{\beta}$ (hence smaller than $\frac{1}{\alpha}$), we have

$$|R_K - R_*| \le O\left((1-\eta\alpha)^K\right).$$

$\square$

[3] $f(x)$ is μ-strongly convex, if for all x and y, it holds that $f(x) \ge f(y) + \langle \nabla f(y), x-y\rangle + \frac{\mu}{2}\|y-x\|_2^2$.

[4] $f(x)$ is L-smooth, if for all x and y, it holds that $f(x) \le f(y) + \langle \nabla f(y), x-y\rangle + \frac{L}{2}\|y-x\|_2^2$. In general, if $f(x)$ is twice-differentiable and x is restricted in a bounded domain, then $f(x)$ is L-smooth in that domain, for some finite L.

[5] $f(x)$ is coercive if $f(x) \to +\infty$ as $\|x\| \to +\infty$. A strongly convex and coercive function admits an unique and finite minimum point.

B.3 Algorithm II: R-X Dual Loop Control

Let us take a closer look at the optimization problem (P) under Assumption 1 and Assumption 2. First we observe the maximal must be attended at the boundary

$$T(X,R) = t_0. \tag{6}$$

Second the boundary constraint Eq. (6) implicitly defines a function $R(X)$, where

$$T(X,R(X)) = t_0.$$

We highlight that $R(X)$ is indeed well defined, since under Assumption 1 and Assumption 2, for any X, there exists an unique $R(X)$ that satisfies the boundary constraint.

With the above observations, we may rephrase the optimization problem (P) as

$$\max_X \quad R(X) \tag{P'}$$

where $R(X)$ is implicitly defined by the boundary constraint. In the following we discussion algorithms that solve problem (P').

Our challenge it that we do not have direct access to $R(X)$; instead at each fast loop step, we have an estimation to $R(X)$, denoted as $R_k(x)$, which approximately satisfies

$$T(X,R_k(X)) \approx t_0.$$

In this set up we can perform *stochastic gradient ascent* (SGA, or *online gradient ascent*) for $R_k(X)$. We summarize the algorithm in the following.

Algorithm II (R-X Dual Loop Control) For $k = 1,\dots,K$, we respectively update X_k and R_k by

1. Apply rate control so that the latency approximates SLO, i.e.,
   R_k be such that $T(X_k,R_k) \approx t_0$;
2. Use gradient ascent to search for the optimal X_k, i.e., *$X_{k+1} = X_k + \eta \frac{\mathrm{d}R_k}{\mathrm{d}X}$, where $T(X,R_k) = t_0$, and η is a positive stepsize.*

There is a rich literature for the theory of online learning when $R_k(X)$ is concave, e.g., see [36]. For completeness, we introduce the following theorem to characterize the behavior of the above algorithm.

Theorem 2. *Suppose $R_k(X)$ is concave. Consider the iterates of SGA, i.e.,*

$$X_{k+1} = X_k + \eta \nabla R_k(X_k).$$

Then we have the following bound for the regret

$$\sum_{k=1}^{K} (R_k(X_*) - R_k(X_k)) \le \frac{\|X_1 - X_*\|_2^2}{2\eta} + \frac{\eta}{2}\sum_{k=1}^{K} \|\nabla R_k(X_k)\|_2^2.$$

If in addition we assume $\|\nabla R_k(X)\|_2 \le L$, and set

$$\eta = \frac{\|X_1 - X_*\|_2}{L\sqrt{K}},$$

then

$$\sum_{k=1}^{K} (R_k(X_*) - R_k(X_k)) \le C \cdot \sqrt{K}, \tag{7}$$

where $C := L\|X_1 - X_*\|_2$ is a constant depends on initialization and gradient bound.

Remark. The sublinear regret bound implies SAG behaviors nearly optimal on average: we see this by setting $X_* = \arg\max_X \sum_{k=1}^{K} R_k(X)$, and noticing that

$$\frac{1}{K}\sum_{k=1}^{K} R_k(X_*) - \frac{1}{K}\sum_{k=1}^{K} R_k(X_k) \le O\left(\frac{1}{\sqrt{K}}\right) \to 0.$$

More concisely, in our algorithm, $\{R_k(X)\}_{k=1}^{K}$ corresponds to a sequence of inaccurate estimations to the true implicit function $R(X)$ — even so the theorem guarantees a sublinear regret bound, which implies that our algorithm behaviors nearly as good as one can ever expect under the estimations, no matter how inaccurate they could be.

Furthermore, if for each k, $R_k(X)$ is an *unbiased estimator* to the true concave function $R(X)$, i.e., $\mathbb{E}R_k(X) = R(X)$, then $\bar{X} = \frac{1}{K}\sum_{k=1}^{K} X_k$ converges to the maximal of $R(X)$ in expectation: we see this by choosing $X_* = \arg\min_X R(X)$ and noticing that

$$\begin{aligned}\mathbb{E}\left[R(X_*) - R(\bar{X})\right] &\le \frac{1}{K}\sum_{k=1}^{K} \mathbb{E}\left[R(X_*) - R(X_k)\right] \\ &= \frac{1}{K}\sum_{k=1}^{K} \mathbb{E}\left[R_k(X_*) - R_k(X_k)\right] \\ &\le C \cdot \frac{1}{\sqrt{K}} \to 0.\end{aligned}$$

Proof of Theorem 2. We first notice the following ascent lemma

$$\begin{aligned}&\|X_{k+1} - X_*\|_2^2 \\ &= \|X_k + \eta \nabla R_k(X_k) - X_*\|_2^2 \\ &= \|X_k - X_*\|_2^2 + \eta^2 \|\nabla R_k(X_k)\|_2^2 + 2\eta \langle \nabla R_k(X_k), X_k - X_* \rangle \\ &\le \|X_k - X_*\|_2^2 + \eta^2 \|\nabla R_k(X_k)\|_2^2 + 2\eta \left(R_k(X_k) - R_k(X_*)\right),\end{aligned}$$

where the last inequality is due to the assumption that $R_k(X)$ is concave. Next we re-arrange the terms and take telescope summation,

$$\begin{aligned}
&\sum_{k=1}^{K}(R_k(X_*)-R_k(X_k))\\
&\leq \sum_{k=1}^{K}\frac{1}{2\eta}\left(\|X_k-X_*\|_2^2-\|X_{k+1}-X_*\|_2^2\right)+\sum_{k=1}^{K}\frac{\eta}{2}\|\nabla R_k(X_k)\|_2^2\\
&=\frac{1}{2\eta}\left(\|X_1-X_*\|_2^2-\|X_{K+1}-X_*\|_2^2\right)+\sum_{t=1}^{T}\frac{\eta}{2}\|\nabla R_k(X_k)\|_2^2\\
&\leq \frac{1}{2\eta}\|X_1-X_*\|_2^2+\sum_{k=1}^{K}\frac{\eta}{2}\|\nabla R_k(X_k)\|_2^2,
\end{aligned}$$

which gives the first regret bound.

If further we have $\|\nabla R_k(X)\|_2 \leq L$, then

$$\sum_{k=1}^{K}(R_k(X_*)-R_k(X_k)) \leq \frac{1}{2\eta}\|X_1-X_*\|_2^2+\frac{\eta}{2}L^2K,$$

by setting $\eta = \frac{\|X_1-X_*\|_2}{L\sqrt{K}}$ we obtain

$$\begin{aligned}
\sum_{k=1}^{K}(R_k(X_*)-R_k(X_k)) &\leq \frac{1}{2\eta}\|X_1-X_*\|_2^2+\frac{\eta}{2}L^2K\\
&\leq L\|X_1-X_*\|_2\cdot\sqrt{K}.
\end{aligned}$$

□

Caerus: NIMBLE Task Scheduling for Serverless Analytics

Hong Zhang *UC Berkeley*   *Yupeng Tang* *Yale University*   *Anurag Khandelwal* *Yale University*   *Jingrong Chen* *Duke University*   *Ion Stoica* *UC Berkeley*

Abstract

Serverless platforms facilitate transparent resource elasticity and fine-grained billing, making them an attractive choice for data analytics. We find that while server-centric analytics frameworks typically optimize for job completion time (JCT), resource utilization and isolation via inter-job scheduling policies, serverless analytics requires optimizing for *JCT* and *cost of execution* instead, introducing a new scheduling problem. We present Caerus, a task scheduler for serverless analytics frameworks that employs a fine-grained NIMBLE scheduling algorithm to solve this problem. NIMBLE efficiently pipelines task executions within a job, minimizing execution cost while being Pareto-optimal between cost and JCT for arbitrary analytics jobs. To this end, NIMBLE models a wide range of execution parameters — pipelineable and non-piplineable data dependencies, data generation, consumption and processing rates, etc. — to determine the ideal task launch times. Our evaluation results show that in practice, Caerus is able to achieve *both* optimal cost and JCT for queries across a wide range of analytics workloads.

1 Introduction

Serverless platforms [1–3] fulfill the promise of transparent resource elasticity in the cloud [4–6]. Under the Function as a Service (FaaS) serverless model, users decompose their applications into short-lived stateless *functions* that read and write data from an external storage service. The sub-second startup latencies and virtually unlimited parallelism in FaaS platforms permit fine-grained compute elasticity, while sub-second billing granularities afford cost-efficiency.

These benefits have driven many recent efforts to port *data analytics* applications to serverless platforms [7–18]. Analytics jobs typically comprise multiple stages of execution organized as directed acyclic graphs (DAGs) based on their data dependencies, with each stage comprising several parallel tasks. While traditional server-centric deployments use clusters provisioned with a fixed pool of storage and compute resources to execute these jobs, serverless deployments implement tasks as serverless functions [7–13] that exchange state via external storage [14, 15]. Since analytics workloads typically have widely varying resource needs over time, both across and during job lifetimes [12, 14], server-centric deployments can frequently suffer from resource under- or over-provisioning [12, 14, 19, 20], leading to resource wastage or performance degradation, respectively. In contrast, serverless compute [1–3] and storage [15, 21–24] platforms facilitate *fine-grained scaling* of resources to match application needs, making them an attractive choice for data analytics [7–18].

We find that the shift from server-centric to serverless analytics results in a shift in goals for schedulers in analytics frameworks. Since the FaaS platforms manage allocation of compute resources *across* jobs, schedulers need no longer be concerned with the conventional goals of maximizing cluster resource-utilization and enforcing fairness across jobs via *inter-job* scheduling policies [25–28]. Instead, under the FaaS billing model, schedulers must now consider the *cost* of each job's execution, which is proportional to the aggregated runtimes across its component tasks. This highlights the need for *inter-task* scheduling policies for serverless analytics jobs to minimize both execution cost and job completion time (JCT).

Unfortunately, task-level scheduling policies employed by server-centric analytics today expose a hard-tradeoff between cost and JCT in serverless platforms. Figure 1 shows a simple map-reduce job where reduce tasks consume and aggregate data generated by map tasks. Traditional analytics frameworks [29–32] typically employ one of the two following extremes: (1) a *lazy* approach that launches a reduce task only when all the map tasks have finished (Figure 1 (a)), and (2) an *eager* approach that launches a reduce task as soon as any map task produces data for it to consume (Figure 1 (b)).

Intuitively, the lazy approach is *cost-efficient*: since reduce tasks waste no time waiting for upstream map tasks to generate data, individual task durations (which governs cost in serverless settings) is always minimized. However, its JCT can be far from optimal since there is no *pipelining* of map and reduce task executions. The eager approach, on the other hand, is *JCT-efficient* since it maximally pipelines the execution of map and reduce tasks. However, its can introduce a much higher cost: reduce tasks can waste a lot of time waiting for upstream map tasks to generate data, which increases reduce task durations and, consequently, execution cost. We discuss this example further in §2, but note for now that this trade-off between execution cost and JCT is even more extreme for multi-stage jobs seen in production workloads [27, 28].

Note that in an ideal solution (Figure 1 (c)), a task would be launched *late enough* to minimize task durations (and therefore, execution cost), but *early enough* to minimize JCT. In this work, we propose a NIMBLE scheduling algorithm that builds on this intuition: at its core, NIMBLE scheduling combines the cost-efficiency of lazy and JCT-efficiency of eager approaches and breaks the tradeoff between them (Figure 1 (d)), by scheduling tasks to run *at just the right time*.

Designing such an optimal scheduling strategy, however, is non-trivial. First, a precise description of the pipelinablity across different job stages is crucial to determine the optimal schedule — task-level DAGs typically used for representing

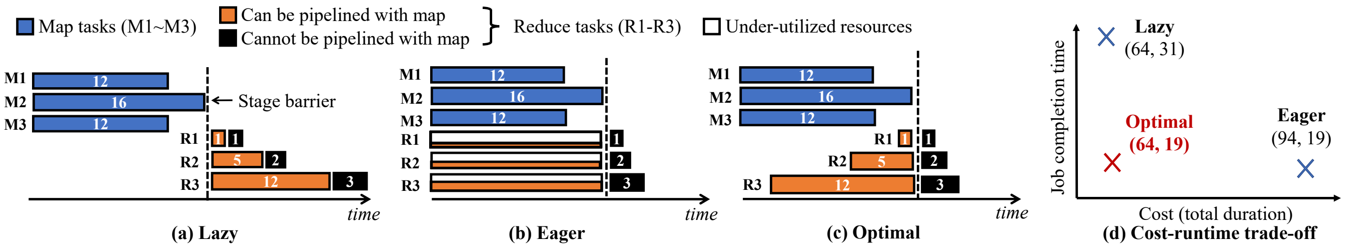


Figure 1: (a, b) Lazy and eager approaches expose a hard trade-off between JCT and cost; numbers within bars correspond to task runtimes. (c, d) Fine-grained scheduling in serverless infrastructures provide opportunities to break this tradeoff with optimal scheduling strategies. The JCT is simply the finish time of the last reduce task, while its cost is calculated as the aggregated durations of all its component tasks.

job executions in existing job schedulers are insufficient. Even for the simple map-reduce example in Figure 1, while parts of reduce task execution can be pipelined with map tasks (orange bars), some parts can only start after map stage finishes (black bars), *e.g.*, when map output must be aggregated at the reduce task before further processing. To this end, we develop a fine-grained *step dependency model* that captures data dependency and pipelinablity information at *sub-task* granularity (§3).

Second, in contrast to the map-reduce example above, tasks in general analytics jobs can have significantly more complex pipeline dependencies. Specifically, a task can consume data from multiple upstream tasks, and tasks across the job's execution DAG may have cascading dependencies. Coupled with time-varying data generation and consumption rates, this makes identifying task launch times for JCT- and cost-efficient job execution challenging. In fact, our analysis shows that even with perfect models for all of the above constraints, it is *impossible* for a task scheduling algorithm to always be able to optimize both execution cost and JCT for arbitrary analytics jobs. Fortunately, we show it *is* possible for a scheduling algorithm to be cost optimal, while being *Pareto-optimal* between execution cost and JCT. We realize this in NIMBLE, a scheduling algorithm that carefully models data produce and consume rates across stages, computes launch times for tasks across them based on both inter- and intra-task data dependencies, and schedules tasks greedily across dependent stages (§4).

Finally, we incorporate the NIMBLE algorithm into Caerus, a new fine-grained task-level scheduler for serverless analytics frameworks (§5). Caerus translates the theory developed for NIMBLE to practice, by extracting step dependencies from user queries via a step annotation API, and estimating NIMBLE algorithm inputs using a combination of job execution histories and information profiled at runtime. Caerus easily integrates with existing serverless analytics frameworks [11, 12, 14] — we implement Caerus in a prototype serverless SQL engine built atop Locus [14], and evaluate its performance on AWS Lambda for a wide range of analytics workloads including TeraSort, TPC-DS and Big-Data Benchmark (§6). Our results show that in practice, Caerus optimizes *both* cost and JCT, outperforming the lazy approach by 1.08–2.2× in JCT, and eager approach by 1.21–1.57× in cost across these workloads.

In summary, we make three main contributions:

- Formulation of a new task-level scheduling problem for serverless analytics to minimize execution cost and JCT. We show that schedulers used in server-centric frameworks expose a hard tradeoff between cost and JCT (§2).
- Design of a new NIMBLE scheduling algorithm, that launches each task in a job at just the right time to optimize *both* cost and JCT. NIMBLE employs a new *step model* to capture sub-task level pipelinablity and data dependencies, and guarantees cost optimality while being *Pareto-optimal* between cost and JCT for any analytics job (§4).
- Design, implementation and evaluation of Caerus, a fine-grained task-level scheduler for serverless analytics frameworks that enables NIMBLE scheduling in practice (§5, §6).

2 Motivation

In this section, we provide a brief background on server-centric and serverless analytics (§2.1). We then describe how serverless analytics introduces a new task scheduling problem (§2.2) and new opportunities to address it (§2.3).

2.1 Background

Server-centric Analytics. Traditional server-centric deployments for data analytics [30, 31, 33–35] operate atop a fixed pool of compute and storage resources, *e.g.*, clusters of provisioned servers or pools of provisioned virtual machines (VMs)[1]. Consequently, such deployments employ a cluster-wide job scheduler to efficiently share the fixed resource-pool among multiple jobs with three key goals: minimizing job runtime, maximizing resource utilization and ensuring resource isolation (or fairness) across jobs. Given the resource demands of each job, the scheduler achieves all or a subset of goals via *inter-job* (i.e., job-granularity) scheduling policies [25–28].

Within a job, the execution is broken down into a DAG of stages, each comprising multiple parallel tasks (see Figure 1 for an example). A task scheduler launches tasks across the compute resources allocated to the job. Tasks in a stage read their initial input from and write their final output to persistent storage (*e.g.*, HDFS [37]), while data exchange between consecutive stages occurs over the network (*e.g.*, shuffle, broadcast, etc.). Existing frameworks typically apply one of two popular approaches to decide *when* to launch tasks: (i) *lazy* (*e.g.*, Spark [30]), which launches a task only when *all*

[1]One can add/remove VMs to scale VM clouds, but at coarse time granularities, *e.g.*, resizing an AWS EMR cluster takes $\sim 6-45$ minutes [36].

tasks in upstream stages have completed, and (ii) *eager* (*e.g.*, MapReduce Online [29]), which launches a task as soon as *any* output from its upstream stages is ready.

Serverless Analytics. In serverless platforms, users no longer provision or manage resources: this is the cloud provider's responsibility. Users simply pay for resources they use. Serverless compute platforms [1–3] allocate and charge for compute resources at function invocation granularity: invoking more functions permits scaling up at a higher cost, and vice versa.

Existing approaches to serverless analytics deploy tasks within a stage as serverless function invocations. Since cloud providers disallow direct communications between serverless functions [7, 11, 14], data is exchanged between functions via external storage [8, 14, 15]. A job is charged for both function execution and external storage, with the former typically dominating the cost[2]. With *sub-second granularity billing* for serverless functions, the job execution cost is proportional to the cumulative runtimes across all tasks of the job.

2.2 Serverless Scheduling: A New Problem

Since the cloud provider is responsible for resource management in serverless platforms, user goals in serverless analytics are different from server-centric deployments. In particular, while minimizing JCT is still a primary goal, metrics like resource utilization and isolation are now the onus of the cloud provider. Instead, the user must now optimize the *cost* of each job's execution, which is proportional to the cumulative task runtime as outlined in §2.1. This shift in goals exposes a new task-level scheduling problem for serverless analytics:

Problem Statement: *Given the execution plan for an analytics job comprising tasks with arbitrary dependencies, can we find a task-level schedule that optimizes for both job execution cost and JCT on a serverless platform?*

Limitations of existing approaches. As we saw in §1, the existing server-centric lazy and eager task scheduling approaches, when applied for serverless analytics, expose a hard tradeoff between cost and JCT. Recall the job execution example in Figure 1, which comprises a map and a reduce stage, each with three tasks — each bar represents the execution of one task over time (numbers in bars show task runtimes).

The lazy approach (Figure 1 (a)) is *cost-optimal* in the serverless model, with a cost of 64 units[3] — starting reduce tasks any later would not affect their runtime (and therefore, cost), while starting them sooner can cause them to stall for more data to be generated by upstream map tasks, increasing cost. However, the lazy approach also leads to high JCT (31 units), since it does not pipeline the execution of map and reduce tasks at all. Similarly, eager scheduling (Figure 1 (b)) is *JCT-optimal* (19 units) since the first part of reduce execution

[2]Cost of AWS Lambda execution is ∼\$0.20/hour [38], while Amazon S3 storage is ∼\$0.02/GB/month [39], with no data transfer cost between them.

[3]The cost is computed as the cumulative sum of the runtimes of the tasks in the job, and assuming unit cost per unit runtime.

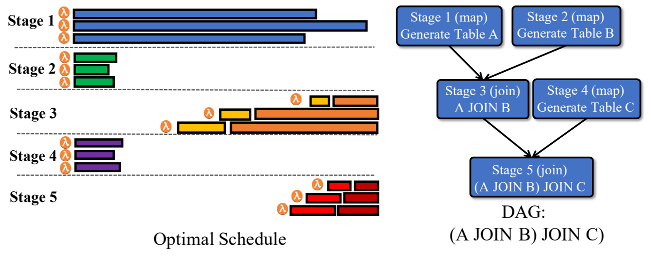

Stage 1
Stage 2
Stage 3
Stage 4
Stage 5
Optimal Schedule
Stage 1 (map) Generate Table A
Stage 2 (map) Generate Table B
Stage 3 (join) A JOIN B
Stage 4 (map) Generate Table C
Stage 5 (join) (A JOIN B) JOIN C
DAG: (A JOIN B) JOIN C)

Figure 2: **Optimal schedule (left) and execution DAG (right) for a multi-stage job.** See §2.3 for details.

(orange bar) can be completely pipelined with the map stage. However, its cost is significantly higher (94 units), since reduce tasks often wait for data to be generated by upstream map tasks, increasing their runtime, and therefore, cost.

This tradeoff can be much more severe for multi-stage jobs. Production traces from Microsoft [27, 28] show that jobs in their workloads have 13 and 121 stages at 50th and 95th percentiles, making it likely for them to have far more opportunities for pipelining tasks across stages. Ignoring these opportunities (*e.g.*, following the lazy approach) would lead to JCTs that are significantly longer than optimal. On the other hand, jobs can also have heavy skew in task runtimes [40–42] — 10% of tasks take more than 10× the median task duration in Microsoft's workloads [40]. Starting tasks across all stages early to maximize pipelining (*e.g.*, following the eager approach) would force most downstream tasks to stall due to slower upstream tasks, significantly increasing execution cost.

2.3 Opportunities & Challenges

New opportunities in serverless scheduling. Serverless frameworks provide new opportunities to break the hard tradeoff between cost and JCT exposed by lazy and eager solutions — on-demand invocation of functions at fine-grained timescales permits the design of *fine-grained task-level schedulers*. Figure 1 (c) shows the optimal schedule for the job in Figure 1 — with fine-grained scheduling, it is possible to achieve such a schedule by launching each task *at just the right time*, minimizing both cost (64 units) *and* JCT (19 units) (Figure 1 (d)).

Moreover, these gains are likely to be even more significant in production workloads comprising multi-stage jobs with complex stage dependencies. For example, Figure 2 shows a multi-stage SQL job which performs join across three tables (A, B and C) using shuffle hash join (SHJ) algorithm [43]. The figure shows the job's execution plan as a DAG of stages on the right, and the corresponding optimal task schedule on the left. The optimal schedule can efficiently pipeline all the five stages in this multi-stage join example, resulting in much higher gains in JCT and cost than for simple two stages map-reduce jobs.

Challenges. Figure 2 also indicates that calculating the optimal launch time for each task is non-trivial due to a number of reasons. First, a task may include multiple parts, where each part may or may not be pipelineable with some part of one of its upstream stages. In Figure 2 (left), Stage 3 is composed of two parts. The first part, which reads Table B from Stage 2 and uses it to build a hash table, can be pipelined with Stage 2 execution. The second part, which reads Table A from Stage 1 and performs online join with the hash table constructed in the first part, can pipelined with Stage 1 execution. Second, the runtime of one task depends on the processing rate of all tasks in its previous stage, and these dependencies cascade to upstream stages. In Figure 2, the execution of Stage 5 depends on Stage 3 and Stage 4, and Stage 4 is further determined by Stage 1 and Stage 2. As such, the first challenge lies in identifying parts of the execution that can be pipelined, and the dependencies between such pipelinable components — we address this in §3. The second challenge lies in using this information to determine ideal launch times for tasks in jobs with complex DAGs, which we address in §4.

Why serverless? Intuitively, the fine-grained task-level scheduling shown in Figures 1 (c) and 2 can also be extended to server-centric settings to optimize average JCT. Moreover, the reduction in per-job resource usage (i.e., cost in serverless settings) enabled by this approach may improve resource utilization via bin-packing more jobs onto the same number of servers. However, while cost improvements in serverless analytics are obvious, achieving improvements in resource utilizations with theoretical guarantees in server-centric deployments is not straightforward, since it is unclear how the resources saved by delaying task launch times can be utilized by other jobs. Specifically, the optimal in Figures 1 (c) and 2 is likely to create staggered task launch times across stages to optimize *each individual* job, and they may not be optimal for bin-packing *across* jobs. Thus, while the clear decoupling from inter-job resource allocation ensures cost and JCT-optimality, extending it to server-centric settings for optimal JCT and resource utilization requires a careful co-design of inter- and intra-job scheduling. We leave this study to future work.

3 Step Dependency Model

As discussed above, a key challenge in identifying ideal task launch times for a job is modeling pipelineable and non-pipelineable dependencies across tasks. In this section, we discuss how we model such dependencies and the flow of data across them, using a new *step dependency model*. We employ this model to design our NIMBLE scheduling algorithm in §4.

Stage dependencies in traditional analytics. As outlined in §2.1, job execution in traditional analytics frameworks [27, 28, 30, 34] is represented as a DAG, where nodes are execution *stages* (comprised of multiple parallel tasks) and edges denote *data dependencies* between them. Figure 3 (a, left) shows the DAG for the map-reduce example from Figure 1, while Figure 3 (b, left) shows a SQL query that performs shuffle hash join (SHJ) on tables generated by two map stages.

Unfortunately, the stage model is not fine-grained enough to capture the information required to determine the ideal launch times for tasks in serverless analytics jobs. To see why,

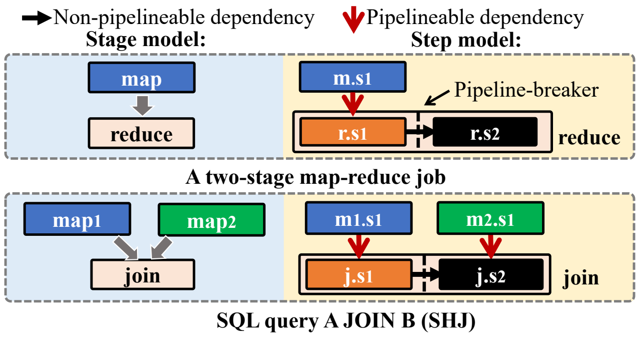


Figure 3: **Stage *vs.* step dependency model** for (a) map-reduce job, and (b) SQL query that joins two tables A and B after applying a map function on each. In the step model, red arrows show dependencies across steps that can be pipelined, while black arrows show dependencies that prevent pipelining. See §3 for details.

consider the map-reduce example from Figure 3 (a, right)[4], where the reduce stage (and therefore, all tasks in the stage) has two distinct parts, shown as orange and black boxes. While the first part (`r.s1`), where reduce tasks read map data, can be pipelined with map execution (`m.s1`), the second part (`r.s2`), where the reduce tasks aggregate and output data, cannot — since final aggregation can only occur after all map data has been read. Clearly, stage dependencies, shown in Figure 3 (a, left), cannot capture such fine-grained information regarding pipelineable and non-pipelineable components of task, nor capture the data dependencies between them. This information is crucial in determining the optimal start time for reduce tasks — early enough to maximally overlap `r.s1` with `m.s1`, but not too early, since pipelining `r.s2` with `m.s1` is impossible.

Modeling pipeline dependencies using *steps*. To precisely model how stages can be pipelined, we refine the stage model into a fine-grained step model to precisely describe how job execution can be pipelined across stages. In our model, the stages are decomposed into one or more *steps*, which are separated by *pipeline breakers* within the stage — operators that produce their first output only after all input have been processed. Pipeline breakers create barriers in execution, demarcating stretches of execution that cannot be pipelined with each other. As such, steps within a stage must be executed sequentially, since pipeline breakers prevent subsequent steps from starting before its upstream step finishes. Across stages, however, steps with data dependencies between them can be pipelined. As a concrete example, consider the step model for the map-reduce job in Figure 3 (a, right) — `m.s1` corresponds to the single step in map stage, while `r.s1` and `r.s2` correspond to two steps in the reduce stage, with a pipeline breaker separating them. The step `r.s1` which consumes data can be pipelined with the upstream step `m.s1` in the map stage that generates the data. We refer to such cross-stage pipelineable step pairs (*e.g.*, (`m.s1`, `r.s1`)) as *parent-child* step pairs. Note that the while above description focuses on the decomposition of a stage into steps, each task within the stage shares the same

[4]This is the same example as the one depicted in Figure 1.

step-level decomposition — we will use the term step to refer to parts of a stage or its tasks interchangeably and clarify the distinction whenever needed.

Figure 3 (b) contrasts the step and stage DAGs for a simple join query. Each of the two map stages comprise a single step, while the hash join stage is divided into two steps. The step `j.s1` reads the left table (Table A generated by `m1.s1`) to create a hash table of unique entries, while step `j.s2` reads the right table (Table B generated by `m2.s1`), joins it with the hash table and writes the output. Each of the two steps can be pipelined with their parent steps (the two map stages), but these two steps have to be executed sequentially within the join stage, since the hash table must be created before the second join step can proceed (pipeline-breaker).

We discuss the details of how the step dependencies can be extracted from user code in §5, but note for now that this model is expressive enough to capture the pipeline dependencies across a wide range of evaluated analytics applications (§6).

Modeling flow of data across steps. We now describe parameters that are used to model the flow of data across steps in the step dependency model. While we discuss how these are estimated in §5, we note for now that these parameters are used as inputs to the NIMBLE algorithm. Consider a stage comprising n steps, `s1`-`s`n, some of which may have a parent step, while some may not. If step `s`i receives data from a parent step, then (1) parent *produce rate* (r_p) is the aggregated data output rate across all tasks of its parent step (referred to as *produce rate* for brevity); and (2) *full consume rate* (r_c) is the rate at which data can be read and processed by step `s`i when there is sufficient data for it to consume. If step `s`i does not have a parent step, then its execution duration $d_{\mathtt{s}i}$ is independent of when the task is launched, allowing us to model $d_{\mathtt{s}i}$ as a constant.

Since the produce rate is determined by the aggregate data output rate across all upstream tasks, each with potentially different start and end times, we model r_p as an arbitrary function of time t. Note that the cumulative area under the $r_p(t)$ curve corresponds to the total input data for the step under consideration; we denote this as P. The full consume rate, on the other hand, is tied to how fast the step can read and process data, and we found it to be stable throughout a the step's execution in our evaluation (§6), allowing us to model r_c as a constant. Note that the parent step may not always produce data as fast as it can be consumed, i.e., the *actual consume rate* (r_{ac}) for the step may be lower than r_c.

4 NIMBLE Scheduling

Armed with the step dependency model, we are now ready to describe our NIMBLE scheduling algorithm. NIMBLE builds on the intuition outlined in §2.3, and combines the cost-optimality of lazy and JCT-optimality of eager approaches to schedule tasks in an analytics job to run at *just the right time*. We first describe NIMBLE scheduling for a simple two-stage map-reduce job (§4.1), and then extend it to general analytics jobs with arbitrary execution DAGs (§4.2).

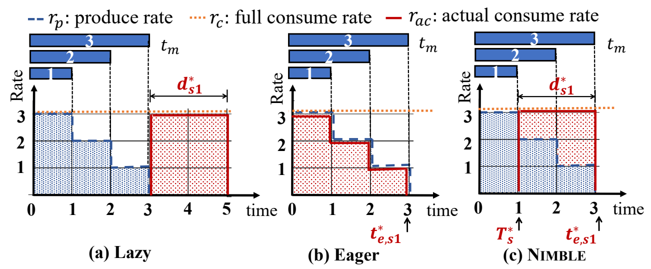


Figure 4: **Optimal launch time for a two-stage map-reduce job.** (a) The total volume of data to be consumed by the reduce step `r.s1` ($P = 6$) is the area under the produce rate ($r_p(t)$) curve. The lazy approach allows us to compute the optimal task runtime ($d^*_{\mathtt{s1}}$) as $P/r_c = 2$. (b) The optimal task finish time ($t^*_{e,\mathtt{s1}} = 3$) is obtained by emulating the eager approach, where the finish time is the maximum of P/r_c ($= 2$) and the map finish time t_m ($= 3$). (c) The optimal launch time ($T^*_s = 1$) is computed as the difference of the optimal finish time and optimal duration. See §4.1 for details.

4.1 NIMBLE for Two-stage Map-Reduce

Consider the step model for the simple two-stage map-reduce job in Figure 3 (a). Note that the JCT of the job is the same as the finish time of the last reducer, and the total cost of the job is proportional to the aggregated duration of all map and reduce tasks. As such, optimizing for the finish time and execution duration of individual tasks also ensures optimality for JCT.

Since map tasks do not have any upstream dependencies, their execution duration is independent of their launch times, and only depends on how fast they can read data from persistent storage and process it. Meanwhile, optimal finish time for map tasks can be achieved by launching them as early as possible (at $t = 0$). On the other hand, due to the parent-child step dependency between the map and reduce tasks (Figure 3 (top)), the *data consumption* in `r.s1` step of reduce tasks can be pipelined with the *data generation* in step `m.s1` of map tasks for minimizing reduce task finish times and execution durations. In particular, a reduce task should be launched early enough to ensure `r.s1` overlaps with `m.s1` as much as possible to minimize finish time, but late enough to ensure that it can always consume data at full rate throughout its execution without stalling, to optimize cost. Our NIMBLE scheduling approach can always find such a "perfect" launch time using the following three steps (Figure 4):

Step 1: Calculate optimal task duration D^*. Since step `r.s2` can only start after `r.s1` finishes, the optimal duration D^* of a reduce task is $d^*_{\mathtt{s1}} + d^*_{\mathtt{s2}}$, where $d^*_{\mathtt{s1}}$ and $d^*_{\mathtt{s2}}$ are the optimal durations of steps `r.s1` and `r.s2` respectively. Note that since `r.s2` does not have a parent step, its duration is independent of when the reduce task is scheduled. As such, the optimal duration D^* depends only on step `r.s1`.

Recall from §2.2 that the lazy approach always ensures optimal duration for reduce tasks — since the entire input is available before the reducer starts, `r.s1` can always consume the input data at consume rate r_c without ever stalling. As such, $d^*_{\mathtt{s1}}$ is simply P/r_c, where P is the total amount of input data for `r.s1`. In Figure 4 (a), $P = 6$ is the area under the curve

$r_p(t)$, which gives $d^*_{\texttt{s1}} = 3$.

Step 2: Calculate optimal finish time T^*_e. The optimal finish time T^*_e for a reduce task is simply $t^*_{e,\texttt{s1}} + d^*_{\texttt{s2}}$ where $t^*_{e,\texttt{s1}}$ is the optimal finish time of `r.s1`. Again, since duration $d^*_{\texttt{s2}}$ is independent of when the task is launched, T^*_e depends only on when step `r.s1` finishes.

We leverage the eager strategy of starting the reduce task at $t = 0$ to compute the optimal step finish time $t^*_{e,\texttt{s1}}$ — intuitively, starting the task any sooner cannot reduce the step finish time any further. Note that since all the map tasks are started at $t = 0$, the produce rate r_p is non-increasing in time. Consequently, if the full consume rate r_c of the reduce task is lower than the *average* produce rate, then the finish time of the step will be bottlenecked by r_c, i.e., $t^*_{e,\texttt{s1}} = P/r_c$. On the other hand, if r_c is higher than the average produce rate, the bottleneck shifts to r_p, and the reduce task can only finish when the map tasks finish generating data at time t_m. Figure 4 (b) shows the latter scenario, where $r_c = 3$ is higher than the average produce rate $(= 2)$, and therefore $t^*_{e,\texttt{s1}} = t_m = 3$.

Step 3: Calculate optimal launch time T^*_s. We find that launching the reduce task at $T^*_s = T^*_e - D^*$, where D^* and T^*_e are computed via the lazy (Step 1) and eager (Step 2) approaches, respectively, ensure that the task is optimal in both execution duration and finish time. This is shown in Figure 4 (c), where starting the reduce task at $T^*_s = 3 - 2 = 1$ ensures optimal duration $(D^* = 2)$, as well as finish time $(T^*_e = 3)$. At first glance, this may seem obvious, since D^* and T^*_e already correspond to optimal task duration and finish time, respectively. But we note that since D^* and T^*_e were computed for two separate approaches, it is not obvious if an approach that starts the task at $T^*_s = T^*_e - D^*$ will always ensure the task takes exactly D^* time to finish. Fortunately, for two-stage map-reduce jobs, we have the following theorem:

Theorem 4.1 *For a reduce task, we can always achieve both optimal execution duration and finish time by launching it at time $T^*_s = T^*_e - D^*$, where T^*_e is the optimal finish time and D^* is the optimal duration computed using Steps 1 and 2 above.*

Proof Since the duration of step `r.s2` is independent of when the reduce task is scheduled, we only need to prove the optimality of finish time and duration for step `r.s1`.

We first show that we can always achieve optimal finish time $t^*_{e,\texttt{s1}}$ if we launch the reduce task at time $T^*_s = T^*_e - D^*$. We prove this by contradiction: assume that a reduce task that is started at T^*_s does not finish executing its first step `r.s1` at $t^*_{e,\texttt{s1}}$. This must be because at some time point $\in [T^*_s, t^*_{e,\texttt{s1}}]$, the task was unable to consume data at full consume rate r_c. We denote the last time instant where this was true as t'. Note that the data produced until time t' (say, $P_{t\le t'}$) must be less than the data that can be consumed by time t' at full consume rate r_c, i.e., $P_{t\le t'} < (t' - T^*_s) \times r_c$. Since the total amount of data produced is $P = d^*_{\texttt{s1}} \times r_c$, the data produced *after* t' must be $P_{t>t'} = P - P_{t\le t'} > (t^*_{e,\texttt{s1}} - t') \times r_c$, and the reduce task will take more time than $(t^*_{e,\texttt{s1}} - t')$ to consume it (since it can consume data at a rate no faster than r_c).

Note that the data produced after t', $P_{t>t'}$, is independent of the reduce task's launch time. This implies that regardless of how early the task is launched, no solution could have achieved optimal finish time $t^*_{e,\texttt{s1}}$ for the step `r.s1`. However, this contradicts with the fact that the eager solution can achieve the optimal finish time by launching the task at $t = 0$. Therefore, our initial assumption must have been false: a reduce task that is started at T^*_s *does* finish executing its first step `r.s1` at $t^*_{e,\texttt{s1}}$.

Proving $T^*_s = T^*_e - D^*$ results in optimal task duration is then trivial: since step `r.s1` finishes at $t^*_{e,\texttt{s1}}$ with start time T^*_s, the corresponding duration $T^*_s - t^*_{e,\texttt{s1}}$ will always be $d^*_{\texttt{s1}}$. ■

Note that the T^*_s for different reduce tasks may be different, since the produce rate r_p to different reduce tasks may vary (*e.g.*, due to data skew). Recall that the finish time of the job is the same as the finish time of the last reduce task, and the total cost of the job is proportional to the aggregated duration of all tasks. As such, Theorem 4.1 shows that we can simultaneously achieve both optimal cost and finish time for the entire job, as long as each reduce task is optimal in duration and finish time, i.e., is launched at T^*_s.

4.2 NIMBLE for General Analytics

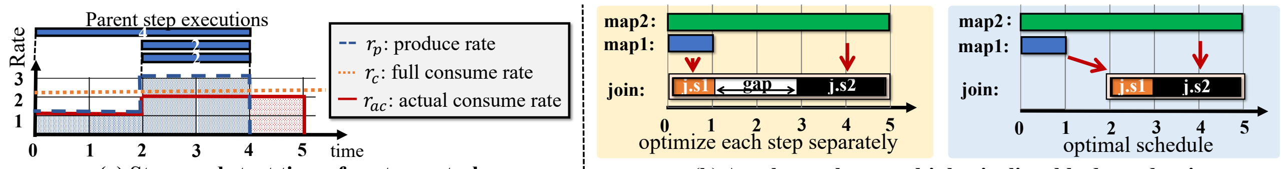
We now extend our analysis to general analytics jobs. We first outline the steps in computing the optimal launch time for tasks in jobs with arbitrary execution DAGs, and then describe how NIMBLE scheduling can be generalized to such DAGs.

4.2.1 Optimal launch time for individual tasks

General analytics jobs with arbitrary execution DAGs introduce two main challenges in determining the optimal task launch time as defined in §4.1. First, unlike two-stage map-reduce jobs, the start times of a step's parent steps need not start at $t = 0$ and can be staggered in time, as shown in Figure 5 (a). This breaks our assumption of a nonincreasing $r_p(t)$ from §4.1, and necessitates a more nuanced treatment of the eager approach to compute the optimal task finish time.

Second, unlike two-stage map-reduce jobs, general analytics jobs, a stage may contain multiple parent-child step pairs, *e.g.*, the join stage in Figure 3 has two steps, and each step has a parent step from a different map stage. For such dependencies, we find that optimally overlapping each parent-child step pair is insufficient to ensure optimal task duration and finish time. Specifically, the optimal task launch time depends not only on the *inter-stage* dependency between parent-child step pairs, but also on the *intra-stage* dependency between steps in the same stage. Figure 5 (b, left) shows a join example where the optimal launch for each step in the task is computed independently. Although each child step is optimally pipelined with its parent, the gap between their execution corresponds to time where no useful work is done, resulting in sub-optimal task duration and, therefore, cost of execution. Figure 5 (b, right) shows how this can be avoided by *deferring* the start time of the first step.

We exploit the above insights to extend NIMBLE scheduling approach from §4.1 to general analytics jobs. We consider the



(a) Staggered start time of upstream tasks **(b) A task may have multiple pipelineable dependencies**

Figure 5: **NIMBLE scheduling for general analytics (§4.2)** (a) Unlike two-stage map-reduce, the produce rate for a task may not be nonincreasing, since the parent step in different tasks can have different start times. (b) Unlike two-stage map-reduce, tasks may have multiple pipelineable dependencies, which requires careful handling to ensure optimal task duration. See §4.2.1 for details.

general case where a task comprises n steps, s1-sn, and make two assumptions to simplify our analysis: (i) each step in a task has at most one parent step, and (ii) steps within a task are executed sequentially in a fixed order. Both assumptions hold for a wide range of analytics jobs, including the join example above, and all of our evaluated workloads (§6). Similar to two-stage map-reduce (§4.1), the optimal launch time T_s^* for a task in the execution DAG is calculated in three steps:

Step 1: Calculate optimal task duration D^*. The optimal task duration D^* is simply the sum of individual optimal step durations $d_{\mathtt{s}i}^*$, $1 \leq i \leq n$. As in §4.1, the duration for steps without a parent is independent of task launch time, while the optimal duration for steps with a parent is computed using the lazy approach, i.e., P/r_c for the corresponding step. In Figure 5 (a), $P = 8$ and $r_c = 2$, so $d_{\mathtt{s}i}^* = 4$.

Step 2: Calculate optimal finish time T_e^*. Since T_e^* is bound by the finish time of the last step sn, we first compute the optimal finish time of a step as computed by the eager approach, similar to §4.1. As noted earlier, however, unlike two-stage map-reduce where the parent step across all the map tasks start at time $t = 0$, the parent step across different tasks in a general DAG may start and end at arbitrary times. This is depicted in Figure 5 (a) where the parent step across two upstream tasks start at time $t = 2$, while the third starts at $t = 0$. Consequently, the produce rate is no longer non-increasing. As such, the optimal step finish time (based on the eager approach) can only be determined by tracking the actual consume rate r_{ac} over time. In the example, r_{ac} is bound by r_p $(= 1)$ between $t = 0-2$, lower than r_p $(= 3)$ and bound by r_c $(= 2)$ between $t = 2-4$, and equals to r_c $(= 2)$ between $t = 4-5$ to clear the surplus data generated between $t = 2-4$. As such, the finish time yielded by the eager approach is 5.

In order to generalize the above example, we discretize time into slots t_1, t_2, ..., t_m, such that the produce rate is constant within a time slot. We introduce a new function $S(t_i)$ to identify time slots where the step accumulates surplus data, i.e., $S(t_i) = 0$ if all the input data produced until t_i has been consumed by time t_i, and 1 otherwise. It is easy to see that when there is no surplus data ($S(t_i) = 0$), the actual consume rate $r_{ac}(t_i)$ is upper-bounded by the produce rate $r_p(t_i)$. When there is surplus data ($S(t_i) = 1$), the actual consume rate increases to the full consume rate ($r_{ac} = r_c$) to clear the surplus. Formally,

$$r_{ac}(t_i) = \begin{cases} \min\left(r_p(t_i), r_c\right) & \text{if } S(t_i) = 0 \\ r_c & \text{if } S(t_i) = 1 \end{cases} \quad (1)$$

For each t_i, we can calculate $S(t_i)$ based on $S(t_{i-1})$, $r_{ac}(t_{i-1})$ and $r_p(t_{i-1})$, and $r_{ac}(t_i)$ based on Equation 1. The time slot t_n where the cumulative data consumed so far equals P, the total input data for the step, corresponds to the optimal finish time; we formally prove optimality in Appendix A.

Unlike the two-stage map-reduce job in §4.1, we have to consider one more constraint — step si can only start after step si-1 has finished, i.e., the finish time of step i is no less than $t_{e,\mathtt{s}i\text{-}1}^* + d_{\mathtt{s}i}^*$. Let the optimal step finish time for si as computed above (which only considers its parent step) be $t'_{e,\mathtt{s}i}$, then the actual optimal finish time of step i is:

$$t_{e,\mathtt{s}i}^* = \begin{cases} \max\left(t'_{e,\mathtt{s}i}, t_{e,\mathtt{s}i\text{-}1}^* + d_{\mathtt{s}i}^*\right) & \text{if } \mathtt{s}i \text{ has a parent} \\ t_{e,\mathtt{s}i\text{-}1}^* + d_{\mathtt{s}i}^* & \text{otherwise} \end{cases} \quad (2)$$

We compute the optimal task finish time by iteratively calculating the optimal finish time for each step s1−sn. Figure 5 (b) shows an example for this computation: the task finish time equals the $t'_{e,\mathtt{s}2}$ $(= 5)$, since $t_{e,\mathtt{s}1}^* + d_{\mathtt{s}2}^*$ $(= 1 + 2)$ is smaller.

Step 3: Calculate optimal launch time T_s^*. As in §4.1, the optimal launch time is computed as $T_s^* = T_e^* - D^*$. Consider the example in Figure 5 (b); the optimal launch time is calculated as $T_s^* = T_e^* - D^*$ $(= 5 - 3)$ for the two steps. Compared to Figure 5 (b, left), doing so automatically delays the first step and removes the gap (Figure 5 (b, right)).

Indeed, Theorem 4.1 extends to general analytics jobs:

Theorem 4.2 *For a task in an analytics job with an arbitrary execution DAG, given the execution (produce rate) of all its parent steps, we can always achieve both optimal execution duration and finish time by launching it at $T_s^* = T_e^* - D^*$, where T_e^* is the optimal finish time and D^* is the optimal duration computed using Steps 1 and 2 above.*

We defer the formal proof to Appendix A, but note here that it employs induction on the number of steps: we assume the statement holds for a task with $n-1$ steps, and use Theorem 4.1 to show that it still holds on adding one more step.

4.2.2 Optimal schedule for the entire job

Algorithm 1 shows NIMBLE scheduling for the entire job based on Theorem 4.2. Stages in the job are scheduled iteratively based on their dependencies: a stage is scheduled when all of its parent stages in the execution DAG have been scheduled. For each task within a scheduled stage, we first calculate the produce rate from its parent stages, and then calculate its optimal launch time as described above.

Algorithm 1 ensures that *each task* achieves optimal duration and finish time given its parent execution (due to Theo-

Algorithm 1 NIMBLE scheduling for a job

```
Launch all stages with no parent stages.
$\mathcal{U} \leftarrow$ Set of unscheduled stages
while $\mathcal{U} \neq \emptyset$ do
    for each stage $S \in \mathcal{U}$, whose parent stages are scheduled do
        for each task in stage $S$ do
            Calculate $r_p$ for each using parent stage schedules
            Calculate $T_e^*$ and $D^*$ based on $r_p$ and $r_c$ of each step
            Calculate $T_s^* = T_e^* - D^*$
```

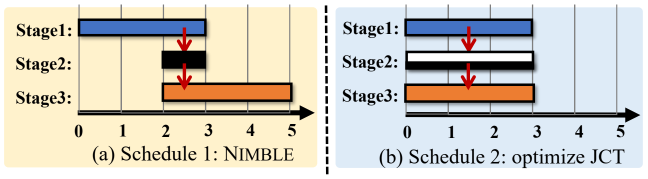


Figure 6: **Example of a job that cannot achieve both (a) optimal cost and (b) finish time simultaneously.** Each stage comprises a single task/step. Stage 2 has a produce rate of 1 and consume rate of 3. Stage 3 has a produce rate of 3 and consume rate of 1.

rem 4.2). However, it still leaves the question: does the algorithm also ensure optimal finish time and cost for the *entire job*? We find that the answer is in the affirmative for jobs with DAGs of depth two, including the map-reduce and SQL jobs in Fig 3. Intuitively, since the stages in the first level of the DAG do not have parent steps, their optimal start time is $t = 0$. As such, given the execution of the stages in the first level, Theorem 4.2 ensures optimal duration and finish time for the stages in the second level of the DAG.

Unfortunately, for general analytics jobs with arbitrary DAGs, the answer is in the negative. In fact, we find that for some jobs, it is *impossible* to find a schedule that achieves both cost and JCT optimality. The key insight behind this observation is that the launch time of a task affects not only itself, but also its downstream tasks — the optimal launch time for one task (Theorem 4.2) may negatively affect tasks in its downstream. We illustrate this with the example in Figure 6, that shows a job with three stages, each with only one step and one task. Stage 2 has a produce rate of 1 and consume rate of 3. Stage 3 has a produce rate of 3 and consume rate of 1. The arrows denote parent-child step pairs. Figure 6 (a) shows NIMBLE approach, which greedily optimizes the duration and finish time from Stage 1-3 based on the produce rate of each stage's parent steps; the end-to-end execution time of the entire job is 5, while its cost is 7. Figure 6 (b) shows an alternative strategy, that launches tasks across all three stages (Stage 1-3) at $t = 0$. This increases the duration of Stage 2 from 1 to 3, and consequently, the cost of execution of the job from 7 to 9. However, doing so also reduces the produce rate for Stage 3 to 1, allowing Stage 3 be completely pipelined with Stage 2. As such, the finish time of the entire job reduces from 5 to 3. Note that no schedule can achieve both a JCT of 3 and a cost of 7, since optimal JCT can only be achieved if Stage 2 and 3 are started at $t = 0$, which ensures a sub-optimal cost.

Cost optimality & cost-JCT *Pareto-optimality*. Despite the negative result above, NIMBLE scheduling efficiently navigates the cost-JCT tradeoff for jobs with arbitrary DAGs:

Theorem 4.3 *For a job with arbitrary DAG,* NIMBLE *scheduling in Algorithm 1 is (1) optimal in cost; and (2) Pareto-optimal between cost and JCT.*

Proof We first consider cost-optimality: since Algorithm 1 ensures optimal execution duration for each task in a job (Theorem 4.2), the aggregated duration across all tasks in the job, and therefore, the job execution cost, is also optimal.

Since the cost is always optimal, for Pareto-optimality we only need to show that no solution can further reduce job finish time without also increasing its cost. Our proof builds on the intuition developed for the example in Figure 6. First, we note that delaying the start time beyond T_s^* for any task cannot reduce its completion time; the only possibility to reduce JCT is to pick a start time earlier than T_s^*. As per Theorem 4.2, starting a task any sooner than T_s^* must increase its duration. Moreover, doing so will not reduce the duration of any other task, since they are already optimal. Thus, even if starting the task before T_s^* did improve JCT, it would only do so by increasing the aggregate duration across all tasks in the job, and therefore, its cost. ■

As an interesting aside, we note that for the example in Figure 6, we face this hard tradeoff between JCT and cost optimality because Stage 3 has a larger duration compared to Stage 2. Instead, if Stage 3 had a duration of 0.5, starting Stage 2 any sooner than $t = 2$ (at higher cost) would not have made stage 3 finish any faster. In practice, downstream stages often have a shorter duration compared to the upstream stages, since frequently used operators such as reduce, filter and join often significantly reduce the output data volume to downstream stages. In such cases, NIMBLE can achieve both optimal cost and JCT simultaneously — evaluation results on a wide range of analytics jobs in §6 validate this argument.

5 Design Details

In this section, we describe how we incorporate NIMBLE scheduling into Caerus, a new fine-grained task-level scheduler for serverless analytics frameworks. We first describe Caerus design components and application workflow (§5.1), and then describe its implementation details (§5.2).

5.1 Caerus System

We now describe Caerus system components and how they fit together (Figure 7). Before describing these components, we first briefly summarize the design employed by existing serverless analytics frameworks.

Primer on serverless analytics frameworks. Recent proposals on serverless analytics frameworks [8, 12, 14] share similar designs. Figure 7 depicts this design (adapted from [12]). The framework takes as input a job execution plan (DAG) that captures dependencies between stages and the number of tasks within each stage. It uses this to generate code for the individual tasks, compiles it and packages it with necessary dependen-

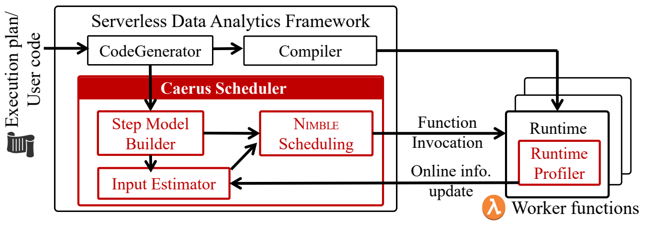


Figure 7: **Caerus system components & workflow (§5.1).**

cies. To execute a job, a scheduler launches tasks as serverless functions and monitors their progress. Pywren [8, 14] is similar, but omits the code-generation and compilation steps and directly takes task code and execution plan as input.

Caerus integrates with these analytics frameworks by simply replacing their task-level scheduler, and taking over the task launching and monitoring responsibilities. We next describe the major components of Caerus scheduler (Figure 7) in detail.

The step model builder is responsible for extracting the fine-grained step dependency model that NIMBLE scheduling expects, either from the job's execution plan or the user code. If the input is user code (e.g., a Python function in PyWren [8, 14]), Caerus provides a step annotation API that users can employ to specify the step information Caerus expects:

```
s = createStep() # Create a step object
s.start() # Notify system about step start
s.end() # Notify system about step end
s.addParent(stageID, stepID) # Specify parent step
```

If the query code is generated by a CodeGenerator based on the execution plan (as in Starling [12]), the step dependencies can be extracted during code generation. Most popular query execution engines (*e.g.*, SparkSQL) generate code based on the Volcano [44] iterator or `WholeStageCodegen` [45] model, which fuses operators as much as possible to maximize pipelining. As such, the generated code in such models is already composed of blocks separated by pipeline breakers, where each block corresponds exactly to a step in our step model. Caerus augments the CodeGenerator to additionally generate step-annotations at the start and end of blocks, along with step dependencies, using the step annotation API outlined above.

Input estimator & runtime profiler. Recall from §3 that NIMBLE scheduling relies on estimates of step produce rate (r_p) and consume rate (r_c) for steps with parents, and duration (d^*_{si}) for steps without parents, to make scheduling decisions. To facilitate accurate estimates, we leverage the observation that task and job-level statistics can be accurately estimated by tracking profiled information from prior job runs, since such analytics jobs in production workloads tend to be recurring in nature [26–28, 46, 47]. In particular, the input estimator in Caerus is responsible for collecting information for prior executions for each job (i.e., the job history) and maintaining estimates for various r_p, r_c and d^*_{si} values.

For higher accuracy, the input estimator continuously refines its r_p, r_c and d^*_{si} estimates based on realtime task progress. To facilitate this, a runtime profiler (similar to [27, 40]) periodically profiles and reports such metadata to the input estimator. Our runtime profiler is incorporated into the function runtime in serverless frameworks [8] that is shared across all tasks.

Caerus workflow. For each job, the step-model builder first extracts the step model from code generator or directly from the annotated function code. The input estimator maintains estimates of algorithm inputs (r_p, r_c and d^*_{si} values) for each step based on prior job runs. The NIMBLE scheduling module then calculates launch time based on both the algorithm inputs and step model, and launches each task at the calculated time. Launched tasks periodically report their progress to the input estimator via the runtime profiler, which is leveraged to refine the input estimates for future runs.

Caerus scalability. Caerus's scheduling performance scales well with the number of available CPU cores due to two main reasons. First, tasks within a stage have independent launch times, which permits parallel calculation and launching. Second, while the number of online update messages from runtime profiler grows linearly with the number of tasks, it can be served in parallel by partitioning input estimates for different tasks across different CPU cores.

Fault-tolerance. Caerus handles task failures by restarting them. For controller fault-tolerance, Caerus relies on traditional primary-backup mechanisms [30, 48]. The backup maintains consistent copies of the job's step model, launched and queued tasks, and runtime profiled information from prior job runs. During recovery, Caerus fetches this metadata from the backup and resumes scheduling queued tasks using NIMBLE.

5.2 Caerus Implementation

Our Caerus prototype is implemented atop Pywren [8], a serverless data analytics engine that runs on AWS Lambda [1]. We use Amazon S3 [21] for persistent data storage and Jiffy [24] for intermediate data storage.

SQL analytics with Caerus. We implement a SQL query execution framework atop Locus to highlight the benefit for Caerus scheduling for SQL analytics workloads. We employ Apache Spark's query planner to generate the query plan from the original SQL query, and then use Pandas to implement the SQL operators. Pandas' current implementation for SQL operators (like `JOIN` and `GROUPBY`) employs the lazy approach, *e.g.*, for the join example in Figure 3 (bottom), Pandas would only start the `JOIN` operation after all the data in both input tables are ready. We therefore modify the operator implementations in Pandas to conform to the step dependency model required by NIMBLE scheduling.

As a concrete example consider the implementation of the SQL job in Figure 3 (bottom) in Caerus. For the first two map stages, each task keeps reading input data from S3. After reading each small chunk of input data, it performs the map function, and partitions the output data into *chunks* based on key hashes. To implement shuffle, we maintain a FIFO queue for each join task in the intermediate storage. Once an output chunk is ready, the map task pushes it to the corresponding join task's queue. Note that a join task receives data from two shuffles (i.e., from map1 and map2). As such, each join task has two receiver queues: queue A (for data from `m1.s1`) and queue B (or data from `m2.s1`). After being launched, each join task fetches data from queue A and builds the hash table incrementally in step `j.s1`. Once the hash-table is built, the step `j.s2` fetches data from queue B, performs a join of the fetched data with the data in the hash-table, and writes the output to persistent storage.

Identifying pipeline-breakers. In Caerus, we implement all commonly used SQL operators (*e.g.*, `FILTER`, `JOIN`, `SORT`, `GROUP BY`, aggregates, etc.), employ the widely-used Volcano iterator model [44] to identify pipeline-breakers, and specify them using the step annotation API. As such, Caerus can run all TPC-DS and Big-data benchmark queries — we evaluate a representative subset in §6.

Accurate parameter estimations. NIMBLE scheduling relies on accurate estimation of various parameters (r_p, r_c, d^*_{si}), which can be complicated due unpredictable variations stemming from a range of sources. Fortunately, we found a majority of these sources had little to no variation across AWS Lambda executions, including (1) processing time for various operators; (2) function launch times[5]; and (3) function ingress/egress bandwidth to intermediate storage.

However, we did observe unpredictable performance variations for Amazon S3 reads and writes, particularly with larger number of parallel tasks (≥ 100). To minimize parameter estimation errors caused by these variations, we adopt a straggler mitigation technique for S3 reads and writes similar to [12] — Caerus tasks proactively establishes a new connection to S3 when a transfer takes longer than expected, and uses the response from whichever connection performs the read or write first. Moreover, we found larger S3 reads/writes to have unpredictable durations, so we break them into multiple smaller chunks. We show in §6 how these modifications ensure negligible estimations errors for a wide range of evaluated workloads.

6 Evaluation

We now evaluate Caerus implementation (§5.2) using three analytics workloads: TeraSort benchmark (§6.1), TPC-DS Benchmark (§6.2) and BigData Benchmark (§6.3). All of our experiments use Lambda instances with 3GB memory and deploy Jiffy on 6 m4.16xlarge EC2 instances.

Compared approaches. We compare Caerus with the eager and lazy scheduling approaches, implemented as a part of Caerus scheduler. These scheduling approaches correspond to the two extremes typically used in server-centric analytics frameworks for task level scheduling — lazy in Spark [30] and MapReduce [48], and eager in Dryad [32] and MapReduce Online [29]). Note that since our main contribution is a new scheduler for serverless analytics, our evaluation focuses on comparing scheduling approaches on a common analytics framework as opposed to comparing different analytics frameworks. As noted in §5, Caerus can integrate with any of existing serverless analytics frameworks [8, 11, 12, 14] and inherit their specific performance optimizations.

[5]We ensure function invocations are warm to avoid cold-start delays.

	Lazy	Eager	NIMBLE
JCT(s)	124	105	107
Cost(s)	10776	15756	11169

Table 1: **Comparison of NIMBLE against lazy and eager approaches for TeraSort on a 100GB dataset (§6.1).**

Performance metrics. We focus on two main metrics: JCT and cost of job execution. The former is measured as the time between job's first task's launch time to the last task's finish time. For the latter, we measure the aggregated duration across all tasks in the job as a proxy for cost. We avoid reporting precise dollar values, since these depend on the cloud provider and can change with market economics.

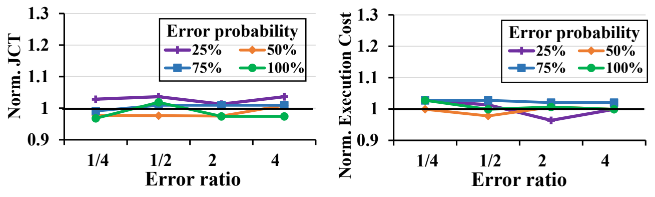
6.1 TeraSort

We port the TeraSort algorithm [49] implementation from Locus [14] to our framework for sorting large datasets. The algorithm operates in two stages: a partition stage that range partitions input data to intermediate storage, and a merge stage that reads these partitions, merges, sorts and writes them out as output. The sort job in our experiments uses 100 lambdas for both the map and reduce stage to sort 100GB of data generated using the Sort benchmark tool [50].

Table 1 compares the results of eager, lazy and NIMBLE scheduling approaches for the sort job. We observe little data skew for the TeraSort benchmark during both the partition and merge stages, and the ideal launch time for merge tasks identified by NIMBLE scheduling is roughly in the middle of the execution for partition stage. As such, NIMBLE achieves $1.16\times$ lower job completion time compared to the lazy scheme, $1.41\times$ lower cost than the eager approach. The results validate our analysis in §4, that NIMBLE scheduling can achieve near-optimal JCT and cost simultaneously for two stage map-reduce jobs in practice ($< 4\%$ in Table 1). NIMBLE's slight departure from optimal is due to delays in launch times introduced by the analytics framework (i.e., PyWren).

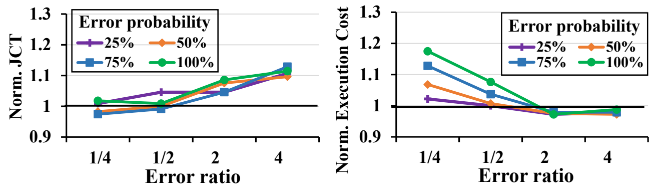
Impact of estimation errors. Caerus's JCT and cost-efficiency is gated on being able to estimate parameters like produce rate (r_p) and consume rate (r_p) accurately. Since Caerus's estimation errors are quite small in practice ($< 4\%$), we study their impact by introducing errors artificially.

To inject errors in produce rate estimation, we randomly select map tasks in our TeraSort job with probability p_e, and for each of them, incorrectly estimate the data output rate by Caerus's offline estimation as $k\times$ the actual value. We denote p_e as error probability and k as the error ratio. Since the produce rate r_p is the aggregated data output rate across all map tasks, our stochastic approach effectively injects errors



(a) JCT (b) Cost

Figure 8: **Impact of produce rate estimation errors (§6.1).** The results are normalized against the performance with no injected errors.

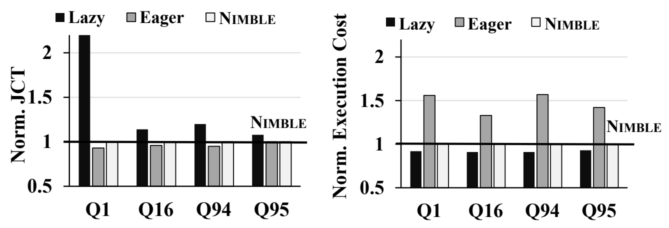


(a) JCT (b) Cost

Figure 9: **Impact of consume rate estimation errors (§6.1).** The results are normalized against the performance with no injected errors.

to the r_p estimate as well. Figure 8 shows the impact of the injected estimation errors on NIMBLE's performance, i.e., JCT and execution cost, with the corresponding metrics normalized against a run with no injected errors. We observe that NIMBLE's performance is minimally affected — across various combinations of error probability and error ratio, the JCT and execution cost is always within $\sim 4\%$ of the run with no injected errors. We attribute this to the runtime profiler, which tracks the real-time progress of each map task and refines the produce rate estimation by continuously re-estimating the task output rates. As such, the runtime profiler is able to correct the offline estimations in produce rate before launching the reducers, minimizing the impact of errors.

To study the impact of consume rate estimation errors, we employ a similar error rate and error ratio driven approach for reduce tasks. Note that runtime profiler is unable to correct for estimation errors in this case, since it can re-estimate the consume rate only *during* reduce task executions, which is *after* the reduce tasks have already been launched. Figure 8 shows the impact of injected errors on NIMBLE performance. For error ratio > 1 (i.e., estimated rate > actual rate), NIMBLE incorrectly estimates that the reduce task would finish faster than it actually does, while for error ratio < 1, it assumes the opposite. As expected, for the former case, Caerus launches reduce tasks later than it should, resulting in a longer JCT, while for the latter scenario, it launches them sooner than necessary, resulting in increased cost. Figure 9(a) shows that the normalized JCT increases from $1.07\times$ to $1.12\times$ as error ratio is increased from 2 to 4, while Figure 9(b) shows that the cost increases from $1.08\times$ to $1.18\times$ as error ratio increases from $1/2$ to $1/4$. Moreover, at higher error probability, the cost increase is greater since more reducers are launched earlier than necessary; while the JCT increase is largely unaffected since it only depends on the *slowest* task. Note that even with extreme estimation errors ($\frac{1}{4}\times$ and $4\times$), the increase in cost or JCT is only 12–18%.



(a) JCT (b) Cost

Figure 10: **NIMBLE performance for TPC-DS queries (§6.2).** Its JCT is comparable to eager and 1.08–2.2× lower than lazy, while its cost is comparable to lazy and 1.33–1.57× lower than eager.

6.2 TPC-DS Benchmark

The TPC-DS benchmark [51] has a set of standard decision support queries based on those used by retail product suppliers. The queries vary widely in terms of compute, storage and network I/O load variations. We evaluate Caerus on TPC-DS with scale factor of 1000, which results in a total input size of 1TB across various tables. Similar to Locus [14], we evaluate four representative queries (in terms of performance characteristics) from the TPC-DS Benchmark, specifically, queries `Q1`, `Q16`, `Q94` and `Q95`. All selected queries have complex DAGs comprising six to eight stages, with each query operating over a subset of the 1TB input — varying from 33GB to 312GB. Note that some late stages in the selected queries process much less data compare to early stages (after several `join` and `groupby` operations) — we adjust the degree of parallelism for these stages based on the amount of data they process.

Figure 10 compares the performance for NIMBLE with the lazy and eager approaches. The results indicate that Caerus can efficiently navigate the JCT-cost trade-off for all evaluated queries. Specifically, NIMBLE achieves JCT comparable to eager for all the queries, while outperforming lazy by 1.08–2.2×. For cost, NIMBLE matches the lazy approach while outperforming eager by 1.33–1.57×.

6.2.1 Diving deeper into NIMBLE benefits

In order to better understand the gains enabled by NIMBLE scheduling, we zoom in on the performance for Query `Q1` of the TPC-DS benchmark. Figure 11 shows the step-level dependencies for `Q1`, while Figure 12 shows the breakdown of execution time across different stages. Note that compute and network I/O take up most of the execution time, highlighting potential gains from pipelining. Figure 14 shows the job execution breakdowns with lazy, eager and NIMBLE scheduling.

Optimal pipelining across stages. We now walk through `Q1`'s execution with Caerus (Figure 14(c)). Caerus identifies 7 step dependencies (i.e., parent-child step pairs) as pipelineable, shown as red arrows in Figure 11.

While all map stages are launched at time $t = 0$, (since they do not have upstream dependencies), Caerus launches tasks across subsequent stages in a manner that ensures child steps in the above parent-child step pairs are optimally pipelined with the parent step, which corresponds to a large portion of the query execution. This is highlighted in Figure 14(c): when con-

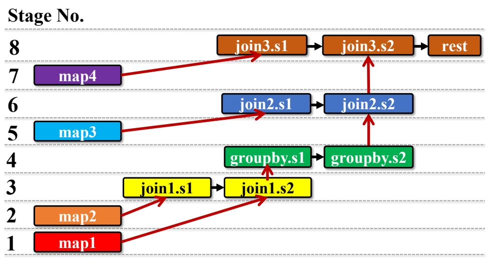


Figure 11: **The step dependency model for Q1.**

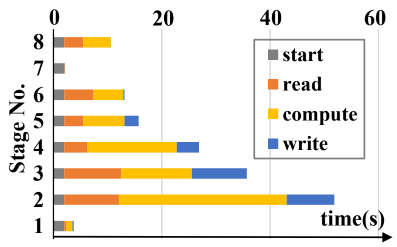


Figure 12: **Time breakdown for Q1.**

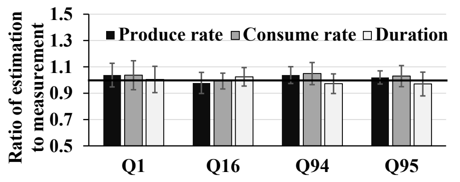
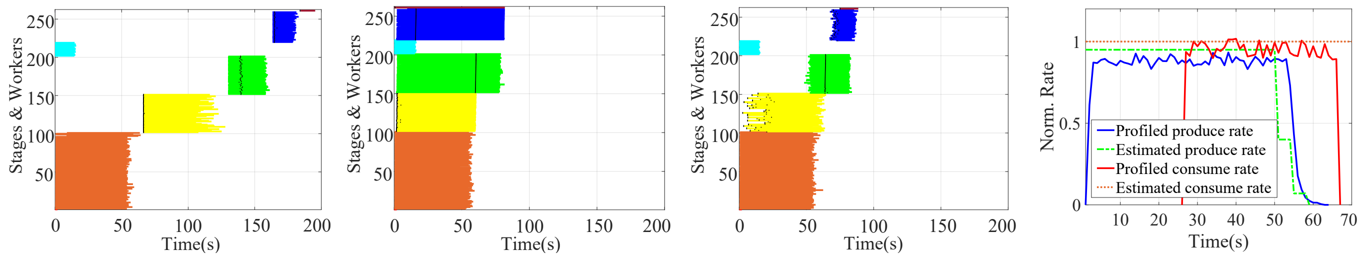


Figure 13: **Ratio of estimated & measured parameters for TPC-DS queries**. Error bars denote standard deviation across all tasks.



(a) Lazy (b) Eager (c) NIMBLE (d) NIMBLE Input Parameters

Figure 14: **Diving deeper into NIMBLE benefits for TPC-DS query Q1 (§6.2.1).** (a, b, c) show Q1 execution breakdown for lazy, eager, and NIMBLE, respectively; the black dots inside a task denote pipeline-breakers between steps. The degree of parallelism for Stages 1-8 is: {1, 100, 50, 50, 20, 40, 1, 1}. Note that Stages 1 (red) and 7 (purple) contain only one very short task, making them hard to see. (d) NIMBLE input parameters as measured by Caerus runtime profiler (solid lines) and as estimated by input estimator (dashed lines) for part of Stage 3 (yellow).

trasted with the lazy approach in Figure 14(a), Caerus enables a JCT that is 2.2× lower than the lazy approach. Meanwhile, Caerus also ensures that the tasks are not launched too soon in order to minimize time spent waiting for input from the parent step to become available, and therefore, the end-to-end job execution cost. As a concrete example, since step `groupby1.s1` is much shorter than step `join1.s2` and cannot finish before `join1.s2`, tasks in the `groupby` stage are started after tasks in `join1` stage are started, but before they finish execution. Compared with Figure 14(b), this allows name to Caerus achieve a cost that is 1.56× lower than the lazy approach.

Decreasing duration across stages. Another interesting takeaway from Q1's execution is that downstream stages in general process smaller amounts of data than upstream stages (since operators such as filter and join significantly reduce the data to downstream stages), and consequently have shorter durations. As noted in §4.2, NIMBLE scheduling enables both optimal cost and JCT simultaneously for such DAGs, which is reflected in Figure 14. Moreover, this observation holds across all of our evaluated TPC-DS queries, ensuring cost and JCT optimality with Caerus for all of them.

Accurate profiling & estimation for NIMBLE inputs. Figure 14(d) shows the normalized produce rate and consume rate of of step `join1.s2` in Stage 3, as profiled by Caerus runtime profiler and as estimated by Caerus input estimator. We make two observations: (1) the consume rate is stable as a function of time, as modeled in §3, and (2) the estimated values are a close approximation of the actual produce and consume rates. We find these observations extend to all stages across query Q1, as well as to all other queries we evaluate in this section — Figure 13 shows that the average error in parameter estimations for r_p, r_c and d^*_{si} is within 4% across all queries. As we already saw in §6.1, NIMBLE scheduling is also robust to higher estimation errors.

Data skew. We note that Stage 3 (yellow) experiences data skew across tasks (Figure 14(a)-14(c)) — our profiling indicates that some tasks process $>1.6\times$ more data than others. Caerus captures the effect of such data skew in its NIMBLE scheduling algorithm, and launches tasks in Stage 3 at a time that still ensures JCT and cost optimality for the job execution.

Fast scheduling decisions. The query Q1 has over 250 tasks across 8 stages — Caerus schedules and launches each task in about 400μs (on average). In contrast, when the task launch request is issued to AWS Lambda, it typically takes an additional $\sim 25-320$ms to start the task's execution [52]. As such, despite making much more fine-grained (i.e., task-level) decisions than traditional job schedulers, Caerus is fast enough to not be the bottleneck in the analytics execution pipeline.

6.3 BigData Benchmark

The Big Data Benchmark [53] is a query suite derived from production databases. We consider Query 3 (Q3), which is a join query with four stages, with a step dependency model similar to the first four stages of TPC-DS benchmark's Q1 (Figure 11). Our implementation uses shuffle hash join (SHJ), and efficiently pipelines the join stage with the map stages. Q3 reads in 123GB of input, and can perform joins with three different sizes: 485,312 rows in Q3A; 53,332,015 rows for Q3B; and 533,287,121 rows for Q3C. This allows us to understand the effect of join size on NIMBLE scheduling.

Figure 15 compares NIMBLE approach with both the lazy and eager approaches with different join data sizes (Q3A-Q3C).

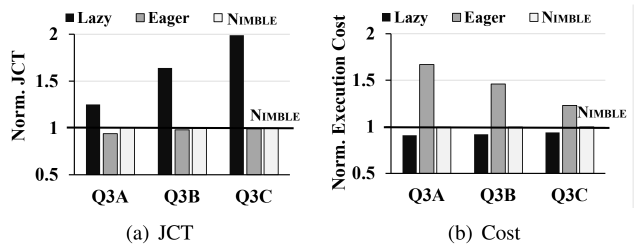


Figure 15: **NIMBLE performance for BigData Benchmark (§6.3).** NIMBLE's JCT improvement over lazy increases as join size increases (Q3A→Q3C), while its cost improvement over eager increases as join size decreases (Q3C→Q3A).

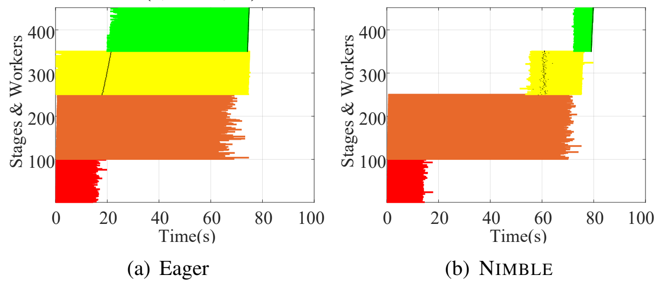


Figure 16: **Q3A execution breakdown** for (a) eager and (b) NIMBLE.

Interestingly, we observe that NIMBLE's relative JCT improvement compared to lazy increases from 1.29× to 1.99×, as join size increases from Q3A to Q3C. Meanwhile, NIMBLE's relative cost improvement compared to eager increases from 1.23× to 1.67×, as join size decreases from Q3C to Q3A.

To better understand the differences in cost and JCT improvements due join sizes, Figure 16 shows the execution breakdown for Q3A. As Q3A's join input data size is small, the `join` (yellow) and subsequent `groupby` (green) stages are much shorter than the initial `map` stage (orange). As such, pipelining these shorter stages with the `map` stage does not improve the JCT by much (1.29×) compared to the lazy approach. However, the eager solution significantly increases the cost by starting these short tasks very early (Figure 16(a)). Caerus, on the other hand, improves cost relative to eager by 1.67× by launching them at just the right time (Figure 16(b)).

Figure 17 compares the execution of lazy and Caerus for Q3C: as the input data size for join is now much larger, the duration of the join stage and groupby stage is comparable to the map stage (orange). As such, the eager approach does not lose as much in terms of cost by launching these tasks early. However, the lazy solution increases the JCT significantly by running these relatively longer stages one after the other (Figure 17(a)). In contrast, Caerus improves JCT by 1.99× relative to lazy by efficiently pipelining the join and groupby stage with the map stages (Figure 17(b)).

7 Related Work

We already discussed related work on server-centric and serverless analytics frameworks in §2.1 and §5.1. We now discuss prior work related to Caerus in other areas.

Some databases [45, 54–56] and data processing frameworks [31, 57] support pipelined execution via an *iterator*

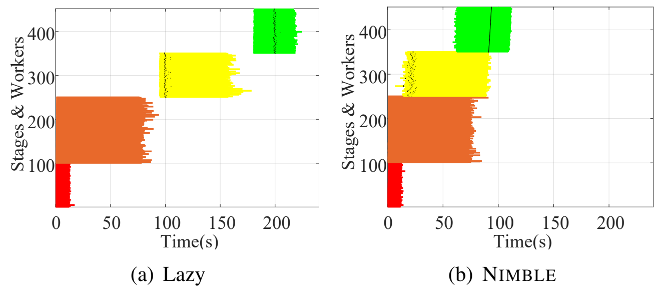


Figure 17: **Q3C execution breakdown** for (a) lazy and (b) NIMBLE.

model [44]. These approaches focus on maximally pipelining operators to minimize query completion time. Similar to these works, we leverages pipelined execution to achieve JCT-optimality for analytic jobs. In fact, our step dependency model draws inspiration from iterator models to identify regions of execution that can or cannot be overlapped with other regions. Unlike prior work, however, our approach also considers cost-optimality, a key concern in serverless analytics.

Caerus's scheduling problem is also related to the parallel query scheduling [58–60] in databases. Many of the proposed algorithms assign CPU and memory resources across operators considering both pipelinable and non-pipelinable dependencies across them. Unlike Caerus, however, these algorithms are designed for server-centric deployments and optimize for the query completion time under limited resource constraints.

Another related problem is Multi-Objective Query Optimization (MOQO), which searches for an query execution plan with an optimal trade-off between multiple conflicting cost metrics in databases [61–67]. While MOQO optimizes a query execution plan to determine its component set of operations and their orderings, Caerus takes the execution plan as input and specifically optimizes the task launch times for JCT and cost, i.e., Caerus approach is *complementary* to MOQO.

8 Conclusion

We have presented Caerus, a task scheduler for serverless analytics that uses a new NIMBLE scheduling algorithm. NIMBLE efficiently pipelines task executions across various stages in serverless analytics jobs, to ensure cost-optimality and Pareto-optimality between cost and JCT. We show that for a wide range of analytics workloads, NIMBLE is often optimal in *both* dimensions. This allows Caerus to outperform existing lazy scheduling approaches by 1.08–2.2× in JCT, and eager approaches by 1.21–1.57× in cost for these workloads.

Acknowledgments

We thank our anonymous reviewers and shepherd Srinivasan Seshan for their insightful comments. We also thank Qifan Pu, Zongheng Yang, Silvery Fu, Chenggang Wu, Danyang Zhuo, Joe Hellerstein and other members of RISELab for their constructive feedback. In addition to NSF CISE Expeditions Award CCF-1730628, this research is supported by gifts from Amazon Web Services, Ant Group, CapitalOne, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk and VMware.

References

[1] AWS Lamda. https://aws.amazon.com/lambda/.

[2] Google Cloud Functions. https://cloud.google.com/functions.

[3] Azure Functions. https://azure.microsoft.com/en-us/services/functions.

[4] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Menezes Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. Technical Report UCB/EECS-2019-3, EECS Department, University of California, Berkeley, 2019.

[5] Joseph M Hellerstein, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651*, 2018.

[6] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. The rise of serverless computing. *Communications of the ACM*, 2019.

[7] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *NSDI*, 2017.

[8] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: distributed computing for the 99%. In *SoCC*, 2017.

[9] Vaishaal Shankar, Karl Krauth, Kailas Vodrahalli, Qifan Pu, Benjamin Recht, Ion Stoica, Jonathan Ragan-Kelley, Eric Jonas, and Shivaram Venkataraman. Serverless linear algebra. In *SoCC*, 2020.

[10] Youngbin Kim and Jimmy Lin. Serverless data analytics with Flint. In *CLOUD*, 2018.

[11] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *ATC*, 2019.

[12] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. Starling: A scalable query engine on cloud function services. In *SIGMOD*, 2020.

[13] Qubole Announces Apache Spark on AWS Lambda. https://www.qubole.com/blog/spark-on-aws-lambda.

[14] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: scalable analytics on serverless infrastructure. In *NSDI*, 2019.

[15] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *OSDI*, 2018.

[16] Databricks Serverless: Next Generation Resource Management for Apache Spark. https://bit.ly/3cbLe3L.

[17] Amazon. Amazon Aurora Serverless. https://aws.amazon.com/rds/aurora/serverless.

[18] Azure. Azure SQL Data Warehouse. https://azure.microsoft.com/en-us/services/sql-data-warehouse.

[19] Charles Reiss. *Understanding Memory Configurations for In-Memory Analytics*. PhD thesis, EECS Department, University of California, Berkeley, 2016.

[20] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient Memory Disaggregation with Infiniswap. In *NSDI*, 2017.

[21] Amazon S3. https://aws.amazon.com/s3.

[22] Introduction to object storage in Azure. https://docs.microsoft.com/en-us/azure/storage/blobs/storage-blobs-introduction.

[23] Google Cloud Storage. https://cloud.google.com/storage/.

[24] Jiffy: A virtual memory abstraction for serverless architectures. https://github.com/resource-disaggregation/jiffy.

[25] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, 2011.

[26] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. *SIGCOMM CCR*, 2014.

[27] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *OSDI*, 2016.

[28] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *OSDI*, 2016.

[29] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *NSDI*, 2010.

[30] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *HotCloud*, 2010.

[31] Spark SQL. https://spark.apache.org/sql/.

[32] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *SIGOPS*, 2007.

[33] Apache Hive. https://hive.apache.org.

[34] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 2008.

[35] Cloudera Impala. http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/impala.html.

[36] Scheller Brandon. Best practices for resizing and automatic scaling in Amazon EMR. https://amzn.to/2ZJYY0D, 2018.

[37] Hadoop Distributed File System. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[38] AWS Lambda pricing. https://aws.amazon.com/lambda/pricing/.

[39] Amazon S3 pricing. https://aws.amazon.com/s3/pricing/.

[40] Ganesh Ananthanarayanan, Srikanth Kandula, Albert G Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in map-reduce clusters using mantri. In *OSDI*, 2010.

[41] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. A study of skew in mapreduce applications. *Open Cirrus Summit*, 2011.

[42] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Skewtune: mitigating skew in mapreduce applications. In *SIGMOD*, 2012.

[43] Cliff Engle, Antonio Lupher, Reynold Xin, Matei Zaharia, Michael J Franklin, Scott Shenker, and Ion Stoica. Shark: Fast Data Analysis Using Coarse-grained Distributed Memory. In *SIGMOD*, 2012.

[44] Goetz Graefe. Volcano, an extensible and parallel query evaluation system; cu-cs-481-90. 1990.

[45] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. 2011.

[46] Mosharaf Chowdhury, Zhenhua Liu, Ali Ghodsi, and Ion Stoica. Hug: Multi-resource fairness for correlated and elastic demands. In *NSDI*, 2016.

[47] Sameer Agarwal, Srikanth Kandula, Nico Bruno, Ming-Chuan Wu, Ion Stoica, and Jingren Zhou. Reoptimizing data parallel computing. In *NSDI*, 2012.

[48] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 2008.

[49] Owen O'Malley. Terabyte sort on apache hadoop. 2008.

[50] gensort Data Generator. http://www.ordinal.com/gensort.html.

[51] TPC-DS. http://www.tpc.org/tpcds/.

[52] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *ATC*, 2018.

[53] AMPLab. The BigData Benchmark. https://amplab.cs.berkeley.edu/benchmark/, 2018.

[54] Luc Bouganim, Daniela Florescu, and Patrick Valduriez. Dynamic load balancing in hierarchical parallel database systems. 1996.

[55] Li Wang, Minqi Zhou, Zhenjie Zhang, Yin Yang, Aoying Zhou, and Dina Bitton. Elastic pipelining in an in-memory database cluster. In *SIGMOD*, 2016.

[56] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age. In *SIGMOD*, 2014.

[57] Yuan Yu Michael Isard Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, and Pradeep Kumar Gunda Jon Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. *Proc. LSDS-IR*, 2009.

[58] Minos N Garofalakis and Yannis E Ioannidis. Multi-dimensional resource scheduling for parallel queries. *ACM SIGMOD Record*, 1996.

[59] Chandra Chekuri, Waqar Hasan, and Rajeev Motwani. Scheduling problems in parallel query optimization. In *SIGACT-SIGMOD-SIGART*, 1995.

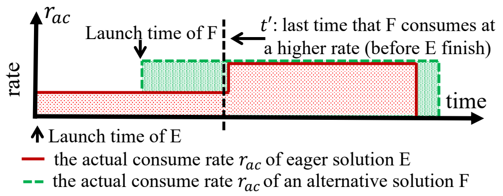


Figure 18: [Example for Lemma A.1] The actual consume rate of (i) the eager approach $\mathcal{E}$; and (ii) an alternate approach $\mathcal{F}$ with a later launch time.

[60] Minos N Garofalakis and Yannis E Ioannidis. Parallel query scheduling and optimization with time-and space-shared resources. *SORT*, 1997.

[61] Immanuel Trummer and Christoph Koch. Approximation schemes for many-objective query optimization. In *SIGMOD*, 2014.

[62] Immanuel Trummer and Christoph Koch. An incremental anytime algorithm for multi-objective query optimization. In *SIGMOD*, 2015.

[63] Sumit Ganguly, Waqar Hasan, and Ravi Krishnamurthy. Query optimization for parallel execution. In *SIGMOD*, 1992.

[64] Sameer Agarwal, Anand P Iyer, Aurojit Panda, Samuel Madden, Barzan Mozafari, and Ion Stoica. Blink and it's done: interactive queries on very large data. 2012.

[65] Immanuel Trummer and Christoph Koch. Multi-objective parametric query optimization. *ACM SIGMOD Record*, 2016.

[66] Christos H Papadimitriou and Mihalis Yannakakis. Multiobjective query optimization. In *SIGMOD-SIGACT-SIGART*, 2001.

[67] Amol Deshpande and Lisa Hellerstein. Parallel pipelined filter ordering with precedence constraints. *TALG*, 2012.

A Theoretical Proofs

Lemma A.1 *The eager approach always optimizes the finish time for a reduce task.*

Proof Figure 18 shows an example of the actual consume rate $r_{ac}(t)$ for `r.s1` under two approaches: the eager solution $\mathcal{E}$ and an alternate approach $\mathcal{F}$ with a later launch time. We see that before $\mathcal{E}$ finishes, there may exist some time t such that $r_{ac}(t)$ of $\mathcal{F}$ is greater than $r_{ac}(t)$ of $\mathcal{E}$. Based on Equation 1, we know that this is because $\mathcal{E}$ has processed all its inputs by t ($S(t) = 0$ for $\mathcal{E}$), but $\mathcal{F}$ has not ($S(t) = 1$ for F) as it is launched later. Denote the last such time point before $\mathcal{E}$ finishes as t', we have: (1) by time t', $\mathcal{E}$ has processed all data generated before t'; (2) after time t', $r_{ac}(t)$ of $\mathcal{E}$ is no less than $\mathcal{F}$ until it finishes. The combination of these two observations shows that $\mathcal{E}$ always has an earlier finish time for `r.s1` than $\mathcal{F}$. Since $T_e^* = t_{e,\mathtt{s1}}^* + d_{\mathtt{s2}}^*$, $\mathcal{E}$ also ensures optimal task finish time. ■

Proof of Theorem 4.2 :

Theorem 4.2 *For a task in an analytics job with an arbitrary execution DAG, given the execution (produce rate) of all its parent steps, we can always achieve both optimal execution duration and finish time by launching it at $T_s^* = T_e^* - D^*$, where T_e^* is the optimal finish time and D^* is the optimal duration computed using Steps 1 and 2 described in §4.2.1.*

Proof We prove Theorem 4.2 by mathematical induction on the number of steps of the task.

Base case: We first show that Theorem 4.2 holds for a task with only one step. In this case, the problem reduces to the single parent-child step case for two-stage map-reduce jobs as in Theorem 4.1, which we have already shown to hold.

Inductive step: We now show that for any $n > 1$, if Theorem 4.2 holds for tasks with n steps, it also holds for tasks with $n+1$ steps. Consider a task with $n+1$ steps. For the first n steps, we denote the optimal finish time as $T_e^*(n)$, and cost as $D^*(n)$. Since Theorem 4.2 holds for any task with n steps, $T_e^*(n)$ and $D^*(n)$ can be achieved simultaneously by launching the $(n+1)$-step task at $T_s^*(n) = T_e^*(n) - D^*(n)$. Based on Equation 2 and the definition of the optimal task duration (Step 1 in §4.2.1), we have:

$$\begin{aligned} T_e^*(n+1) &= max(t'_{e,\mathtt{s}n+1}, T_s^*(n) + d_{\mathtt{s}n+1}^*) \\ D^*(n+1) &= D^*(n) + d_{\mathtt{s}n+1}^* \end{aligned} \tag{3}$$

Based on Equation 3, the launch time calculated from Theorem 4.2 for the $(n+1)$-step task is

$$\begin{aligned} T_s^*(n+1) &= T_e^*(n+1) - D^*(n+1) \\ &\geq (T_s^*(n) + d_{\mathtt{s}n+1}^*) - (D^*(n) + d_{\mathtt{s}n+1}^*) \\ &= T_e^*(n) - D^*(n) \\ &= T_s^*(n) \end{aligned} \tag{4}$$

Note that if a step j starts at time t_1 and executes at full load (i.e., it will never stall for data to become available), then it must also be able to execute at full load if it starts at any time $t_2 \geq t_1$. Since we have $T_s^*(n+1) \geq T_s^*(n)$ from Equation 4, if we launch the task at $T_s^*(n+1)$, we can always execute the first n steps at full load (i.e., with optimal duration $D^*(n)$). As such, with launch time $T_s^*(n+1)$, the corresponding finish time of the first n steps is $T_s^*(n+1) + D^*(n)$, which is also the start time of the last step $\mathtt{s}n+1$. Based on Equation 3 we have:

$$\begin{aligned} \text{Start time of step } \mathtt{s}n+1 &= T_s^*(n+1) + D^*(n) \\ &= (T_e^*(n+1) - D^*(n+1)) + D^*(n) \\ &= T_e^*(n+1) - d_{\mathtt{s}n+1}^* \\ &= max(t'_{e,\mathtt{s}n+1}, T_s^*(n) + d_{\mathtt{s}n+1}^*) - d_{\mathtt{s}n+1}^* \\ &\geq t'_{e,\mathtt{s}n+1} - d_{\mathtt{s}n+1}^* \end{aligned} \tag{5}$$

Equation 5 indicates that if we launch the task at $T_s^*(n+1)$, the start time of the step $\mathtt{s}n+1$ is no less than $t'_{e,\mathtt{s}n+1} - d_{\mathtt{s}n+1}^*$. Note that $t'_{e,\mathsf{s}n+1}$ is the optimal finish time of step $\mathsf{s}n+1$ calculated only based on its parent. Just as in the proof of Theorem 4.1, which covers the single parent-child step pair case, we can then show that the step $\mathsf{s}n+1$ must execute at full rate if it is launched at $t'_{e,\mathsf{s}n+1} - d^*_{\mathsf{s}n+1}$.

Taken together, if we launch the task at $T^*_s(n+1)$, all $n+1$ steps can execute at full load, which indicates it achieves the optimal duration $D^*(n+1)$. Moreover, recall that $T^*_s(n+1) = T^*_e(n+1) - D^*(n+1)$. This means that if the task starts at $T^*_s(n+1)$ and has a duration of $D^*(n+1)$, it must finish at $T^*_e(n+1)$. As such, we can achieve both $T^*_e(n+1)$ and $D^*(n+1)$ by launching the task at $T^*_s(n+1)$.

Conclusion: Since both the base case and the inductive step hold, Theorem 4.2 holds by mathematical induction. ■

Ownership: A Distributed Futures System for Fine-Grained Tasks

Stephanie Wang*, Eric Liang*, Edward Oakes*, Ben Hindman, Frank Luan, Audrey Cheng, Ion Stoica
*UC Berkeley, *and Anyscale*

Abstract

The distributed futures interface is an increasingly popular choice for building distributed applications that manipulate large amounts of data. Distributed futures are an extension of RPC that combines futures and distributed memory: a distributed future is a reference whose eventual value may be stored on a remote node. An application can then express distributed computation without having to specify when or where execution should occur and data should be moved.

Recent distributed futures applications require the ability to execute *fine-grained computations*, i.e., tasks that run on the order of milliseconds. Compared to coarse-grained tasks, fine-grained tasks are difficult to execute with acceptable system overheads. In this paper, we present a distributed futures system for fine-grained tasks that provides fault tolerance without sacrificing performance. Our solution is based on a novel concept called *ownership*, which assigns each object a leader for system operations. We show that this decentralized architecture can achieve horizontal scaling, 1ms latency per task, and fast failure handling.

1 Introduction

RPC is a standard for building distributed applications because of its generality and because its simple semantics yield high-performance implementations. The original proposal uses *synchronous* calls that *copy* return values back to the caller (Figure 2a). Several recent systems [4, 34, 37, 45] have extended RPC so that, in addition to distributed communication, the system may also manage *data movement* and *parallelism* on behalf of the application.

Data movement. Pass-by-value semantics require all RPC arguments to be sent to the executor by copying them directly into the request body. Thus, performance degrades with *large* data. Data copying is both expensive and unnecessary in cases like Figure 2a, where a process executes an RPC over data that it previously returned to the same caller.

To reduce data copies, some RPC systems use *distributed memory* [16, 27, 37, 40, 41]. This allows large arguments to be passed by *reference* (Figure 2b), while small arguments can still be passed by value. In the best case, arguments passed by reference to an RPC do not need to be copied if they are already on the same node as the executor (Figure 2b). Note that, like traditional RPC, we make all values *immutable* to simplify the consistency model and implementation.

Parallelism. RPCs are traditionally *blocking*, so control is only returned to the caller once the reply is received (Figure 2a). *Futures* are a popular method for extending RPC with asynchrony [8, 29], allowing the system to execute functions in parallel with each other and the caller. With *composition* [29, 37], i.e., passing a future as an argument to another RPC, the application can also express the parallelism and dependencies of future RPCs. For example, in Figure 2c, `add` is invoked at the beginning of the program but only executed by the system once `a` and `b` are computed.

```
a_future = compute()
b_future = compute()
c_future = add(a_future, b_future)
c = system.get(c_future)
```

Figure 1: A distributed futures program. `compute` and `add` are stateless. `a_future`, `b_future`, and `c_future` are distributed futures.

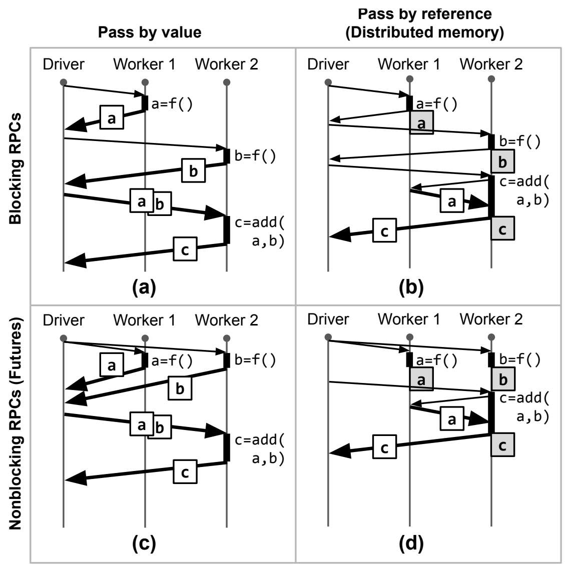


Figure 2: Example executions of the program from Figure 1. **(a)** With RPC. **(b)** With RPC and distributed memory, allowing the system to reduce data copies. **(c)** With RPC and futures, allowing the system to manage parallel execution. **(d)** With distributed futures.

Distributed futures are an extension of RPC that combines futures with distributed memory: a *distributed future* is a reference whose eventual value may be stored on a remote node (Figure 2d). An application can then express distributed computation without having to specify when or where execution should occur and data should be moved. This is an increasingly popular interface for developing distributed applications that manipulate large amounts of data [4, 34, 37, 45].

As with traditional RPC, a key goal is generality. To achieve this, the system must minimize the overhead of each function call [13]. For example, the widely used gRPC provides horizontal scalability and sub-millisecond RPC latency, making it practical to execute millions of *fine-grained* functions, i.e. millisecond-level "tasks", per second [2].

Similarly, there are emerging examples of large-scale, fine-grained applications of distributed futures, including reinforcement learning [34], video processing [22,43], and model serving [49]. These applications must optimize parallelism and data movement for performance [39,43,49], making distributed futures apt. Unfortunately, existing systems for distributed futures are limited to *coarse-grained* tasks [37].

In this paper, we present a distributed futures system for fine-grained tasks. While others [34,37,45] have implemented distributed futures before, our contribution is in identifying and addressing the challenges of providing *fault tolerance for fine-grained tasks without sacrificing performance*.

The primary challenge is that distributed futures introduce *shared state* between processes. In particular, an object and its metadata are shared by its reference holder(s), the RPC executor that creates the object, and its physical location(s). To ensure that each reference holder can dereference the value, the processes must coordinate, a difficult problem in the presence of failures. In contrast, traditional RPC has no shared state, since data is passed by value, and naturally avoids coordination, which is critical to scalability and low latency.

For example, in Figure 2a, once worker 1 copies `a` to the driver, it does not need to be involved in the execution of the downstream `add` task. In contrast, worker 1 stores `a` in Figure 2d, so the two workers must coordinate to ensure that `a` is available long enough for worker 2 to read. Also, worker 1 must garbage-collect `a` once worker 2 executes `add` and there are no other references. Finally, the processes must coordinate to detect and recover from the failure of another process.

The common solution in previous systems is to use a centralized master to store system state and coordinate these operations [34,37]. A simple way to ensure fault tolerance is to record and replicate metadata at the master *synchronously* with the associated operation. For example, in Figure 2d, the master would record that `add` is scheduled to worker 2 *before* dispatching the task. Then, it can correctly detect *c*'s failure if worker 2 fails. However, this adds significant overhead for applications with a high volume of fine-grained tasks [].

Thus, decentralizing the system state is necessary for scalability. The question is how to do so without complicating coordination. The key insight in our work is to exploit the application structure: a distributed future may be shared by passing by reference, but *most distributed futures are shared within the scope of the caller*. For example, in Figure 1, `a_future` is created then passed to `add` in the same scope.

We thus propose *ownership*, a method of decentralizing system state across the RPC *executors*. In particular, the caller of a task is the *owner* of the returned future and all related metadata. In Figure 2d, the driver owns `a`, `b`, and `c`.

This solution has three advantages. First, for horizontal scalability, the application can use nested tasks to "shard" system state across the workers. Second, since a future's owner is the task's caller, task latency is low because the required metadata writes, though synchronous, are local. This is in contrast to an application-agnostic method of sharding, such as consistent hashing. Third, each worker becomes in effect a centralized master for the distributed futures that it owns, simplifying failure handling.

The system guarantees that if the owner of a future is alive, any task that holds a reference to that future can eventually dereference the value. This is because the owner will coordinate system operations such as reference counting, for memory safety, and lineage reconstruction, for recovery. Of course, this is not sufficient if the owner fails.

Here, we rely on lineage reconstruction and a second key insight into the application structure: in many cases, the references to a distributed future are held by tasks that are a descendant of the failed owner. The failed task can be recreated through lineage reconstruction by *its* owner, and the descendant tasks will also be recreated in the process. Therefore, it is safe to *fate-share* any tasks that have a reference to a distributed future with the future's owner. As we expect failures to be relatively rare, we argue that this reduction in system overheads and complexity outweighs the cost of additional re-execution upon a failure.

In summary, our contributions are:

- A decentralized system for distributed futures with transparent recovery and automatic memory management.
- A lightweight technique for transparent recovery based on lineage reconstruction and fate sharing.
- An implementation in the Ray system [34] that provides high throughput, low latency, and fast recovery.

2 Distributed Futures

2.1 API

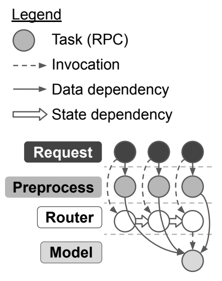
The key benefit of distributed futures is that the system can transparently manage parallelism and data movement on behalf of the application. Here, we describe the API (Table 1).

To spawn a *task*, the caller invokes a *remote function* that immediately returns a `DFut` (Table 1). The spawned task comprises the function and its arguments, resource requirements, etc. The returned `DFut` refers to the *object* whose value will be returned by the function. The caller can *dereference* the `DFut` through `get`, a blocking call that returns a copy of the object. The caller can *delete* the `DFut`, removing it from scope and allowing the system to reclaim the value. Like other systems [34,37,45], all objects are *immutable*.

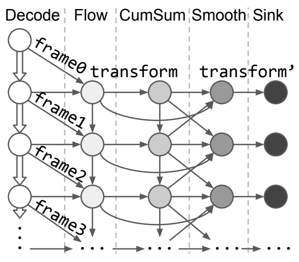
After the creation of a `DFut` through task invocation, the caller can create other references in two ways. First, the caller can pass the `DFut` as an argument to another task. `DFut` task arguments are implicitly dereferenced by the system. Thus, the task will only begin once all upstream tasks have finished, and the executor sees only the `DFut` *values*.

Operation	Semantics
`f(DFut x)` → `DFut`	Invoke the remote procedure `f`, and pass `x` by reference. The system implicitly dereferences `x` to its `Value` before execution. Creates and returns a distributed future, whose value is returned by `f`.
`get(DFut x)` → `Value`	Dereference a distributed future. Blocks until the value is computed and local.
`del(DFut x)`	Delete a reference to a distributed future from the caller's scope. Must be called by the program.
`Actor.f(DFut x)` → `DFut`	Invoke a stateful remote procedure. `f` must execute on the actor referred to by `Actor`.
`shared(DFut x)` → `SharedDFut`	Returns a `SharedDFut` that can be used to pass `x` to another worker, without dereferencing the value.
`f(SharedDFut x)` → `DFut`	Passes `x` as a first-class `DFut`: The system dereferences `x` to the corresponding `DFut` instead of the `Value`.

Table 1: Distributed futures API. The full API also includes an actor creation call. A task may also return a `DFut` to its caller (nested `DFuts` are automatically flattened).



(a) Model serving



(b) Video processing

Figure 3: Distributed futures applications.

Second, the `DFut` can be passed or returned as a *first-class value* [21], i.e. passed to another task without dereferencing. Table 1 shows how to cast a `DFut` to a `SharedDFut`, so the system can differentiate when to dereference arguments. We call the process that receives the `DFut` a *borrower*, to differentiate it from the original caller. Like the original caller, a borrower may create other references by passing the `DFut` or casting again to a `SharedDFut` (creating further borrowers).

Like recent systems [4, 34, 45], we support *stateful* computation with actors. The caller creates an actor by invoking a *remote constructor function*. This immediately returns a reference to the actor (an `ARef`) and asynchronously executes the constructor on a remote process. The `ARef` can be used to spawn tasks bound to the same process. Similar to `DFuts`, `ARefs` are first-class, i.e. the caller may return or pass the `ARef` to another task, and the system automatically collects the actor process once all `ARefs` have gone out of scope.

2.2 Applications

Typical applications of distributed futures are those for whom performance requires the flexibility of RPC, as well as optimization of data movement and parallelism. We describe some examples here and evaluate them in Section 5.2.

Distributed futures have previously been explored for data-intensive applications that cannot be expressed or executed efficiently as data-parallel programs [34, 37]. Ciel identified the key ability to *dynamically* specify tasks during execution, e.g., based on previous results, rather than specify the entire graph upfront [37]. This enabled new workloads such as dynamic programming, which is recursive by nature [54].

Our goal is to expand the application scope to include those with *fine-grained* tasks that run in the milliseconds. We also explore the use of actors and first-class distributed futures.

Model serving. The goal is to reduce request latency while maximizing throughput, often by using model *replicas*. Depending on the model, a latency target might be 10-100ms [20]. Typically, an application-level scheduling policy is required, e.g., for staged rollout of new models [46].

Figure 3a shows an example of a GPU-based image classification pipeline. Each client passes its input image to a `Preprocess` task, e.g., for resizing, then shares the returned `DFut` with a `Router` actor. `Router` implements the scheduling policy and passes the `DFut` by reference to the chosen `Model` actor. `Router` then returns the results to the clients.

Actors improve performance in two ways: (1) each `Model` keeps weights warm in its local GPU memory, and (2) `Router` buffers the preprocessed `DFuts` until it has a batch of requests to pass to a `Model`, to leverage GPU parallelism for throughput. With *dynamic* tasks, the `Router` can also choose to flush its buffer on a timeout, to reduce latency from batching.

First-class distributed futures are important to reduce routing overhead. They allows the `Router` to pass the references of the preprocessed images to the `Model` actors, instead of copying these images. This avoids creating a bottleneck at the `Router`, which we evaluate in Figure 15a. While the application could use an intermediate storage system for preprocessed images, it would then have to manage additional concerns such as garbage collection and failures.

Online video processing. Video processing algorithms often have complex data dependencies that are not well supported by data-parallel systems such as Apache Spark [22, 43]. For example, video stabilization (Figure 3b) works by tracking objects between frames (Flow), taking a cumulative sum of these trajectories (CumSum), then applying a moving average (Smooth). Frame-to-frame dependencies are common, such as the video decoding state stored in an actor in Figure 3b. Each stage runs in 1-10s of milliseconds per frame.

Safe and timely garbage collection in this setting can be challenging because a single object (e.g., a video frame) may be referenced by multiple tasks. Live video processing is also latency-sensitive: output must be produced at the same frame rate as the input. Low latency relies on pipelined parallelism between frames, as the application cannot afford to wait for multiple input frames to appear before beginning execution.

With distributed futures, the application can specify the logical task graph *dynamically*, as input frames appear. Meanwhile, the system manages the physical execution, i.e. pipelined parallelism and garbage collection, according to the specified graph. Concurrent video streams can easily be

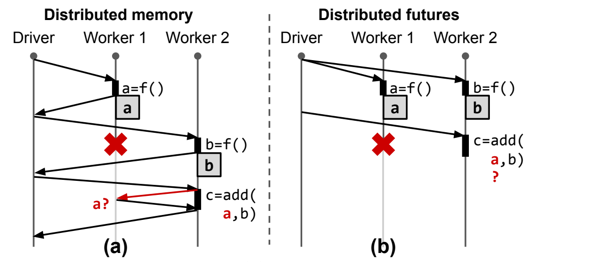


Figure 4: Failure detection. **(a)** `a`'s location is known by the time worker 2 receives the reference. **(b)** `a`'s location may not be known when worker 2 receives `add`, so worker 2 cannot detect the failure.

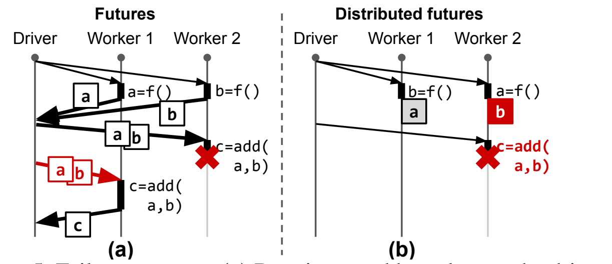


Figure 5: Failure recovery. **(a)** Data is passed by value, so the driver recovers by resubmitting `add`. **(b)** `b` is also lost. `f`'s description must be recorded during runtime so that `b` can be recomputed.

supported using nested tasks, one "driver" per stream. The system can then manage inter-video parallelism.

3 Overview

3.1 Requirements

The system guarantees that each `DFut` can be *dereferenced* to its value. This involves three problems: automatic memory management, failure detection, and failure recovery.

Automatic memory management is a system for dynamic memory allocation and reclamation of objects. The system must decide at run time whether an object is currently referenced by a live process, e.g., through reference counting [42].

Failure detection is the minimum functionality needed to ensure progress in the presence of failures. The system detects when a `DFut` cannot be dereferenced due to worker failure.

With distributed memory but no futures, this is straightforward because *the location of the value is known by the time the reference is created*. In Figure 4a, for example, the driver learns that `a` is stored on worker 1 and could then attach the location when passing `a` to worker 2. Then, when worker 2 receives `add`, it can detect `a`'s failure.

The addition of futures complicates failure detection because references can be created *before* the value. Even the future *location* of the value may not be known at reference creation time. Of course, the system could wait until a task has been scheduled before returning the reference to the caller. However, this would defeat the purpose of futures as an asynchronous construct. It is also impractical because a realistic scheduler must be able to update its decision at run time, e.g., according to changes in the environment such as resource availability and worker failures.

Thus, it is possible that there are no locations for `a` when worker 2 receives the `add` RPC in Figure 4b. Then, worker 2 must decide whether `f` is still executing, or if it has failed. If it is the former, then worker 2 should wait. But if there is a failure, then the system must recover `a`. To solve this problem, the system must record the locations of all *tasks*, i.e. pending objects, in addition to created objects.

Failure recovery. The system must also provide a method of recovering from a failed `DFut`. The minimum requirement is to throw an error to the application if it tries to dereference a failed `DFut`. We further provide an option for *transparent* recovery, i.e. the system will recover a failed `DFut`'s value.

With futures but no distributed memory, if a process fails, then we will lose the reply of any pending task on that process. Assuming idempotence, this can be recovered through retries, a common approach for pass-by-value RPC. For example, in Figure 5a, the driver recovers by resubmitting `add(a,b)`. Failure recovery is simple because *all data is passed by value*.

With distributed memory, however, tasks can also contain arguments passed by *reference*. Therefore, a node failure can cause the loss of an object value that is still referenced, as `b` is in Figure 4b. A common approach to this problem is to record each object's *lineage*, or the subgraph that produced the object, during runtime [17, 30, 56]. The system then walks a lost object's lineage and recursively reconstructs the object and its dependencies through task re-execution. This approach reduces the runtime overhead of logging, since the data itself is not recorded, and the work that must be redone after a partial failure, since objects cached in distributed memory do not need to be recomputed. Still, achieving low run-time overhead is difficult because the lineage itself must be recorded and collected at run time and it must survive failures.

Note that we focus specifically on *object recovery* and, like previous systems [34, 37, 56], assume *idempotence* for correctness. Thus, our techniques are directly applicable to idempotent functions and actors with read-only, checkpointable, or transient state, as we evaluate in Figure 15c. Although it is not our focus, these techniques may also be used in conjunction with known recovery techniques for actor state [17, 34] such as recovery for nondeterministic execution [52].

Metadata requirements. In summary, during normal operation, the system must at minimum record (1) the location(s) of each object's value, so that reference holders can retrieve it, and (2) whether the object is still referenced, for safe garbage collection. For failure detection and recovery, the system must further record, respectively, (3) the location of each *pending* object, i.e. the task location, and (4) the object lineage.

The key question is where and when to record this system metadata such that it is *consistent*[1] and *fault-tolerant*. By consistent, we mean that the system metadata matches the

[1]Unrelated to the more standard definition of replica consistency [50].

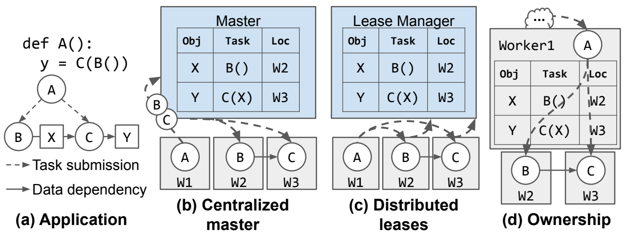


Figure 6: Distributed futures systems. **(a)** An application. **(b)** Master manages metadata and object failures. **(c)** Workers write metadata asynchronously, coordinate failure handling with leases. **(d)** Workers manage metadata. Worker 1 handles failures for workers 2 and 3. Worker 1 failure is handled by `A`'s owner elsewhere in the cluster.

current physical state of the cluster. By fault-tolerant, we mean that the metadata should survive individual node failures.

In some cases, it is safe for metadata to be *asynchronously updated*, i.e. there is a transient mismatch between the system metadata and the system state. For example, the system may transiently believe that an object `x` is still on node *A* even though it has been removed. This is safe because a reference holder can resolve the inconsistency by asking *A* if it has `x`.

On the other hand, metadata needed for failure handling should ideally be *synchronously updated*. For example, the metadata should never say that a task *T* is on node *A* when it is really on node *B*. In particular, if node *A* then fails, the system would incorrectly conclude that *T* has failed. As we will see next, synchrony simplifies fault tolerance but can add significant runtime overhead if done naively.

3.2 Existing solutions

Centralized master. Failure handling is simple with a synchronously updated centralized master, but this design can also add significant runtime overhead. For example, failure detection requires that the master record a task's scheduled location *before* dispatch (Figure 6b). Similarly, the master must record every new reference before it can be used. This makes the master a bottleneck for scalability and latency.

The master can be sharded for scalability, but this can complicate operations that coordinate multiple objects, such as garbage collection and lineage reconstruction. Also, the latency overhead is fundamental. Each task invocation must first contact the master, adding at minimum one round-trip to the critical path of execution, even without replicating the metadata for fault tolerance. This overhead can be detrimental when the task itself is milliseconds long, and especially so if the return value is small enough to be passed by value. Small values may be stored in the master directly as an optimization, but still require 1 RTT for retrieval [38].

Distributed leases. Decentralization can remove such bottlenecks, but often leads to complex coordination schemes. One approach is to use *distributed leases* [19]. This is similar to a centralized master that is updated *asynchronously*.

As an example, consider asynchronous task location updates (Figure 6c). To account for a possibly stale master, the worker nodes must coordinate to detect task failures, in this case using leases. Each worker node acquires a lease for each locally queued task and repeatedly renews the lease until the task has finished. For example, in Figure 6c, worker 3 can detect a failure of `B` by waiting for worker 2's lease to expire.

This design is horizontally scalable through sharding and reduces task latency, since metadata is written asynchronously. However, the reliance on timing to reconcile system state can slow recovery (Figure 14). Furthermore, this method of decentralization introduces a new problem: the workers must also coordinate on *who* should recover an object, i.e. re-execute the creating task. This is trivial in the centralized scheme, since the master coordinates all recovery operations.

3.3 Our solution: Ownership

The key insight in our work is to "shard" the centralized master, for scalability, but to do so based on the application structure, for low run-time overhead and simple failure handling. In ownership, the worker that calls a task stores the metadata related to the returned `DFut`. Like a centralized master, it coordinates operations such as task scheduling, to ensure it knows the task location, and garbage collection. For example, in Figure 6d, worker 1 owns `X` and `Y`.

The reason for choosing the task's caller as the owner is that in general, it is the worker that accesses the metadata most frequently. The caller is involved in the initial creation of the `DFut`, via task invocation, as well as the creation of other references, by passing the `DFut` to other RPCs. Thus, task invocation latency is minimal because the scheduled location is written locally. Similarly, if the `DFut` stays in the owner's scope, the overhead of garbage collection is low because the `DFut`'s reference count can be updated locally when the owner passes the `DFut` to another RPC. These overheads can be further reduced for small objects, which can be passed by value as if without distributed memory (see Section 4.2).

Of course, if all tasks are submitted by a single driver, as in BSP programs, ownership will not scale beyond the driver's throughput. Nor indeed will any system for dynamic tasks. However, with ownership, the *application* can scale horizontally by distributing its control logic across multiple nested tasks, as opposed to an application-agnostic method such as consistent hashing (Figure 12e). Furthermore, the worker processes hold much of the system metadata. This is in contrast to previous solutions that push all metadata into the system's centralized or per-node processes, limiting the *vertical* scalability of a single node with many worker processes (Figure 12).

However, there are problems that are simpler to solve with a fully centralized design, assuming sufficient performance:

First-class futures. First-class futures (Section 2) allow non-owning processes to reference a `DFut`. While many applications can be written without first-class futures (Figure 3b), they are sometimes essential for performance. For example, the model serving application in Figure 3a uses first-class futures to delegate task invocation to a nested task, without having to dereference and copy the arguments.

A first-class DFut may leave the owner's scope, so we must account for this during garbage collection. We avoid centralizing the reference count at the owner, as this would defeat the purpose of delegation. Instead, we use a distributed hierarchical reference counting protocol (Section 4.2). Each borrower stores a local reference count for the DFut on behalf of the owner (Table 2) and notifies the owner when the local reference count reaches zero. The owner decides when the object is safe to reclaim. We use a reference counting approach as opposed to tracing [42] to avoid global pauses.

Owner recovery. If a worker fails, then we will also lose its owned metadata. For transparent recovery, the system must recover the worker's state on a new process and reassociate state related to the previously owned DFuts, including any copies of the value, reference holders, and pending tasks.

We choose a minimal approach that guarantees progress, at the potential cost of additional re-execution on a failure: we *fate share* the object and any reference holders with the owner, then use *lineage reconstruction* to recover the object and any of the owner's fate-shared children tasks (Section 4.3). This method adds minimal run-time overhead and is correct, i.e. the application will recover to a previous state and the system guarantees against resource leakage. A future extension is to persist the owner's state to minimize recovery time at the cost of additional recovery complexity and run-time overhead.

4 Ownership Design

Each node in the cluster hosts one to many workers (usually one per core), one scheduler, and one object store (Figure 7). These processes implement future resolution, resource management, and distributed memory, respectively. Each node and worker process is assigned a unique ID.

Workers are responsible for the resolution, reference counting, and failure handling of distributed futures. Each worker executes one task at a time and can invoke other tasks. The root task is executed by the "driver".

Each task has a unique TaskID that is a hash of the parent task's ID and the number of tasks invoked by the parent task so far. The root TaskID is assigned randomly. Each task may return multiple objects, each of which is assigned an ObjectID that concatenates the TaskID and the object's index. A DFut is a tuple of the ObjectID and the owner's address (Owner).

The worker stores one record per future that it has in scope in its local *ownership table* (Table 2). A DFut borrower records a subset of these fields (* in Table 2). When a DFut is passed as an argument to a task, the system implicitly resolves the future's value, and the executing worker stores only the ID, Owner, and Value for the task duration. The worker also caches the owner's stored Locations.

Field	Value
*ID	The ObjectID. Also used as a distributed memory key.
*Owner	Address of the owner (IP address, port, WorkerID).
*Value	(1) Empty if not yet computed, (2) Pointer if in distributed memory, or (3) Inlined value, for small objects (Section 4.2).
*References	A list of reference holders: Number of dependent tasks and a list of borrower addresses (Section 4.2 and appendix A).
Task	Specification for the creating task. Includes the ObjectIDs and Owners of any DFuts passed as arguments.
Locations	If Value is empty, the location of the task. If Value is a pointer to distributed memory, then the locations of the object.

Table 2: Ownership table. The owner stores all fields. A borrower (Section 3.2) only stores fields indicated by the *.

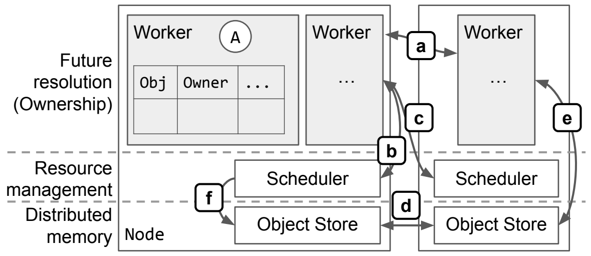


Figure 7: Architecture and protocol overview. **(a)** Task execution. **(b)** Local task scheduling. **(c)** Remote task scheduling. **(d)** Object transfer. **(e)** Task output storage and input retrieval. Ownership layer manages distributed memory garbage collection and recovery. **(f)** Scheduler fetches objects in distributed memory to fulfill task dependencies.

An actor is a stateful task that can be invoked multiple times. Like objects, an actor is created through task invocation and *owned* by the caller. The ownership table is also used to locate and manage actors: the Location is the actor's address. Like a DFut, an ARef (an actor reference) is a tuple of the ID and Owner and can be passed as a first-class value to other tasks.

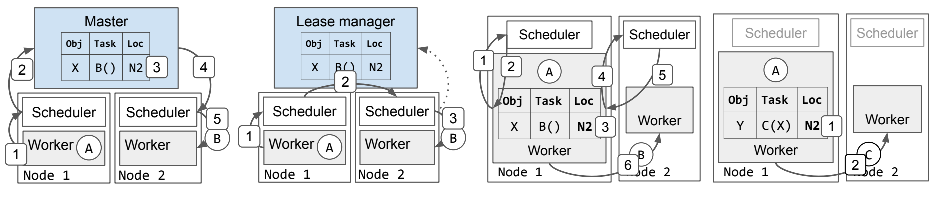
A worker requests resources from the scheduling layer to determine task placement (Section 4.1). We assume a decentralized scheduler for scalability: each scheduler manages local resources, can serve requests from remote workers, and can redirect a worker to a remote scheduler.

The distributed memory layer (Section 4.2) consists of an immutable distributed object store (Figure 7d) with Locations stored at the owner. The Locations are updated asynchronously. The object store uses shared memory to reduce copies between reference holders on the same node.

Workers store, retrieve, reclaim, and recover large objects in distributed memory (Figure 7f). The scheduling layer sends requests to distributed memory to fetch objects between nodes according to worker requests (Figure 7g).

4.1 Task scheduling

We describe how the owner coordinates task scheduling. At a high level, the owner dispatches each task to a location chosen by the distributed scheduler. This ensures that the task location in the ownership table is updated synchronously with dispatch. We assume an abstract scheduling policy that takes



	Node	L. RTTs	R. RTTs
Master	L.	1	1
	R.	1	1
Leases	L.	1	0
	R.	1	0.5
Owner-ship	L.	0.5 or 1.5	0
	R.	0 or 1	0.5 or 1.5

(e)

Figure 8: Task scheduling and the method of recording a task's location for the program in Figure 6a. **(a)** Centralized master. **(b)** Distributed leases. **(c)** Scheduling with ownership. (1-2) Local scheduler redirects owner to node 2. (3) Update task location. (4-5) Remote scheduler grants worker lease. (6) Task dispatch. **(d)** Direct scheduling by the owner, using the worker and resources leased from node 2 in (c). **(e)** Length of critical path of local (L) and remote (R) task execution, in terms of local (L) and remote (R) RTTs.

in resource requests and returns the ID of a node where the resources should be allocated. The policy may also update its decision, e.g., due to changes in resource availability.

Figure 8c shows the protocol to dispatch a task. Upon task invocation, the caller, i.e. the owner of the returned `DFut`, first requests resources from its local scheduler[2]. The request is a tuple of the task's required resources (e.g., `{"CPU": 1}`) and arguments in distributed memory. If the policy chooses the local node, the scheduler accepts the request: it fetches the arguments, allocates the resources, then *leases* a local worker to the owner. Else, the scheduler rejects the request and redirects the owner to the node chosen by the policy.

In both cases, the scheduler responds to the owner with the new location: either the ID of the leased worker or the ID of another node. The owner stores this new location in its local ownership table before dispatching the task to that location. If the request was granted, the owner sends the task directly to the leased worker for execution; otherwise, it repeats the protocol at the next scheduler.

Thus, the owner always dispatches the task to its next location, ensuring that the task's pending `Location` (Table 2) is synchronously updated. This also allows the owner to *bypass* the scheduler by dispatching a task directly to an already leased worker, if the task's resource requirements are met. For example, in Figure 8d, worker 1 reuses the resources leased from node 2 in Figure 8c to execute `C`. The owner returns the lease after a configurable expiration time, or when it has no more tasks to dispatch. We currently do not reuse resources for tasks with different distributed memory dependencies, since these are fetched by the scheduler. We leave other policies for lease revocation and worker reuse for future work.

The *worst-case* number of RTTs before a task executes is higher than in previous solutions because each policy decision is returned to the owner (Figure 8e). However, the throughput of previous solutions is limited (Figure 12) because they cannot support direct worker-to-worker scheduling (Figure 8d). This is because workers do not store system state, and thus all tasks must be routed through the master or per-node scheduler to update the task location (Figures 8a and 8b).

Actor scheduling. The system schedules actor constructor tasks much like normal tasks. After completion, however, the owner holds the worker's lease until the actor is no longer referenced (Section 4.2) and the worker can only execute actor tasks submitted through a corresponding `ARef`.

A caller requests the actor's location from the owner using the `ARef`'s `Owner` field. The location can be cached and requested again if the actor restarts (Section 4.3). The caller can then dispatch tasks directly to the actor, as in Figure 8d, since the resources are leased for the actor's lifetime. For a given caller, the actor executes tasks in the order submitted.

4.2 Memory management

Allocation. The distributed memory layer consists of a set of object store nodes, with locations stored at the owner (Figures 9b to 9d). It exposes a key-value interface (Figure 9a). The object store may replicate objects for efficiency but is not required to handle recovery: if there are no copies of an object, a `Get` call will block until a client (i.e. a worker) `Creates` the object.

Small objects may be faster to copy than to pass through distributed memory, which requires updating the object directory, fetching the object from a remote node, etc. Thus, at object creation time, the system transparently chooses based on size whether to pass by value or by reference.

Objects over a configurable threshold are stored in the distributed object store (step 1, Figure 9b) and returned by reference to the owner (step 2). This reduces the total number of copies, at the cost of requiring at least one IPC to the distributed object store for `Get` (steps 4-5, Figure 9c). Small objects are returned by value to the owner (step 6, Figure 9c), and each reference holder is given its own copy. This produces more copies in return for faster dereferencing.

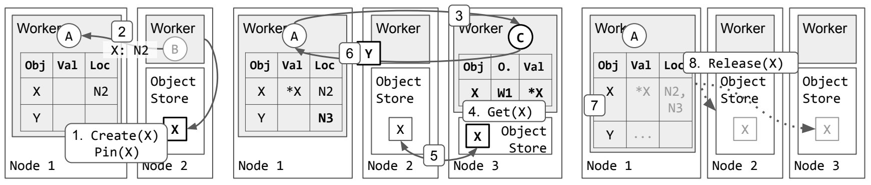
The initial copy of a large object is known as the *primary*. This copy is pinned (step 1, Figure 9b) until the owner releases the object (step 8, Figure 9d) or fails. This allows the object store to treat additional capacity as an LRU cache without having to consult the owners about which objects are safe to evict. For example, the *secondary* copy of `X` created on node 3 in Figure 9c is cached to reduce `Get` and recovery time (Section 4.3) but can be evicted under memory pressure.

Dereferencing. The system dereferences a task's `DFut` arguments before execution. The task's caller first waits for the `Value` field in its local ownership table to be populated (Fig-

[2]The owner can also choose a remote scheduler, e.g., for data locality.

Operation	Semantics
`Create(ObjID o, Value v)`	Store an object.
`Pin(ObjID o, NodeID loc)` → `bool`	Pin `o` on `loc` until released. Returns false if `loc` failed.
`Release(ObjID o)`	Object `o` is safe to evict.
`Get(ObjID o)` → `Value`	Get the object value. May fetch copy from remote node.

(a)



(b) (c) (d)

Figure 9: **(a)** Distributed memory store API, and **(b-d)** Memory management for the program in Figure 6a. (1-2) `B` returns a large object `X` in distributed memory. The primary copy is pinned until all references have been deleted. (3) Worker 1 dispatches `C` once `X` is available. (4-5) `Get` the value from distributed memory (location lookup not shown). (6) `C` returns a small object `Y` directly to the owner. (7-8) Object reclamation.

ure 9b), then copies the `Value` into the dispatched task description. The executing worker then copies the received `Value` into its local table (Figure 9c). For large objects, the sent value is a pointer to distributed memory, so the worker must also call `Get` to retrieve the actual value (step 4, Figure 9c).

If the task's caller is also the owner of its `DFut` arguments, the above protocol is sufficient. If the task's caller is *borrowing* an argument, then it must populate the `Value` field through a protocol with the owner. Upon receiving a `DFut`, the borrower sends the associated `Owner` a request for the `Value`. The owner replies with the `Value` (either the inlined value or a pointer) once populated. The borrower populates its local `Value` field by copying the reply.

Reclamation. The owner reclaims the object memory once there are no more reference holders (Figure 9d) by deleting its local `Value` field (step 7) and, if necessary, calling `Release` on the distributed object store (step 8). An object's reference holders are tracked with a distributed reference count maintained by the owner and borrowers.

Each process with a `DFut` instance keeps a local count of submitted tasks (`References`, Table 2). The task count is incremented each time the process invokes a dependent task and decremented when the task completes. Each process also keeps a local set of the worker IDs of any borrowers that it created, by passing the `DFut` as a first-class value. This forms a tree of borrowers with the owner at the root (see Appendix A). The owner releases the object once there are no more submitted tasks or borrowers anywhere in the cluster.

Actors. Actors are reference-counted with the same protocol used to track borrowers of a `DFut`. Once the set of reference holders is empty, the owner of the actor reclaims the actor resources by returning the worker lease (Section 4.1).

4.3 Failure recovery

The system guarantees that any reference holder will eventually be able to resolve the value in the presence of failures.

Failure detection. Failure notifications containing a worker or node ID are published to all workers. Workers do not exchange heartbeats; a worker failure is published by its local scheduler. Node failure is detected by exchanging heartbeats between nodes, and all workers fate-share with their node.

Upon receiving a node or worker failure notification, each worker scans its local ownership table to detect a `DFut` failure. A `DFut` is considered failed in two cases: 1) loss of an owned object (Figure 10a), by comparing the `Location` field, or 2) loss of an owner (Figure 11a), by comparing the `Owner` field. We discuss the handling for these two cases next, using lineage reconstruction and fate sharing, respectively.

Note that a non-owner does not need to detect the loss of an object. For example, in Figure 10a, node 2 fails just as worker 3 receives `C`. When worker 3 looks up `X` at the owner, it may not find any locations. From worker 3's perspective, this means that either node 2's write to the directory was delayed, or node 2 failed. Worker 3 does not need to decide which it is; it simply waits for `X`'s owner to handle the failure.

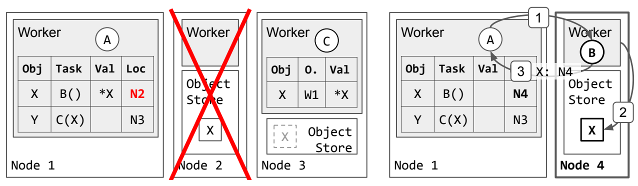
Object recovery. The owner recovers a lost value through lineage reconstruction. During execution, the owner records the object's lineage by storing each invoked `Task` in its ownership table (Table 2). Then, upon detecting a `DFut` failure, the owner resubmits the corresponding task (Figure 10b). The task's arguments are recursively reconstructed, if needed.

Like previous systems [34, 37, 56], we can avoid lineage reconstruction if other copies of a required object still exist. Thus, when reconstructing an object, the owner will first try to locate and designate a secondary copy as the new primary. To increase the odds of finding a secondary copy, object reclamation (Section 4.2) is done lazily: the owner releases the primary copy once there are no more reference holders, but the copy is not evicted until there is memory pressure.

Often, the owner of an object will also own the objects in its lineage (Section 5.2). Thus, upon failure, the owner can locally determine the set of tasks to resubmit, with a recursive lookup of the `Task` fields. In some cases, an object's lineage may also contain *borrowed* references. Then, the borrower requests reconstruction from the owner.

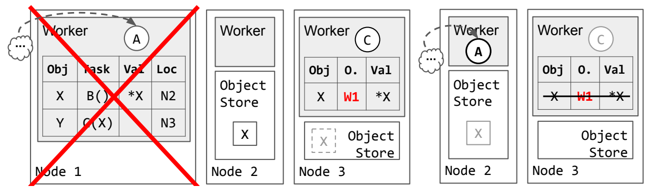
The owner can delete the `Task` field once the task has finished and all objects returned by reference will never be reconstructed again. When a worker returns an object by value, the owner can immediately delete the corresponding `Task` field. This is safe because objects passed by value do not require reconstruction (Section 3.1).

For an object passed by reference, the owner keeps a *lineage reference count* to determine when to collect the `Task`. The



(a) Failure detection. (b) Lineage reconstruction.

Figure 10: Object recovery.



(a) Failure detection. (b) Fate sharing.

Figure 11: Owner recovery.

count is incremented each time the `DFut` is passed to another task and decremented when that `Task` is itself collected. The owner collects a record after collecting both the `Task` and `Value` (Section 4.2) fields. We also plan to support object checkpointing to allow the lineage to be collected early.

Owner recovery. An owner failure can result in a "dangling pointer": a `DFut` that cannot be dereferenced. This can happen if the object is simultaneously lost from distributed memory. For example, `C` in Figure 11a will hang if node 2 also fails.

We use *fate sharing* to ensure that the system can make progress upon an owner's failure. First, all resources held by the owner and any reference holders are reclaimed. Specifically, upon notification of the owner's failure, either the distributed object store frees the object (if it exists) or the scheduling layer reclaims the worker lease (if the object is pending), shown in Figure 11b. All reference holders, i.e. borrowers and dependent tasks, also fate-share with the owner.

Then, to recover the fate-shared state, we rely on *lineage reconstruction*. In particular, the task or actor that was executing on the failed owner must itself have been owned by another process. That process will eventually resubmit the failed task. As the new owner re-executes, it will recreate its previous state, with no system intervention needed. For example, the owner of `A` in Figure 11a will eventually resubmit `A` (Figure 11b), which will again submit `B` and `C`.

For correctness, we show that all previous reference holders are recreated, with the address of the new owner. Consider task T that computes the value of a `DFut` x. T initially executes on worker W and re-executes on W' during recovery. The API (Section 2) gives three ways to create another reference to x: (1) pass x as a task argument, (2) cast x to a `SharedDFut` then pass as a task argument, and (3) return x from T.

In the two former cases, the new reference holder must be a child task of T. In case (2), when x is passed as a first-class value, the child task can create additional reference holders by passing x again. All such reference holders are therefore descendants of T. Then, when T re-executes on W', W' will recreate T's descendants.

T can also return x, which can be useful for returning a child task's result without dereferencing with `get`. Suppose T returns x to its parent task P. Then, P's worker becomes a borrower and will fate-share with W. In this case, P is recovered by *its* owner, and again submits T and receives x.

Thus, because any borrower of `x` must be a child or ancestor of T, fate-sharing and re-execution guarantees that the borrower will be recreated with W' as the new owner. Note that for actors, this requires that an actor not store borrowed `DFuts` in its local state. Of course, this is only required for transparent recovery; the application may also choose to handle failures manually and rely on the system for failure detection only.

While fate-sharing and lineage reconstruction add minimal run-time overhead, it is not suitable for all applications. In particular, the application will fate-share with the driver. In fact, this is the same failure model offered by some BSP systems [3], which can be written as a distributed futures program in which the driver submits all tasks. As shown by these systems, this approach can be extended to reduce the re-execution needed during recovery. We leave such extensions, including application-level checkpointing (Section 5.2), and persistence of the ownership table, for future work.

Actor recovery. Actor recovery is handled through the same protocols. If an actor fails, its owner restarts the actor through lineage reconstruction, i.e. resubmitting the constructor task. If the owner fails, the actor and any `ARef` holders fate-share.

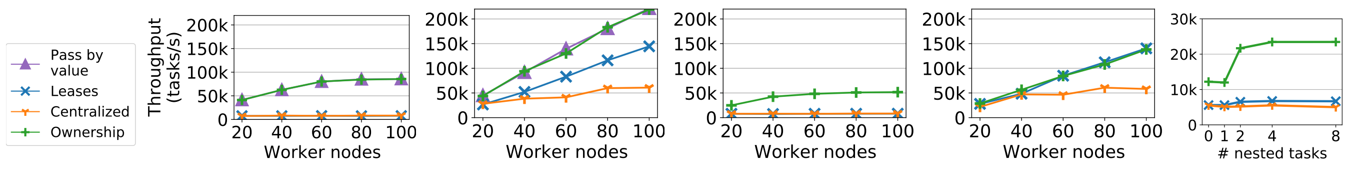
Unlike functions, actors have local state that may require recovery. This is out of scope for this work, but is an interesting future direction. Ownership provides the infrastructure to manage and restart actors, while other methods can be layered on top for transparent recovery of local state [17, 34, 52].

5 Evaluation

We study the following questions:

1. Under what scenarios is distributed futures beneficial compared to pass-by-value RPC?
2. How does the ownership architecture compare against existing solutions for distributed futures, in terms of throughput, latency, and recovery time?
3. What benefits does ownership provide for applications with dynamic, fine-grained parallelism?

We compare against three baselines: (1) a pass-by-value model with futures but no distributed memory, similar to Figure 2c, (2) a decentralized lease-based system for distributed futures (Ray v0.7), and (3) a centralized master for distributed futures (Ray v0.7 modified to write to a centralized master before task execution). All distributed futures systems use sharded, unreplicated Redis for the global metadata store, with asynchronous requests. All systems use the Ray distributed scheduler and (where applicable) distributed object store. Ownership and pass-by-value use gRPC [2] for worker-



(a) Small objects, colocated. (b) Small objects, spread. (c) Large objects, colocated. (d) Large objects, spread. (e) Single node, nested tasks.

Figure 12: Throughput and scalability. **(a-d)** Task submission is divided across multiple intermediate drivers, either colocated on the m5.8xlarge head node or spread with one m5.8xlarge node per driver. 1 intermediate driver is added per 5 worker nodes. Each task returns either a small (short binary string) or large (1MB blob) object. **(e)** Scaling task submission using nested tasks and first-class distributed futures.

to-worker communication. All benchmarks schedule tasks to predetermined nodes to reduce scheduling variation.

All experiments are run on AWS EC2. Global system metadata, such as an object directory, is hosted on the same node as the driver, where applicable. Unless stated otherwise, this "head node" is an m5.16xlarge instance. Other node configuration is listed inline. All benchmark code is available at [53].

5.1 Microbenchmarks

Throughput and scalability. The driver submits one nested task for every 5 worker nodes (m5.8xlarge). Each intermediate "driver" submits no-op tasks to its 5 worker nodes. We report the total throughput of the leaf tasks, which return either a short string (Figures 12a and 12b) or a 1MB blob (Figures 12c and 12d). The drivers are either colocated (Figures 12a and 12c) on the same m5.8xlarge node as the root driver, or spread (Figures 12b and 12d), each on its own m5.8xlarge node. We could not produce stable results for pass-by-value with large objects due to the lack of backpressure in our implementation.

At <60 nodes, the centralized and lease-based architectures achieve about the same throughput because the centralized master is not yet a bottleneck. In general, ownership achieves better throughput than either because it distributes some system operations to the workers. In contrast, the baselines handle all system operations in the global or per-node processes.

The gap between ownership and the baselines is more significant with small return values (Figures 12a and 12b). For these, ownership matches pass-by-value because small objects are returned directly to their owner. The baseline systems could implement a similar optimization, e.g., by inlining small objects in the object directory (Section 4.2), but this would still require at minimum one RPC per read.

When the drivers are spread (Figures 12b and 12d), ownership and leases both scale linearly. Ownership scales better than leases in Figure 12b because more work is offloaded onto the worker processes. Ownership and leases achieve similar throughput in Figure 12d, but the ownership system also includes memory safety (Section 4.2). The centralized design (2 shards) scales linearly to ∼60 nodes. Adding more shards would raise this threshold, but only by a constant amount.

When the drivers are colocated (Figures 12a and 12c), both baselines flatline because of a centralized bottleneck: the scheduler on the drivers' node. Ownership also shows this, but there is less scheduler load overall because the drivers reuse resources for multiple tasks (Section 4.1). A comparable optimization for the baselines would require each driver to batch task submission, at the cost of latency. Throughput for ownership is lower in Figure 12c than in Figure 12a due to the overhead of garbage collection.

Thus, because ownership decentralizes system state among the workers, it can achieve vertical (Figures 12a and 12c) and horizontal (Figures 12b and 12d) scalability. Also, it matches the performance of pass-by-value RPC while enabling new workloads through distributed memory (Section 2.2).

Scaling through borrowing. We show how first-class futures enable delegation. Figure 12e shows the task throughput for an application that submits 100K no-op tasks that each depend on the same 1MB object created by the driver. The tasks are submitted either by the driver (x=0) or by a number of nested tasks that each *borrow* a reference to the driver's object. All workers are colocated on an m5.16xlarge node.

For all systems, the throughput with a single borrower (x=1) is about the same as when the driver submits all tasks directly (x=0). Distributing task submission across multiple borrowers results in a 2× improvement for ownership and negligible improvement for the baselines. Thus, with ownership, an application can scale past the task dispatch throughput of a single worker by *delegating* to nested tasks. This is due to (1) support for first-class distributed futures, and (2) the hierarchical distributed reference counting protocol, which distributes an object's reference count among its borrowers instead of centralizing it at the owner (Section 4.2). In contrast, the baselines would require additional nodes to scale.

Latency. Figure 13 measures task latency with a single worker, hosted either on the same node as the driver ("local"), or on a separate m5.16xlarge node ("remote"). The driver submits 3k tasks that each take the same 1MB object as an argument and that immediately returns a short string. We report the average duration before each task starts execution.

First, distributed memory achieves better latency than pass-by-value in all cases because these systems avoid unnecessary copies of the task argument from the driver to the worker.

Second, compared to centralized and leases, ownership achieves on average 1.6× lower latency. This is due to (1)

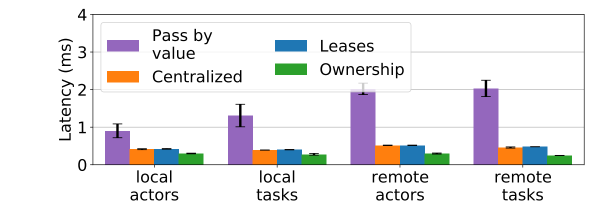
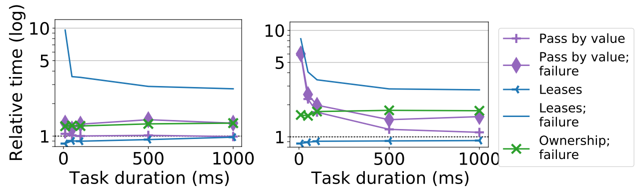


Figure 13: Task latency. Local means that the worker and driver are on the same node. Error bars for standard deviation (across 3k tasks).



(a) Small objects. (b) Large objects.

Figure 14: Total run time (log-scale), relative to ownership without failures. The application is a chain of dependent tasks that execute on one node. Each task sleeps for the duration on the x-axis (total 10s) and returns either **(a)** a short binary string, or **(b)** a 10MB blob.

the ability to write metadata locally at the owner instead of a remote process, and (2) the ability to reuse leased resources, in many cases bypassing the scheduling layer (Section 4.1).

Recovery. This benchmark submits a chain of tasks that execute on a remote m5.xlarge node. Each task depends on the previous, sleeps for the time on the x-axis (total duration 10s), and returns either a short binary string (Figure 14a) or a 10MB blob (Figure 14b). We report the run time relative to ownership without failures. To test recovery, the worker node is killed and restarted 5s into the job (1s heartbeat timeout). We do not include centralized due to implementation effort.

Normal run time for leases is up to 1.18× faster than ownership, but recovery time is more than double, worse than restarting the application. This is because a task's lease must expire before it can be re-executed, adding delay for short tasks. The recovery delay for longer tasks is also high because the implementation (Ray v0.7) repeatedly doubles a lease's expiration time to reduce renewal overhead. A shorter lease interval would reduce recovery delay but can be unstable.

Ownership recovers within 2× the normal run time. Recovery time is the same as pass-by-value for small objects because only in-flight tasks are re-executed (Figure 14a). For large objects (Figure 14b), ownership achieves better normal run time than pass-by-value because arguments are passed by reference; the gap decreases as task execution dominates.

Thus, ownership can achieve the same or better normal run-time performance as leases *and* pass-by-value, while also guaranteeing timely recovery through lineage reconstruction.

5.2 End-to-end applications

Model serving. We implement Figure 3a. Figure 15a shows the latency on 4 p3.16xlarge nodes, each with 1 `Router` and 8 ResNet-50 [23] `Models`. We use a GPU batch size of 16 and generate 2300 requests/s. Ownership and centralized achieve the same median latency (54ms), but the tail latency for centralized is 9× higher (1s vs. 108ms). We also show the utility of first-class distributed futures: in "-borrow", the `Router` receives the image *values* and must copy these to the `Model`. As expected, the `Router` is a bottleneck (p50=80ms, p100=3.2s).

Online video processing. We implement Figure 3b with 60 concurrent videos. The tasks for each stream are executed on an m5.xlarge "worker" node (1 per stream) and submitted by a driver task on a separate m5.xlarge "owner" node. Each owner node hosts 4 drivers. Each video source uses an actor to hold frame-to-frame decoder state. However, tasks are idempotent: a previous frame may be reread with some latency penalty. We use a YouTube video with a frame rate of 29 frames/s and a radius of 1s for the moving average.

Figure 15b shows latency without failures. All systems achieve similar median latency (∼65ms), but leases and centralized have a long tail (1208ms and 1923ms, respectively). Figure 15c shows latency during an injected failure, 5s after the start, of the `Decoder` actor (Figure 3b). Lease-based recovery is slow because the decoder actor must replay all tasks, and each task accumulates overhead from lease expiration. Checkpointing the actor was infeasible because the leases implementation does not safely garbage-collect lineage.

Figure 15c also shows different failure scenarios for ownership, with a failure after 10s. The owner uses lineage reconstruction to recover quickly from a worker failure (1.9s in O;WF). Owner recovery is slower because the failed owner must re-execute from the beginning (8.8s in O;OF). To bound re-execution, we use application-level checkpoints (O+CP, checkpoints to a remote Redis instance once per second). Each checkpoint includes all intermediate state needed to transform the given frame, such as the cumulative sum so far (Figure 3b). When the sink receives the transformed frame, it "commits" the checkpoint by writing the frame's index to Redis. This results in negligible overhead (O vs. O+CP) and faster recovery (1.1s in O+CP;OF).

6 Related Work

Distributed futures. Several systems [4, 34, 37, 45, 48, 52] have implemented a distributed futures model. Most [37, 45] use a centralized master (Section 3.2). In contrast, ownership is a decentralized design that stores system state directly in the workers that invoke the tasks. Ray [34] shards the centralized state, but must still write to the centralized store *before* task execution and does not support automatic memory management. Lineage stash [52] is a complementary technique for recovering nondeterministic execution; ownership provides infrastructure for failure detection and memory management.

Other dataflow systems. Distributed data-parallel systems provide high-throughput batch computation and transparent data recovery [15, 25, 54, 56]. Many of our techniques build on

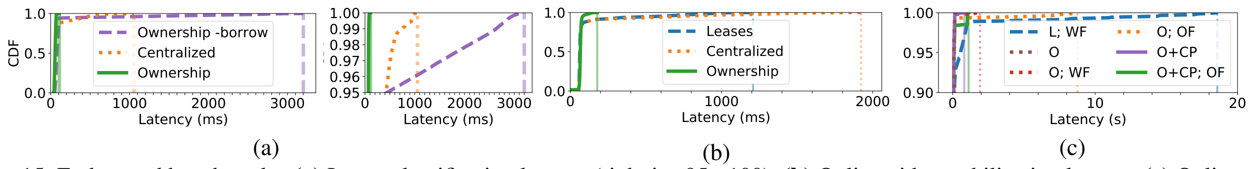


Figure 15: End-to-end benchmarks. **(a)** Image classification latency (right is p95-p100). **(b)** Online video stabilization latency. **(c)** Online video stabilization latency with failures (starting at p90). L=leases; O=ownership; CP=checkpointing; WF=worker failure; OF=owner failure.

these systems, in particular the use of distributed memory [25, 56] and lineage re-execution [15, 25, 54, 56]. Indeed, a data-parallel program is equivalent to a distributed futures program with no nested functions.

Most distributed data-parallel systems [15, 25, 54, 56] employ some form of centralized master, a bottleneck for applications with *fine-grained* tasks [32, 44, 51]. Naiad [35, 36] and Canary [44] support fine-grained tasks but, like other data-parallel systems, implement a *static* task graph, i.e. all tasks must be specified upfront. In contrast, distributed futures are an extension of RPC, which allows tasks to be dynamically invoked. Nimbus [32] supports both fine-grained *and* dynamic tasks with a centralized controller by leveraging *execution templates* for iterative computations. In contrast, ownership distributes the control plane and schedules tasks one at a time. These approaches are complementary; an interesting future direction is to apply execution templates to distributed futures.

Actor systems. Distributed futures are compatible with the actor model [7, 24]. Other actor frameworks [1, 12] already use futures for asynchrony, but with pass-by-value semantics, making it expensive to process large data. Actors can be extended with distributed memory to enable pass-by-reference semantics. Since distributed memory is immutable, it does not violate the condition of no shared state.

Our fault tolerance model is inspired by *supervision* in actor systems [7]. In this model, a supervisor actor delegates work to its children actors and is responsible for handling any failures among its children. By default, an actor also fate-shares with its supervisor. Our contribution is in extending the supervision model to *objects* and object recovery.

Parallel programming systems. MPI [18] exposes a low-level pass-by-value interface. In contrast, distributed futures supports pass-by-reference and heterogeneous processes.

Distributed futures are more similar in interface to other parallel programming runtimes [10, 14, 21, 31, 47]: the user annotates a sequential program to designate procedures that can be executed in parallel. Out of these systems, ownership is perhaps most similar to Legion [10], in that the developer specifies a task hierarchy that dictates system behavior. Our contribution is in identifying and addressing the challenges of failure detection and recovery for distributed futures.

Distributed memory. Distributed shared memory [40] provides the illusion of a single globally shared and *mutable* address space across a physically distributed system. Transparency has historically been difficult to achieve without adding exorbitant runtime overhead. Mutability makes consistency a major problem [11, 26, 28, 40], and fault tolerance has never been satisfactorily addressed [40].

More recent distributed memory systems [6, 9, 16, 27, 41] implement a higher-level key-value store interface. Most target a combination of performance, consistency, and durability. Similar to our use of distributed memory (Section 4.2), in-memory data replicas are used to improve durability and recovery time. Indeed, many of these systems could likely be used in place of our distributed memory subsystem.

However, the requirements of our distributed memory subsystem are minimal compared to previous work, e.g., durability is only an optimization. This is because we target an even higher-level interface that integrates directly with the programming language: unlike a key, a `DFut` can be used to express rich application semantics to the system, such as an RPC's data dependencies. Also, like previous data processing systems [15, 37, 56], data is immutable. Thus, fine-grained mutations are expensive, but consistency is not a problem.

7 Discussion

Ownership is the basis of the Ray architecture in v1.0+ [5], implemented in ∼14k C++ LoC. Previously, Ray used a sharded global metadata store [34]. There were two problems with this approach: (1) latency, and (2) worker nodes still had to coordinate for operations such as failure detection. Ray v0.7 introduced leases (Section 3.2), which solved the latency problem but not coordination. It became impractical to introduce distributed protocols involving multiple objects, such as for garbage collection. We designed ownership for this purpose.

While transparent recovery is an explicit goal of this paper, it is not the only benefit of ownership. Anecdotally, the two main benefits of ownership for Ray users are performance and reliability. In particular, reliability includes correct and timely failure detection and garbage collection. Notably, ownership-based transparent recovery is not yet widely used.

We believe that this is due to: (1) applications having custom recovery requirements that cannot be met with lineage reconstruction alone, and (2) the cost of transparent recovery. Thus, one design goal was to ensure that only applications that needed transparent recovery would have to pay the cost. Ownership is a first step towards this: it provides reliability to all applications and transparent object recovery as an option.

In the future, we hope to extend this work to support a *spectrum* of application recovery requirements. For example, we could extend ownership with options to recover actor state.

Acknowledgements

We thank our anonymous reviewers and our shepherd Ryan Huang for their insightful feedback. We also thank Alvin Cheung, Michael Whittaker, Joe Hellerstein, and many others at the RISELab for their helpful discussion and comments. In addition to NSF CISE Expeditions Award CCF-1730628, this research is supported by gifts from Alibaba Group, Amazon Web Services, Ant Group, CapitalOne, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk, and VMware.

References

[1] Akka. https://akka.io/.

[2] gRPC. https://grpc.io.

[3] Improved Fault-tolerance and Zero Data Loss in Apache Spark Streaming. https://databricks.com/blog/2015/01/15/improved-driver-fault-tolerance-and-zero-data-loss-in-spark-streaming.html.

[4] PyTorch - Remote Reference Protocol. https://pytorch.org/docs/stable/notes/rref.html.

[5] Ray v1.0. https://github.com/ray-project/ray/releases/tag/ray-1.0.0.

[6] David G Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. Fawn: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 1–14, 2009.

[7] Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, Mikroelektronik och informationsteknik, 2003.

[8] Henry C Baker Jr and Carl Hewitt. The incremental garbage collection of processes. *ACM SIGART Bulletin*, (64):55–59, 1977.

[9] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobbler, Michael Wei, and John D Davis. {CORFU}: A shared log design for flash clusters. In *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, pages 1–14, 2012.

[10] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2012.

[11] John K Bennett, John B Carter, and Willy Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 168–176, 1990.

[12] Phil Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. Orleans: Distributed virtual actors for programmability and scalability. Technical Report MSR-TR-2014-41, March 2014.

[13] Andrew D Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1):39–59, 1984.

[14] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing*, 37(1):55–69, 1996.

[15] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[16] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, pages 401–414, 2014.

[17] Elmootazbellah Nabil Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408, 2002.

[18] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[19] Cary Gray and David Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. *ACM SIGOPS Operating Systems Review*, 23(5):202–210, 1989.

[20] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving dnns like clockwork: Performance predictability from the bottom up. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 443–462, 2020.

[21] Robert H Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(4):501–538, 1985.

[22] Brandon Haynes, Amrita Mazumdar, Armin Alaghi, Magdalena Balazinska, Luis Ceze, and Alvin Cheung. Lightdb: A DBMS for virtual reality video. *Proc. VLDB Endow.*, 11(10):1192–1205, 2018.

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[24] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, page 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[25] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[26] Pete Keleher, Alan L Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. *Distributed Shared Memory: Concepts and Systems*, pages 211–227, 1994.

[27] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[28] Kai Li. Ivy: A shared virtual memory system for parallel computing. *ICPP (2)*, 88:94, 1988.

[29] Barbara Liskov and Liuba Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. *ACM SIGPLAN Notices*, 23(7):260–267, 1988.

[30] Nirmesh Malviya, Ariel Weisberg, Samuel Madden, and Michael Stonebraker. Rethinking main memory oltp recovery. In *2014 IEEE 30th International Conference on Data Engineering*, pages 604–615. IEEE, 2014.

[31] Simon Marlow. *Parallel and concurrent programming in Haskell: Techniques for multicore and multithreaded programming*. " O'Reilly Media, Inc.", 2013.

[32] Omid Mashayekhi, Hang Qu, Chinmayee Shah, and Philip Levis. Execution templates: Caching control plane decisions for strong scaling of data analytics. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pages 513–526, 2017.

[33] Luc Moreau. Hierarchical distributed reference counting. In *Proceedings of the 1st international symposium on Memory management*, pages 57–67, 1998.

[34] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, Carlsbad, CA, 2018. USENIX Association.

[35] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA, 2013. ACM.

[36] Derek G. Murray, Frank McSherry, Michael Isard, Rebecca Isaacs, Paul Barham, and Martin Abadi. Incremental, iterative data processing with timely dataflow. *Commun. ACM*, 59(10):75–83, September 2016.

[37] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. CIEL: A universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 113–126, Berkeley, CA, USA, 2011. USENIX Association.

[38] D.G. Murray. *A Distributed Execution Engine Supporting Data-dependent Control Flow*. University of Cambridge, 2012.

[39] Robert Nishihara, Philipp Moritz, Stephanie Wang, Alexey Tumanov, William Paul, Johann Schleier-Smith, Richard Liaw, Mehrdad Niknami, Michael I. Jordan, and Ion Stoica. Real-time machine learning: The missing pieces. In *Workshop on Hot Topics in Operating Systems*, 2017.

[40] B. Nitzberg and V. Lo. Distributed shared memory: a survey of issues and algorithms. *Computer*, 24(8):52–60, 1991.

[41] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, et al. The RAMCloud storage system. *ACM Transactions on Computer Systems (TOCS)*, 33(3):7, 2015.

[42] David Plainfossé and Marc Shapiro. A survey of distributed garbage collection techniques. In *International Workshop on Memory Management*, pages 211–249. Springer, 1995.

[43] Alex Poms, Will Crichton, Pat Hanrahan, and Kayvon Fatahalian. Scanner: Efficient video analysis at scale. *ACM Trans. Graph.*, 37(4):138:1–138:13, July 2018.

[44] Hang Qu, Omid Mashayekhi, Chinmayee Shah, and Philip Levis. Decoupling the control plane from program control flow for flexibility and performance in cloud computing. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[45] Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In Kathryn Huff and James Bergstra, editors, *Proceedings of the 14th Python in Science Conference*, pages 130 – 136, 2015.

[46] Danilo Sato, Arif Wider, and Windheuser Christoph. Continuous delivery for machine learning, Sep 2019.

[47] Elliott Slaughter, Wonchan Lee, Sean Treichler, Michael Bauer, and Alex Aiken. Regent: a high-productivity programming language for hpc with logical regions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015.

[48] Vikram Sreekanti, Chenggang Wu Xiayue Charles Lin, Jose M Faleiro, Joseph E Gonzalez, Joseph M Hellerstein, and Alexey Tumanov. Cloudburst: Stateful functions-as-a-service. *arXiv preprint arXiv:2001.04592*, 2020.

[49] Vikram Sreekanti, Harikaran Subbaraj, Chenggang Wu, Joseph E Gonzalez, and Joseph M Hellerstein. Optimizing prediction serving on low-latency serverless dataflow. *arXiv preprint arXiv:2007.05832*, 2020.

[50] Andrew S Tanenbaum and Maarten Van Steen. *Distributed systems: principles and paradigms*. Prentice-Hall, 2007.

[51] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Ali Ghodsi, Michael Armbrust, Benjamin Recht, Michael Franklin, and Ion Stoica. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the Twenty-Sixth ACM Symposium on Operating Systems Principles*, SOSP '17. ACM, 2017.

[52] Stephanie Wang, John Liagouris, Robert Nishihara, Philipp Moritz, Ujval Misra, Alexey Tumanov, and Ion Stoica. Lineage stash: fault tolerance off the critical path. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 338–352, 2019.

[53] Stephanie Wang, Edward Oakes, and Frank Luan. Ownership nsdi'21 artifact. `https://github.com/stephanie-wang/ownership-nsdi2021-artifact`.

[54] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2012.

[55] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking tla+ specifications. In *In Correct Hardware Design and Verification Methods (CHARME '99), Laurence Pierre and Thomas Kropf editors. Lecture Notes in Computer Science, Springer-Verlag.*, volume 1703, pages 54–66, June 1999.

[56] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

A Distributed Reference Counting

Type	Description
Local reference	A flag indicating whether the `DFut` has gone out of the process's scope.
Submitted task count	Number of tasks that depend on the object that were submitted by this process and that have not yet completed execution.
Borrowers	The set of worker IDs of the borrowers created by this process, by passing the `DFut` as a first-class value.
Nested `DFuts`	The set of `DFuts` that are in scope and whose values contain this `DFut`.
Lineage count	Number of `Tasks` that depend on this `DFut` that may get re-executed. This count only determines when the lineage (the `Task` field) should be released; the *value* can be released even when this count is nonzero.

Table 3: Full description of the `References` field in Table 2. Every process with an instance of the `DFut` (either the owner or a borrower) maintains these fields.

If a `DFut` never leaves the scope of its owner, it does not require a distributed reference count. This is because the owner always has full information about which pending tasks require the object. However, since our API allows passing `DFuts` to other tasks as first-class values, we use a distributed reference count to decide when the object is out of scope.

Our reference counting protocol is similar to existing solutions [33, 42]. As explained in Section 4.2, the reference count is maintained with a tree of processes. Each process keeps a local set of borrower worker IDs, i.e. its children nodes in the tree. Most of the messages needed to maintain the tree are piggy-backed on existing protocols, such as for task scheduling.

A borrower is created when a task returns a `SharedDFut` to its parent task, or passes a `SharedDFut` to a child task. In both cases, the process executing the task adds the ID of the worker that executes the parent or child task to its local borrower set.

In many cases, a child task will finish borrowing the `DFut` by the time it has finished execution. Concretely, this means that the worker executing the child task will no longer have a local reference to the `DFut`, nor will it have any pending dependent tasks. Thus, when the worker returns the task's result to its owner, the owner can remove the worker from its local set of borrowers, with no additional messages needed. This optimization is important for distributing load imposed by reference counting among the borrowers, rather than requiring all reference holders to be tracked by the owner.

However, in some cases, the worker may borrow the `DFut` past the duration of the child task. There are two cases: (1) the worker passed the `DFut` as an argument to a task that is still pending execution, or (2) the worker is an actor and stored the `DFut` in its local state. In these cases, the worker notifies the owner that it is still borrowing the `DFut` when replying with the task's return value.

Eventually, the owner must collect all of the borrowers in its local set. It does this by sending a request to each borrower to reply once the borrower's reference count has gone to zero. Borrowers themselves never delete from their local set of borrowers. Once a borrower no longer has a reference or any pending dependent tasks, it replies to the owner with its accumulated local borrower set. The owner then removes the borrower, merges the received borrowers into its local set and repeats the same process with any new borrowers. If a borrower dies before it can be removed, the owner removes it upon being notified of the borrower's death.

When a `DFut` is *returned* by a task, it results in a nested `DFut`. Nested `DFuts` can be automatically flattened, e.g., when submitting a dependent task, but we must still account for nesting during reference counting. We do this by keeping a set of `DFuts` whose values contain the `DFut` in question in the ownership table (Table 2). The `DFut`'s value is pinned if its nested set is non-empty.

B Formal Specification

We developed a formal specification for the ownership-based system architecture [53]. It models the system state transitions of the ownership table for task scheduling, garbage collection, and worker failures. The goal is to check the correctness of the system design, which is manifested in the following properties:

- Safety: A future's lineage information is preserved as long as a task exists that depends on the value of the future. This is defined recursively: at any time, either the value of a future is stored inline (thus cannot be lost), or all futures that this future depends on for computing its value must be safe. Formally, it means the following invariant holds at any given time: $\forall x$,

$$\begin{aligned}&\text{LineageInScope}(x) \triangleq\\ &\quad \lor x = \texttt{INLINE\_VALUE}\\ &\quad \lor \forall arg \in x.args : \text{LineageInScope}(arg)\end{aligned}$$

- Liveness: The system will eventually execute all tasks and resolve all future values, even in case of failures, i.e., all `Get` calls eventually return.

- No Resource Leakage: The system will eventually clean up all task states and future values, after the all references to futures become out-of-scope.

We checked the model using the TLA$^+$Model Checker [55] for up to 3 levels of recursive remote function calls, where each function creates up to 3 futures, and verified that the safety and liveness properties hold in more than 44 million distinct states. Currently, the model does not include first-class futures or actors; we plan to include these and open-source the full TLA$^+$specification in the future.

Fault-Tolerant Replication with Pull-Based Consensus in MongoDB

Siyuan Zhou[1], Shuai Mu[2]
[1]*MongoDB Inc.* [2]*Stony Brook University*

Abstract

In this paper, we present the design and implementation of strongly consistent replication in MongoDB. MongoDB provides linearizability and tolerates any minority of failures through a novel consensus protocol that derives from Raft. A major difference between our protocol and vanilla Raft is that MongoDB deploys a unique *pull-based* data synchronization model: a replica pulls new data from another replica. This pull-based data synchronization in MongoDB can be initiated by any replica and can happen between any two replicas, as opposed to vanilla Raft, where new data can only be pushed from the primary to other replicas. This flexible data transmission topology enabled by the pull-based model is strongly desired by our users since it has an edge on performance and monetary cost. This paper describes how this consensus protocol works, how MongoDB integrates it with the rest of the replication system, and the extensions of the replication protocol that support our rich feature set. Our evaluation shows that MongoDB effectively achieved the design goals and can replicate data efficiently and reliably.

1 Introduction

MongoDB is a general purpose, document-based, distributed database. In the last few years, we have been focusing on improving its support for replication, as there has been an increasing demand for stronger fault tolerance. In previous papers we discussed how MongoDB supports tunable consistency [29] and how causal consistency works [34]. In this paper we present the details of how MongoDB provides linearizable [12] replication with fault tolerance.

A common approach to fault-tolerant linearizable replication is through consensus protocols [24, 4]. After studying the popular consensus protocols including Paxos [17] and Raft [26], we concluded that no existing consensus protocols directly fit our needs without heavy modifications. The key reason is that these existing protocols are *push-based*: there is usually a primary server and the primary will push new data to all replicas. Yet in MongoDB we aim for a *pull-based* synchronization model: a replica fetches new data proactively from another replica, and not necessarily from the primary.

There are a few reasons for why we target the pull-based synchronization model. First, allowing data synchronization to happen between any two replicas enables a more flexible data transmission topology, that could utilize networks in more optimal ways. Many of our users prefer being able to configure how their network is utilized, especially for those who deploy their systems across different datacenters. Second, using a pull-based data synchronization model gives us backward-compatibility as we have previously implemented a preliminary version of a pull-based primary-backup replication scheme that is not backed by any consensus protocols, and thus has limited fault tolerance.

As there is no direct fit, we developed a new replication (consensus) scheme based on the Raft protocol. We chose Raft as the base because it is more accepted for industry use, easy to understand, and similar to our previous replication protocol, but we believe other bases such as Paxos should work too. The principle of our approach is to decouple data synchronization in Raft (mostly the AppendEntries RPC) into two parts: replicas pulling new data from the peers, and replicas reporting their latest replication status so that a request can commit after it reaches a majority of replicas.

The development of the new replication scheme is, however, easier said than done. The main challenge is in the subtlety of the Raft (and any other) consensus protocol. During our development we found that any unthoughtful changes to the protocol would easily introduce new corner cases that would break the correctness of the system. To verify that our design and implementation are correct, we have done extensive verification and testing on the protocol including model checking using TLA+, unit testing, integration testing, fuzz testing [11] and fault-injection testing.

Our developed protocol achieves the goal of allowing data pulling between *any* two replicas. Unlike Raft which can only push data from the primary to other replicas (broadcast), our system supports arbitrary data synchronization paths: from linear chaining to broadcast. This gives MongoDB several advantages over using vanilla Raft, including both performance-wise (e.g., saving leader bandwidth) and management-wise (e.g., controlling data transmission paths).

The main contributions of this paper include:

- We design a new consensus protocol based on Raft. Our protocol better meets the needs of MongoDB. It enables more flexible and customizable data synchronization paths during replication.
- We describe the design choices in how MongoDB adopts

this consensus protocol. MongoDB offers a unique feature set, which brings extra challenges in designing and implementing its replication. For example, we provide speculative execution to be compatible with MongoDB's weak consistency features, but also guarantee linearizable log replication with rollbacks.

- We report the evaluation results of MongoDB for different replication parameters. Our evaluation shows that MongoDB replication is reliable and efficient.

We will organize the rest of paper as follows. Section 2 gives an overview of the background. Section 3 describes the main body of the consensus protocol. Section 4 gives a few important extensions to our design. Section 5 discusses the evaluation results. Section 6 discusses related works before we conclude.

2 Background

MongoDB interfaces and architecture. MongoDB is a database that stores data as *documents* and supports general CRUD operations on one or many documents with a rich query language. Each document is a binary JSON-like (called BSON) object. Documents are identified by unique ids and grouped in *collections*, which are similar to tables in a SQL database.

To provide high availability, MongoDB provides the ability to run a database as a *replica set*, which is a set of MongoDB nodes that act as a consensus group, where each node maintains a logical copy of the database state. MongoDB also supports *sharding* for horizontal scaling, which distributes all data in a collection onto different shards in a share-nothing manner. Each shard is deployed as a replica set. In this paper we focus on a single replica set, as sharding is orthogonal.

Consistency and fault tolerance. Previous papers have described how MongoDB can achieve weaker consistency levels, including causal consistency [34, 29]. In this paper, we focus on the strongest consistency level—linearizability [12]. We assume a (partially) asynchronous environment where messages can be arbitrarily delayed and there are no perfect failure detectors. For each replica set, at most a minority of servers can fail in order to maintain availability. The problem of fault-tolerant linearizable replication is commonly solved by consensus protocols. Examples include Viewstamped Replication [24], Paxos [17], Zab [13], and Raft [26]. Our solution started with adopting the recent and popular Raft, but ended up basically inventing a new protocol with many heavy modifications.

Evolution of MongoDB's pull-based replication. Starting from MongoDB version 1.0 over a decade ago, MongoDB supported replication with a primary-backup scheme. Unlike conventional primary-backup replication schemes where updates are usually *pushed* from where they are firstly received—the primary—to the secondaries (backups), we chose a design in which a secondary server can constantly *pull* updates from other servers, and not necessarily from the primary.

A major benefit of the pull-based approach is that it enables a more flexible control of how data is transmitted over the network. Depending on users' needs, the data transmission can be in a star topology, a chaining topology, or a hybrid one. The ability of controlling data transmission paths is a strong customer need, mostly for reasons related to performance and monetary cost. For example, when deployed in clouds like Amazon EC2, data transmission inside a datacenter is free and fast, but is expensive and subject to limited bandwidth across datacenters.

The pull-based approach has led our designs while we continually evolved our replication protocols in the last few releases. In earlier releases several years ago, we assumed a semi-synchronous network: either there is manual control of failover (the user needs to appoint a node as the new primary when the old one fails), or all messages are bounded to arrive within 30 seconds for failure detection. Starting from 2015, we remodeled our replication scheme based on the Raft protocol. This new protocol guarantees safety in an asynchronous network (i.e., messages can be arbitrarily delayed or lost) and supports fully autonomous failure recovery with a smaller failover time. Same as before, this new protocol is still pull-based. We will describe how it works in the next section.

3 Design

This section describes how replication in MongoDB works, including the overall architecture and data structures (§3.1), the main body of the replication protocol (§3.2), a discussion of correctness, (§3.3), and how the system chooses data transmission paths (§3.4). Attached to this paper is an appendix that summarizes the difference between the Raft protocol and MongoDB's consensus protocol.

3.1 Preliminaries

In MongoDB, the object for replication is called the *oplog*. An oplog is a sequence of log entries; each log entry contains a database operation. Figure 1 shows an example of oplog entry. The oplog is stored in the *oplog collection*, which behaves in almost all regards as an ordinary collection of documents. The oplog collection automatically deletes its oldest documents when they are no longer needed and appends new entries at the other end.

An oplog entry needs to be replicated to at least a majority of servers to *commit*: a committed entry persists through any minority failures. The system will wait for the oplog entry to commit before it acknowledges the client. MongoDB also

```
{
  // The oplog entry timestamp
  "ts": Timestamp(1597904287, 12),
  // The term of this entry
  "t":  NumberLong(40),
  // The operation type, "i" for insert
  "op": "i",
  // The collection name
  "ns": "test.collection",
  // A unique collection identifier
  "ui": UUID("947b54f...852f62")),
  // The document to insert
  "o":{
      "_id": ObjectId("5f3e...b950"),
      "x": 1
  }
}
```

Figure 1: Example of key oplog entry fields for an "insert" operation

supports operations with weaker consistency, in which case the system can acknowledge clients before the oplog entry commits [29, 34]. After replication, all servers of the same replica set will have identical oplogs. Oplog entries will be applied in the same order on all servers.

A server can act as either a primary or a secondary. [1] Only a primary can process write requests. A server has a third role as a *candidate* when it is transitioning from a secondary to a primary through *elections*. MongoDB's election rules are the same as Raft's. When a node decides to start an election, for example, because it has not seen a primary for an election timeout, the node transitions to a candidate role, increases its term, and sends vote requests to others. A voter can grant its vote to only one candidate in a given term and only if the candidate has the same or a more up-to-date log than the voter. The candidate wins the election if it is able to collect votes from a majority of nodes including the candidate itself; it then becomes a primary.

After the election, the new primary will have a unique, monotonically increasing *term* number. When the primary generates a new oplog entry, it will append this entry into its own oplog, and replicate the entries through the *data replication* protocol described in Section 3.2. It could happen that more than one server is acting as a primary, but the data replication and the election protocols collectively guarantee that at most one primary can successfully commit log entries at a particular index.

In MongoDB, each oplog entry is assigned a timestamp and annotated with the term of the primary. The timestamp is a monotonically increasing logical clock that exists in the system before this work. It is used to index the oplog entries, similar to the log index in Raft. A pair of term and timestamp, referred to as an OpTime, can identify an oplog entry uniquely in a replica set and give a total order of all oplog entries among all replicas. OpTimes are compared lexicographically, i.e., an OpTime is greater than another if its term is higher or the terms are the same but its timestamp is higher.

[1]Our choice of terminology is for historical reasons. They are the same as a *leader* and a *follower* used in Raft.

3.2 Data Replication

As mentioned in previous sections, MongoDB uses a pull-based replication scheme. Unlike Raft and other common consensus protocols that would initiate RPCs from the primary to secondaries when the primary tries to replicate new log entries (for example, in Raft, this is the AppendEntries RPC), in MongoDB, the primary waits for the secondaries to pull the new entries that are to be replicated.

After appending an entry to the oplog, the primary can process two types of RPCs from secondaries: PullEntries and UpdatePosition. A secondary will use PullEntries to fetch new logs, and use UpdatePosition to report its status so that the primary can determine which oplog entries have been safely replicated to a majority of servers and commit them. Similar to Raft, once an entry is committed, all prior entries are committed indirectly.

3.2.1 PullEntries

Note that a key design choice is that a secondary does not have to send PullEntries only to the primary. Instead, the secondary can pull new entries from any (nearby) servers. This secondary is called the *syncing server*, while the upstream server that receives the PullEntries RPC is called the *sync source*.

A secondary continuously sends PullEntries to the selected sync source (see more details about the sync source selection in §3.4) to retrieve new log entries when they become available. The PullEntries RPC includes the latest oplog timestamp (prevLogTimestamp) of the syncing server as an argument. When receiving PullEntries, a server will reply with its oplog entries after and including that timestamp if it has a longer or the same log, or the server could reply with an empty array if its log sequence is shorter. Before returning a response when the log is the same, PullEntries waits for new data for a given timeout (5 seconds by default) to avoid busy looping.

When the syncing server receives the reply of PullEntries, it will try to merge the log entries in the reply into its own oplog. Before merging, it checks if the incoming log entries concatenate with the local oplog. In particular, it checks if the first received entry has the same OpTime as the last local oplog entry. Only if so, the syncing server continues to merge by appending the received entries to the oplog. If the received oplog entries don't overlap with the local ones and the received entries are "newer" by comparing their last OpTimes, the syncing server will traverse the oplog on its sync source in order to find their last common entry, then discard any diverged oplog entries since then. Afterwards, the syncing server should be able to pull new data and append it to the local oplog. Discarding diverged logs will require more work than ordinary Raft implementations because of our optimizations on speculative execution (see details in §4.1).

3.2.2 UpdatePosition

After retrieving new entries into its local oplog with PullEntries, the secondary will send UpdatePosition to its sync source, reporting the latest log entry's OpTime. When receiving the UpdatePosition, the server will forward the message to its sync source, and so forth, until the UpdatePosition reaches the primary. The primary keeps a non-persistent map in memory that records the latest known log entry's OpTime on every replica, including its own, as their log positions. When receiving a new UpdatePosition, the primary will compare the received OpTime with its local record. If the received one is newer, the primary will replace its local record with the received one. Afterwards, the primary will do a count on the log positions of all replicas: if a majority of replicas have the same term and the same or greater timestamp, the primary will update its lastCommitted to that OpTime and notify secondaries of the new lastCommitted by piggybacking onto other messages, such as heartbeats and the responses to PullEntries. lastCommitted is also referred to as the *commit point*.

3.2.3 Implementation

In MongoDB, instead of initiating continuous RPC's on the syncing node, the PullEntries RPC is implemented as a query on the oplog collection with a "greater than or equal to" filter on the timestamp field. The query can be optimized easily since the oplog is naturally ordered by timestamp. Using database cursors allows the syncing node to fetch oplog entries in batches and also allows the RPC to work in a streaming manner, so that a sync source can send new data without waiting for a new request, reducing the latency of replication.

To avoid a flood of forwarded UpdatePosition messages, a server passively batches the received UpdatePosition requests between two rounds of forwarding and consolidates the requests by only keeping the highest log position for each server. A server only maintains at most one in-progress UpdatePosition request to its sync source, so it waits to send the next request until the previous one returns.

We also introduced Heartbeats RPC, which decoupled the heartbeat responsibility from Raft's AppendEntries RPC. Heartbeats are sent among all replicas, used for liveness monitoring, commit point propagation and sync source selection (§3.4).

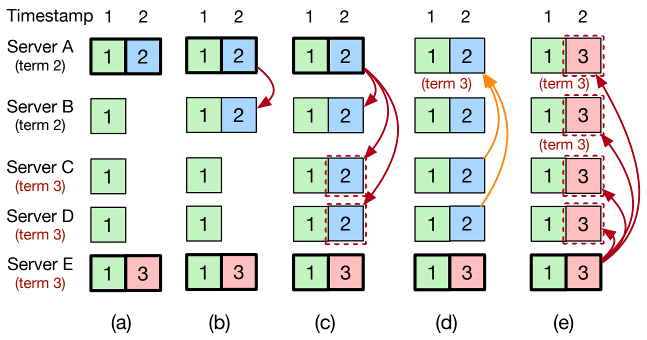


Figure 2: A corner case
Each box represents an oplog entry with its term (shown in different colors) in the box. (a) start state with both A and E being primaries (thick border); (b) Raft only allows replicating (red arrows) from A to B; (c) MongoDB can replicate from A to B/C/D; (d) C/D report log positions (orange arrows) with term 3, forcing A to step down. (e) data replicated in (c) may be rolled back.

3.3 Correctness

Careful readers may have noticed that our data replication protocol (how PullEntries are processed) only checks the OpTimes in the logs; it does not check if the sync source has a higher or equal term than the syncing server. This is different from what Raft would do in data replication. The data replication in Raft is done via AppendEntries RPC, which contains the term of the primary. AppendEntries can only succeed if the primary has a term that is not lower than the secondary's.

This crucial protocol change means that log replication in MongoDB will behave differently from Raft. In Raft, if a server has voted for a higher term in an election, the server cannot take new log entries sent from an old primary with a lower term. But in our system, because the PullEntries RPC does not check the term of the sync source, even if the sync source is a stale primary, it is possible that after a server has voted for a higher term, the server could still fetch new log entries generated by the stale primary.

Figure 2 shows an example of this disparity occurring on secondaries. Initially, all five servers acknowledged entries in term 1. Server A first wins the election in term 2 with votes from Server A/B/C and writes down one entry locally. Server E then wins the election in term 3 with votes from Server C/D/E and writes a different entry locally. In Raft, if Server A broadcasts its AppendEntries, only Server B will accept the new entry from Server A; Server C/D/E will all reject.

In MongoDB, however, Server B/C/D could all take the new log from Server A even after Server C/D have voted for the new primary in term 3. If A is the sync source of Server B/C/D, then B/C/D will still accept the new entry with term 2 because the entry is newer than their local ones. Now that the entry in term 2 has been replicated to the majority of servers, it would be considered by Server A as committed if we did not make further changes to Raft's rule that *a log entry is committed once the leader that created the entry has replicated it on a majority of the servers* [26]. Later, Server E's new entry with term 3 could propagate to all servers and overwrite the committed entry. Without more changes from Raft, this would violate safety. [2]

To prevent cases like this from happening, we added a new argument in the UpdatePosition RPC: the term of the syncing server. The recipient of UpdatePosition will update its local term if the received term is higher. If the recipient is the stale primary, seeing a higher term will make the primary step down before committing anything, thus avoiding any safety issue. In the above example, when server A receives UpdatePosition from Server C/D, it will see term 3 and step down immediately without updating its lastCommitted. Even though the entry with term 2 is in a majority of servers' logs, it is not committed.

Until now, we have assumed it is a secondary that fetches oplog entries from a stale primary after voting for a new primary. In fact, this syncing server could be the new primary itself. Even if a primary (or candidate) has voted for itself in a higher term, it could still fetch data generated in lower terms from other replicas, as long as it has not generated new oplog entries with the new term and appended the entries to its own oplog. This important difference between MongoDB and Raft allows MongoDB to preserve uncommitted data as much as possible during failovers (see more in §4.3).

More formally, the revisions we made to UpdatePosition are to maintain a key invariant in Raft—Leader Completeness Property. This property refers to the fact that "if a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms." (See Figure 3 in the Raft paper.) In MongoDB, in order to commit an entry in term T, the primary in term T has to receive UpdatePosition RPCs with term T from a majority of nodes. A later new primary in term U > T must collect votes from a majority of nodes too. The two majorities must overlap on at least one voter. This voter is the key to guarantee the safety. Either the voter sent UpdatePosition in term T to commit the entry **before** voting, thus implying the new primary had the committed entry due to the Log Matching Property; or the voter voted first and sent UpdatePosition with a term higher than T, thus leading the primary in term T to step down without committing the entry. Either way, the Leader Completeness Property is guaranteed.

In addition to this property, other invariants of Raft still hold so that one can prove the correctness of our protocol following the proof of Raft. Further, to mechanically verify the correctness of our system, we have written a formal specification of the protocol in TLA+ and applied model checking to it [33].

[2]Another angle to look at this problem is that, allowing UpdatePosition from a server with a higher term is actually "counting replicas" rather than "counting replies". Raft has explained well why counting replicas is incorrect in its paper. Indeed, Raft's TLA+ spec in [25] specifies a stronger definition of "commit" than its paper. The definition in TLA+ is similar to our modification: an entry $\langle index, term \rangle$ is **immediately committed** if it is acknowledged by a quorum (including the leader) during *term*.

3.4 Sync Source Selection

Servers learn about the status of other servers, including their log positions, via Heartbeat RPC. A server chooses its sync source only if the sync source has newer oplog entries than itself by comparing their log positions. This condition is double-checked on receiving PullEntries RPC's responses in case rollback (§4.1) occurs on the sync source. As a result, it's guaranteed that the replicas can never form a cycle of sync sources. Once a server starts to pull entries from its sync source, it keeps fetching from the source until the source is not available or a better source shows up. Thus a server should not change its sync source frequently in a stable environment.

4 Extensions

In this section we introduce a few key features of MongoDB that extend the elementary design.

4.1 Speculative Execution and Rollback

The standard approach of applying log entries in Raft-based systems is that a replica waits until the log entries are committed and then applies the log entries in timestamp order. MongoDB introduces an optimization that speculatively applies an oplog entry when it is added to the oplog. If a failover happens, speculatively applied oplog entries could be deleted (§3.2.1). In this case, the system needs to roll back the operations in these entries. The common approach for rollbacks in databases is through undo or redo logs. The way MongoDB achieves this is through a consolidated design of the storage engine (named WiredTiger) and the replication protocol.

The WiredTiger storage engine is a multi-version transactional storage engine that can use oplog timestamps as versions of data updates. There are three key functions implemented in the storage engine that enable this consolidation. First, the storage engine supports speculative updates, so multiple versions are visible to clients depending on their requested consistency levels even if not all of them are committed. Second, the storage engine provides fast rollback to a timestamp and discards all updates after that timestamp. Third, when oplog entries are committed, the storage engine can be notified to merge all data updates with lower timestamps in the on-disk checkpoint and garbage collect those versions.

When a node needs to roll back, it will determine the newest oplog entry it has in common with its sync source. The timestamp of this oplog entry is referred to as t_{common}. The node needs to truncate all oplog entries with a timestamp after t_{common}. In addition to oplog truncation, it must undo the speculative effects of the operations deleted from the oplog.

Since MongoDB version 4.0, the WiredTiger storage engine has provided the ability to revert the replicated data to the version at a given timestamp. MongoDB periodically informs the storage engine of a *stable timestamp* (t_{stable}), which is the timestamp of the commit point known by this node and must be less than or equal to t_{common} when the node starts rollback.

To undo the effects of truncated oplog entries, the rolling back node reverts its replicated data to the version at t_{stable}, then applies the oplog entries forward from t_{stable} up to and including t_{common}.

4.2 Initial Sync

MongoDB discards stale oplog entries once the storage space used to store the entries reaches a configurable threshold, by default 5% of free disk space at startup. In most consensus systems, such as Chubby and the vanilla Raft, a *snapshot* of the database is obtained before discarding stale log entries. The snapshot will be used by a new server to catch up when joining the system. However, MongoDB does not rely on a snapshot mechanism for this initial synchronization, referred to as *initial sync*.

The major reason for MongoDB not using snapshots for initial sync is that MongoDB has a pluggable storage API that does not require the storage engine to support snapshots. For example, the initial storage engine before WiredTiger at its core used `mmap` [3], which does not support snapshots. This `mmap`-based storage engine needs to be supported due to backward-compatibility.

The initial sync in MongoDB works as follows. First, when a new server is joining, it chooses a sync source and uses this sync source for the whole duration of initial sync. Once the initial sync starts, the syncing node records the current applied oplog timestamp on the sync source as the initial sync start point. Then, the new server starts to clone the database of the sync source by scanning the database. The database scan gets a cursor at the beginning of each collection and iterates over the cursor to the end. Note that this clone may be inconsistent when there are concurrent updates happening in the system—some cloned values are up-to-date, some may be obsolete, and some may be missing. The final step is to fix this inconsistency. The new server will retrieve all oplog entries on the sync source starting from the initial sync starting point, and apply the oplog entries locally on the database. After the database clone, the new server records the current applied oplog timestamp on the sync source again as the initial sync end point. Once the new server applies oplog entries beyond this point, the data becomes consistent and the initial sync is complete. If the sync source fails during the initial sync, the new server chooses a different sync source and restart the process.

Note that some oplog entries may be applied twice in the database on the new server. If an operation gets applied on the sync source after the initial sync begins, the operation's effect may be included in the database that the new server cloned gradually. Later this oplog entry will be applied again on the new server. To avoid any data inconsistency caused by this effect, MongoDB requires any sequences of operations in the oplog entries to be *idempotent*: applying the same sequence of operations multiple times will lead to the same system state.

For operations that change the database states (inserts, updates, deletes, etc.), we changed their semantics during initial sync to make them idempotent. Inserts in MongoDB will be ignored if the document id already exists; updates and deletes will be ignored if the modified document's id does not exist. MongoDB supports rich update operations, such as incrementing a field's value in a document. These operations will be converted to unconditional field assignments by the primary. For example, for an increment request, the primary will read the current value from its local database and compute the result of the increment, and replicate an oplog entry setting the field to the computed result. As a result, the consistency between the new server and its sync source is guaranteed.

4.3 Preserving Uncommitted Oplog Entries

After a failover, the uncommitted oplog entries on the previous primary are likely lost. This would be fine for a different system because the clients could retry uncommitted updates. This is, however, an issue for MongoDB because MongoDB supports fast but weak consistency levels that acknowledge writes as soon as they are applied on the primary or just replicated to a fewer number of nodes than a majority. Thus, a failover could cause a large loss of uncommitted writes. Though the clients are not promised durability with weak consistency levels, we still prefer to preserve their uncommitted writes as much as possible.

For this purpose, we introduced an extra phase for a newly elected primary—the primary catchup phase. The new primary will not accept new writes immediately after winning an election. Instead, it will keep retrieving oplog entries from its sync source until it does not see any newer entries, or a timeout occurs. This timeout is configurable in case users prefer faster failovers to preserving uncommitted oplog entries.

As explained in §3.3, the primary catchup design is only possible because a primary is allowed to keep syncing oplog entries generated by the old primary after voting for a higher term as long as it hasn't written any entry with its new term.

[3]When using this deprecated `mmap` engine, features including speculative execution and rollback are implemented in different approaches and are omitted due to space limitation.

4.4 Additional Replica Roles

4.4.1 Arbiters

MongoDB supports a special replica role called *arbiter*. An arbiter is like a secondary with respect to voting but does not store any data to save the cost of storage and replication while being a tie-breaker in elections. For example, in a Primary-Secondary-Arbiter deployment, when the primary crashes, the secondary can take over with the vote from the arbiter to serve reads and writes. The writes will be applied speculatively but cannot be committed. [4] If a new node needs to be added to the replica set to replace the crashed one through initial sync, the arbiter allows a safe reconfiguration to change the membership.

4.4.2 Non-Voting Members

In addition to fault tolerance requirements, users often deploy replica sets to offload reads from the primary or to access a local data copy with lower latency. For example, users may maintain some replicas for a heavy analytical workload using MongoDB's expressive aggregation framework. These replicas are not deployed for fault tolerance and may have unstable write performance due to the heavy analytical workload. As another example, a geo-distributed application may prefer to read from a nearby datacenter for lower latency, thus need to deploy dozens of replicas globally. Assuming it's extremely rare for more than a few servers to fail at the same time, it would be unnecessary to count all replicas towards a quorum for fault tolerance and undesired to wait for a majority of such many servers to replicate in order to commit writes.

MongoDB introduced *Non-Voting Members* particularly for this purpose. Non-voting members replicate data as normal secondaries, but they do not participate in elections or count towards a quorum for committing oplog entries, as opposite to *Voting Members*. MongoDB supports up to 50 replicas but only up to 7 voting members. Therefore, writes with strong consistency levels can return faster, as long as they are committed after replicating to a quorum of voting members rather than a quorum of all replicas.

Non-voting members work well with the pull-based data replication model since it's possible to minimize their performance impact on the primary and voting members by offloading their significant oplog read workload to other non-voting members as much as possible.

4.5 Election Optimizations

4.5.1 Election Handoff

On failovers, secondaries wait for an election timeout (10 seconds by default) to run for election in order to detect that the primary is no longer available. However, on planned failovers, it is known that the old primary has already stepped down. MongoDB introduced *Election Handoff* to shorten the planned failover time by avoiding the next candidate's waiting.

When a primary server steps down on administrator commands, it will pause new writes and wait for any eligible secondary to catch up its oplog within a user specified timeout. The primary then chooses this caught-up secondary to immediately run for election on a best-effort basis. In common cases, the chosen secondary will win the election and become the new primary. This election handoff mechanism will likely shorten the failover time as it does not need the the election timeout to elapse. This is similar to the leadership transfer extension in Raft.

The election handoff is leveraged by the planned maintenance on MongoDB Atlas[21], MongoDB's hosted database as a service. Atlas uses a rolling upgrade strategy for executing maintenance or infrastructure operations, such as applying security patches, scaling up an Atlas cluster, and upgrading to the latest MongoDB minor versions. As part of the rolling upgrade, the primary will be stepped down and shut down for upgrade. The election handoff minimizes the unavailability window on planned failovers. In fact, the vast majority of failovers (89.03%) on Atlas are caused by planned maintenance (§5.2) and can benefit from the election handoff.

4.5.2 Member Priority

It is common that users have a preference for which server should act as primary, especially in a multi-datacenter setup where users prefer to deploy the primary in the datacenters closest to the applications (clients) for lower latency. MongoDB supports setting election priority among servers in the replica set configuration. If a secondary realizes it has a higher priority than the current primary, it will start an election after a timeout based on its relative priority. The higher the priority it has, the smaller the timeout value will be. In this way, the server with the highest priority is likely to win the election first. If the election fails due to competition but the server still has a higher priority than the new primary, it will continue calling for elections until it becomes primary or the new primary has a higher priority. Setting a server's priority to zero prevents it from running for election.

One potential problem is that when the election caused by a high-priority server fails, it will propagate its larger term number to the rest of the replica set and force the current primary to step down. This disruption is particularly serious when this high-priority server is lagged due to shutdown and rejoins the replica set after a restart. Until this high-priority server is caught up on its oplog, it keeps running elections periodically and causing disruptions. Although other servers will elect a new primary after each disruption, the system will be unavailable for at least an election timeout on every

[4] This speculative execution can benefit reads with weaker consistency levels described in [29].

disruption. To prevent disruptions when a server rejoins the replica set, Raft describes a pre-vote algorithm in Section §9.6 in [25], where a candidate only increments its term and runs the real election if it learns from a majority of nodes that they would grant their votes. MongoDB implements this algorithm with an election dry-run. When a stale candidate with a higher priority starts an election even though it won't be able to win, the candidate will fail during the election dry-run, protecting the existing primary from being disrupted by a higher term.

Note that in rare cases such as network partitions, it could happen that two nodes repeatedly initiate elections and cause liveness issues. This is a different issue from the priority design and having priorities does not make it worse. In these cases, the dry-run mechanism cannot address the issue completely, as liveness in asynchronous networks is impossible to guarantee [9]. In reality we did not find this to be a problem.

4.6 Read-only Operations

A strawman solution to support linearizable reads is to turn read operations into log entries similarly to treating write operations. MongoDB's optimized approach is that if the primary has other concurrent oplog entries to replicate, the primary can piggyback the read linearization point with those entries. Note that this optimization is different from weakly consistent reads although they both skip turning read operations into oplog entries. Even though weakly consistent reads are supported on both primary and secondary nodes and they do not need to wait for synchronization between nodes, linearizable reads can only happen on the primary and need to wait for a roundtrip of synchronization.

5 Evaluation

Our evaluation section has two parts. First, we benchmarked the system under different configurations on Amazon EC2 and report the performance measurements (§5.1). Second, we collected metrics from our own cloud platform, MongoDB Atlas [21], and report the analysis of the operational data on failovers (§5.2).

5.1 Benchmarks on EC2

5.1.1 Setup

Our tests use AWS m5d.2xlarge instances, each with 8 vCPUs, 32GB memory, and a local SSD. We tested 5-way replication with 5 server VMs and 2 client VMs. 3 servers are deployed in the US East (N. Virginia) region and 2 servers in the US West (Oregon) region. The primary is always deployed in the US East region.

We use the following benchmark for our main tests. Each client thread continuously and randomly updates an entire document out of 1 million documents containing one random string field of 1000 bytes in a closed loop. These updates will not return until they are committed. We vary the number of client threads to control the offered load in the system.

To measure the impact of allowing secondaries to sync from other secondaries, we run the tests with two settings: (1) chaining enabled and a secondary in the US West region forced to sync from another secondary in the same region while all other secondaries syncing from the primary, and (2) chaining disabled and all secondaries syncing from the primary, which mimics similar data transmission paths to vanilla Raft.

In addition to the above basic benchmarks, we also tested two interesting cases: (1) a failure recovery test in which we crash the primary, wait for the new primary to step up, and then recover the old primary; (2) a comparison with a previous version of MongoDB that uses a deprecated replication protocol not based on known consensus protocols.

Appendix B has an additional TPCC benchmark.

5.1.2 Benchmark Results

Figure 3 and 4 respectively show the 50-percentile and 90-percentile latency vs. throughput of both chaining enabled and disabled cases. Their performance is almost the same. However, as shown in Figure 5, the cross-datacenter traffic is halved by leveraging chaining, so is its cost. Using \$0.01~\$0.147/GB (EC2's pricing) to estimate, the savings of chaining in this test is about \$300~\$5,000 per month.

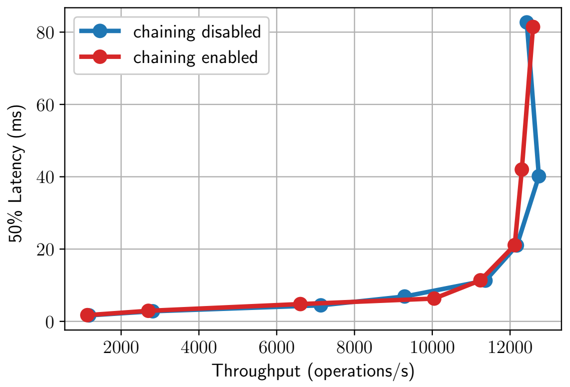


Figure 3: 50-percentile latency vs. throughput of chaining disabled and enabled

One may expect that with chaining enabled the system should be able to achieve a higher maximum throughput because the primary would have more available CPU resources for clients. The system is indeed bottlenecked by the primary's CPU at the maximum throughput in our tests and most of the CPU is used to serve clients. Each client request is handled by a separate OS thread, so there are a few thousand

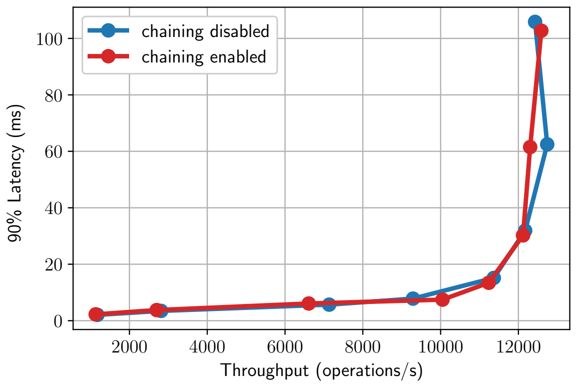


Figure 4: 90-percentile latency vs. throughput of chaining disabled and enabled

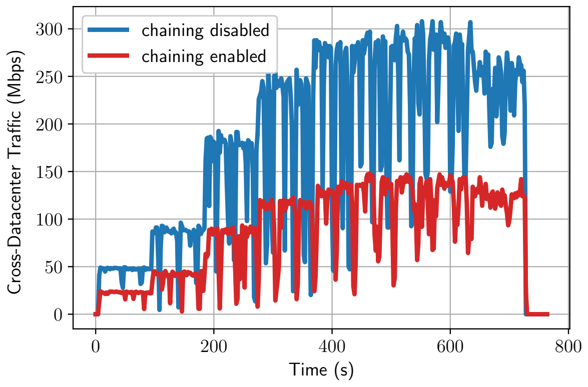


Figure 5: Cross datacenter traffic with chaining disabled and enabled

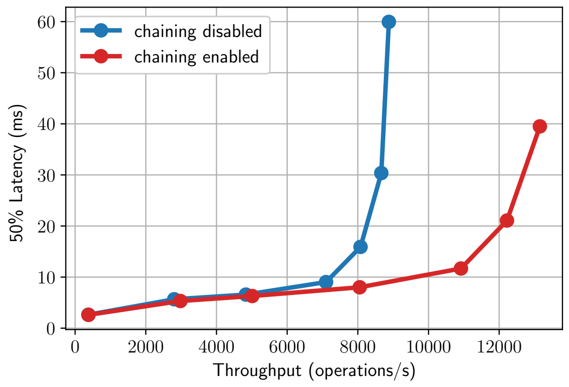


Figure 6: 50-percentile latency vs. throughput of chaining disabled and enabled with limited 200Mbps bandwidth on primary

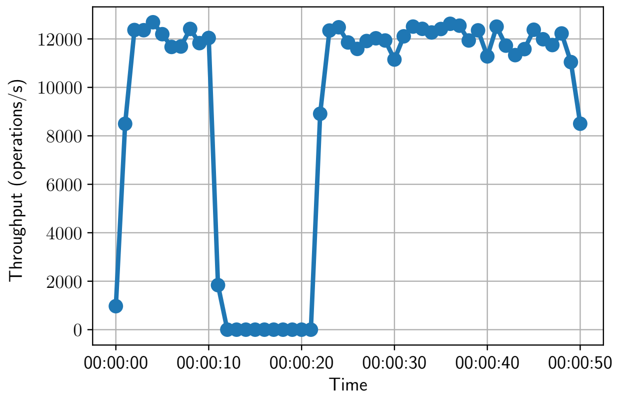


Figure 7: Throughput during failover

client threads in our tests. We observed more than 250k context switches per second on the primary consistently when the system is saturated. The heavy interleaving of threads indicates that the performance will not scale linearly given extra available CPU time. Meanwhile, each secondary oplog read takes one thread on the primary. In fact, a secondary in a steady replication state only consumes about 5% of a single CPU core on the primary in our experiments. Therefore we did not observe throughput improvement with chaining enabled when the primary CPU is the bottleneck.

Nevertheless, if the network bandwidth between nodes is the bottleneck, we expect that enabling chaining will greatly improve the throughput. In our tests, we did not observe throttled bandwidth across datacenters on AWS EC2. But it is reported that the bandwidth of the cloud could be affected by time, space, and other factors such as VM instance types [16]. Besides, our users could deploy MongoDB in other environments with networks that may be less reliable and have less bandwidth than EC2. Therefore, we conducted an additional experiment with 3 nodes, where the primary's bandwidth is limited using the `tc` tool. The result is in Figure 6) and it shows that when the network bandwidth is the bottleneck, chaining can drastically improve the system throughput as expected.

5.1.3 Failover Tests

The failover tests are conducted with all 5 replicas deployed in the same datacenter. In each test, we run 64 clients concurrently and crash the primary after the system runs for 10 seconds in a stable state. The timeout set for the system to elect a new primary is 10 seconds. After the new primary takes over and the system is again in a stable state, we recover the old primary. The old primary will catch up by synchronizing the missing oplog entries.

Figure 7 shows the throughput of the system during failover when chaining is enabled. The new primary steps up after losing the old primary over an election timeout. The clients can discover the new primary and issue writes to it immediately, thus the throughput resumes to the level before the failover. When the old primary is recovering (at 40s), it fetches the missing oplog entries from another secondary in this case. In our tests not shown here with chaining disabled, the old primary catching up its oplog from the new primary increases the workload on the new primary but doesn't affect the performance since the new primary isn't saturated.

5.1.4 Comparing with Previous Implementation

We compare the latest released version 4.4 with a previous MongoDB version 3.6 released in 2017. Version 3.6 is the last version that supports the old and deprecated replication protocol, which works in most cases but is unproven. Additionally, it has severe limitations due to its strong assumptions about the deployment environment: all messages must be replied within 30 seconds or otherwise the nodes must have failed. Thus, it does not tolerate faults like network partitions and could suffer from a "split-brain" if such faults happen.

The main advantage of our current protocol is fault tolerance as it makes fewer assumptions of the deployment. In most cases, we observed comparable performances, as shown in Figure 8. The performance of the newer version is better when the system isn't saturated. We believe this is not only because of the algorithmic and engineering improvements of the replication system, but also thanks to optimizations of many other parts of the server, e.g., journaling.

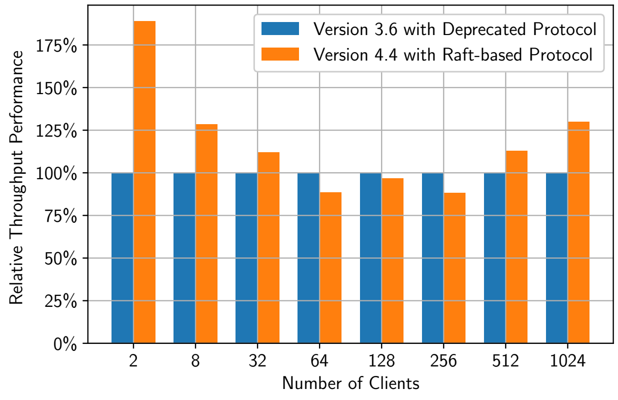


Figure 8: Relative throughput comparison between deprecated and Raft-base protocols

5.2 Metrics on MongoDB Atlas

To examine the performance of failover and its impact in production, we analyzed the failover metrics of MongoDB instances deployed on MongoDB Atlas, a hosted database as a service. When the data was collected in June 2020, almost all replica set instances were on the latest major releases: 3.6.18 (26.86%), 4.0.18 (44.17%) and 4.2.6 (28.54%). The vast majority of failovers (89.03%) are caused by planned maintenance, 6.15% by priority takeover (§4.5.2) and 4.82% by election timeout. We focused on those caused by planned maintenance and election timeout to measure the impact of expected and unexpected failovers.

5.2.1 Planned Maintenance

As part of rolling upgrade for planned maintenance on MongoDB Atlas, the old primary will be stepped down via a command. The stepdown command pauses new writes, waits for any eligible secondary to catch up and then asks the eligible secondary to step up, as discussed in §4.5.1. We measure the time duration from starting the election to when the new primary is available for new writes, referred to as *Local Write Unavailability*, and from starting election to when the first no-op write on stepup gets committed, referred to as *Majority Write Unavailability*. Note that the entire unavailable windows perceived by clients are longer since they start from when the old primary pauses writes. The extra unavailability window beyond our measurements heavily depends on the workload and is less comparable across replica sets.

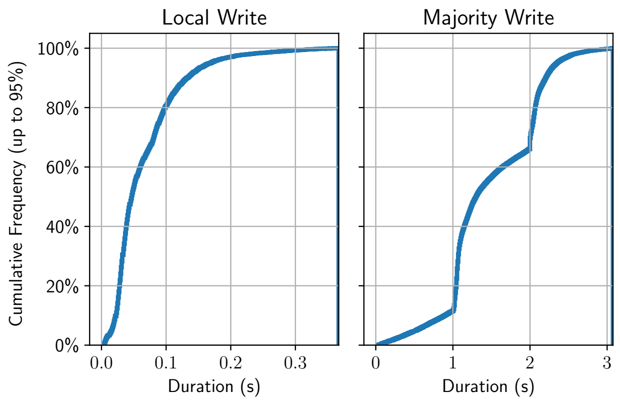


Figure 9: Unavailability due to planned maintenance

Figure 9 shows the cumulative frequency of local and majority write unavailability up to 95th-percentile, 0.37 seconds and 3.08 seconds respectively. Since an election only involves a few round-trips in the same data center, a candidate can finish its election quickly and start to accept new writes. However, it takes longer for other nodes to learn there is a new primary and start to sync from it directly or indirectly. We can observe two clusters of duration for majority writes: around 1 and 2 seconds. We believe this is due to the implementation of sync source selection based on heartbeats. We are actively working on delivering performance improvements in this area to all supported versions.

5.2.2 Election Timeout

When a secondary cannot see a primary for a given election timeout, it runs for election. Failovers caused by election timeout are the fault-tolerant scenarios for which the replication system is designed. We measure the same local and majority write unavailability as above. However, both measurements include the primary catchup phase (§4.3) which involves more work in election timeout cases, so both measurements are longer than that of planned maintenance. By contrast, it is essentially a no-op in planned maintenance since the old primary has waited already. Additionally, overloaded systems are a common reason of failover, which leads to longer primary catchup phases.

The perceived unavailable windows by clients are also longer since they start from when the old primary becomes unresponsive. Usually, they add about an election timeout (10 or 5 seconds on Atlas) to what we measured.

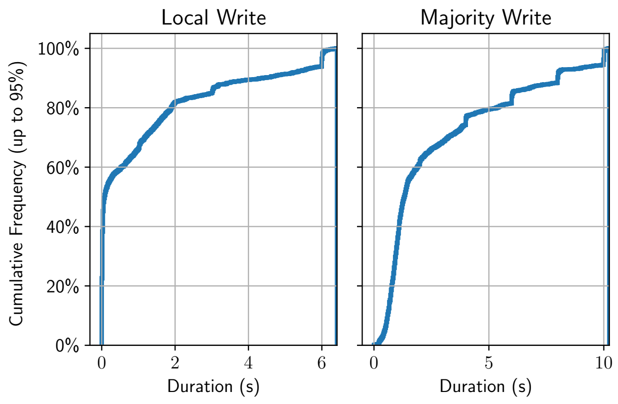


Figure 10: Unavailability due to election timeout

As shown in Figure 10, 95% of failovers due to election timeout start to accept writes within 6.41 seconds after election, and commit their first majority writes within 10.24 seconds. The sharp steps are also aligned with the default 2 second heartbeat intervals.

5.2.3 Replication Network Traffic

MongoDB's consensus protocol allows flexible replication paths and can save cross-datacenter traffic by allowing secondaries syncing from others from the same datacenter. We examined the replication network traffic on Atlas to estimate the cost of cross-datacenter traffic.

Figure 11 shows the distribution of daily replication network traffic on secondaries on Atlas during a week. While some replica sets have no writes and may mainly be used for reads, there are some others that generate gigabytes or terabytes of data. 50% of replica sets generate less than 8.08MB per day, and 95% generate less than 7.94GB per day. Figure 12 shows the distribution of the same data weighted by the daily replication network traffic. It is obvious that a small portion of replica sets generate disproportionately large amount of replication traffic. In fact, the top 5% of all replica sets account for more than 92.6% of all replication network traffic. Given the high traffic volume, it is valuable for a multi-node cross-datacenter deployment to minimize the cost of cross-datacenter traffic from the primary. The cost could be expensive (e.g., \$0.01~\$0.147 per GB on AWS [1]). MongoDB made it possible to minimize cross-datacenter traffic by allowing syncing from another node in the same datacenter instead of the primary.

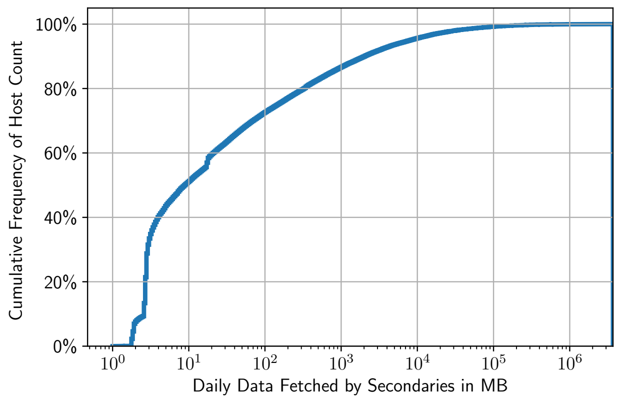


Figure 11: Distribution of daily replication network traffic

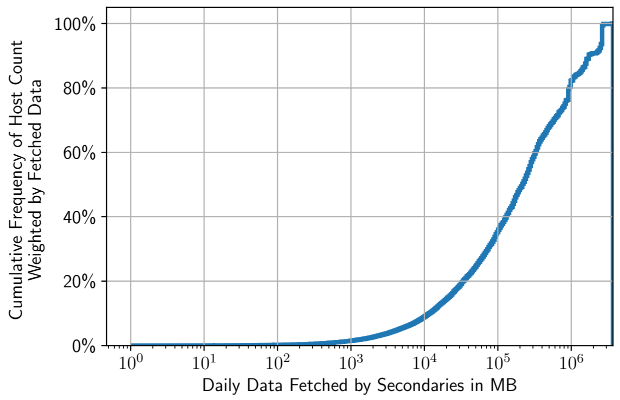


Figure 12: Weighted distribution of daily replication network traffic

6 Related Work

Replication and consensus are both very well-studied areas. This section reviews related works in two main categories: linearizable replication in production databases, and developments in consensus algorithms.

Linearizable replication in production databases. Many production databases build their replication systems via consensus protocols, mostly using Paxos or Raft. One recent popular Paxos-based system is Google's Spanner [7], and examples of Raft-based systems include TiDB [32], RethinkDB [28], CockroachDB [6, 31]. Another approach with similar data replication paths is the primary-backup scheme with external membership management (e.g., a consensus service). Recent database systems of this type include Aurora [35] and FaRM [8, 30]. To the best of our knowledge, these systems do not support pull-based data transfer in their data replication paths.

Another important scheme is Chain Replication (CR) [27]. It can achieve a chaining topology similar to MongoDB. The differences between CR and our proposed scheme are threefold. First, our scheme can support many types of topology and chaining is one typical use case. Second, in our scheme a request can commit after replicating to a majority, while in CR a request needs to replicate to all nodes before it commits. Third, similar to primary-backup schemes, CR needs a third-party to perform a safe leader (head) change. Nevertheless, CR can perform consistent read requests on the tail node, which can improve the system performance, while common consensus-based systems cannot.

CORFU [3] and Delos [2] have the clients (i.e., the learner role in Paxos) pull logs from the server; our approach takes one step further and has the servers (i.e., the acceptor role) pull logs from each other. During this process, we found that at least for Raft, modifying the protocols to enable servers to pull logs from other servers is challenging.

Study in consensus algorithms. Except for searching for more understandable basic consensus protocols, the developments in consensus algorithms can be classified into two categories: 1) optimizations that only require a minimal change to an existing protocol (usually Paxos); 2) new protocols built from scratch or heavily modifying a previous protocol.

The first category of works, in our experience, is closer to our needs in practice. PigPaxos [5] has a group of secondaries relay the messages from the primary to alleviate the primary bottleneck, which achieves similar flexibility as we do. Reconfiguration [18] of a replication group is a major challenge in MongoDB and we developed a refined version of reconfiguration in the latest release 4.4. Leases [10, 22] provide a way to allow consistent local reads in an efficient way. We are investigating this direction for future improvements. One thing we need to point out is that incorporating these or other optimizations of this category into MongoDB may be more difficult than into other systems that use stock Paxos protocols, because our protocol is a heavily modified version of Raft. However, it has been demonstrated [36] that such porting of optimizations can be achieved by drawing a one-to-one correspondence between each step of these protocols, e.g., by using refinement mapping.

It is an interesting and open question how MongoDB can benefit from the optimizations in the second category of works. These optimizations are often disruptive but effective. For example, to improve the performance for geo-replication, one can shard the log space [20], use fast quorums [23], or rethink the layering between replication and the rest of the system [15, 19, 37]. Replaying these works in MongoDB may require more efforts because it could suggest a complete reconstruction of the system.

7 Conclusion

In this paper we presented the design and implementation of the fault-tolerant and linearizable replication system in MongoDB. We proposed a novel pull-based consensus protocol that is a modification of Raft. With this pull-based scheme, MongoDB allows a more flexible control of data transmission paths. We described how this consensus protocol works, how MongoDB integrates it with the rest of the replication system, and the extensions of the replication protocol that support our rich feature set. We reported our evaluation on EC2 and our data analysis on MongoDB's cloud platform, and concluded that MongoDB can replicate data efficiently and reliably.

Acknowledgments

We thank our shepherd Mahesh Balakrishnan and anonymous reviewers at NSDI for providing valuable feedback. We thank Robert Guo for his guidance and assistance on performance testing. We thank William Schultz for contributing to the formal definition of majority commitment. We thank Bruce Lucas and Eric Milkie for helping investigate performance bottlenecks in the experiments. We thank the Data Engineering team and the Cloud team for collecting and maintaining the Atlas metrics and enabling the analysis of Atlas metrics. We thank our internal reviewers: Tess Avitabile, William Schultz, Andy Schwerin, Michael Cahill, Judah Schvimer, Robert Guo, and Asya Kamsky. Finally, we would like to thank the entire MongoDB Replication team and its alumni, all of whom contributed to the design and implementation of the protocol and features described in this paper.

References

[1] *Amazon EC2 On-Demand Pricing*. https://aws.amazon.com/ec2/pricing/on-demand/.

[2] M. Balakrishnan, J. Flinn, C. Shen, M. Dharamshi, A. Jafri, X. Shi, S. Ghosh, H. Hassan, A. Sagar, R. Shi, et al. Virtual Consensus in Delos. In *Proc. OSDI*. Oct. 2020.

[3] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobbler, M. Wei, and J. D. Davis CORFU: A Shared Log Design for Flash Clusters. In *Proc. NSDI*. Apr. 2012.

[4] M. Burrows The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *Proc. OSDI*. Nov. 2006.

[5] A. Charapko, A. Ailijiang, AND M. Demirbas PigPaxos: Devouring the communication bottlenecks in distributed consensus. *arXiv preprint arXiv:2003.07760* (2020).

[6] *CockroachDB*. http://www.cockroachlabs.com/.

[7] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, ET AL. Spanner: Google's globally distributed database. In *Proc. OSDI*. Oct. 2012.

[8] A. Dragojević, D. Narayanan, M. Castro, AND O. Hodson FaRM: Fast remote memory. In *Proc. NSDI*. Apr. 2014.

[9] M. J. Fischer, N. A. Lynch, AND M. S. Paterson Impossibility of distributed consensus with one faulty process. *JACM* 32, 2 (Apr. 1985).

[10] C. Gray, AND D. Cheriton Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. *ACM SIGOPS Operating Systems Review* 23, 5 (Nov. 1989).

[11] R. Guo Mongodb's javascript fuzzer. *ACM Queue* 15, 1 (Mar. 2017).

[12] M. P. Herlihy, AND J. M. Wing Linearizability: A correctness condition for concurrent objects. *TOPLAS* 12, 3 (1990).

[13] P. Hunt, M. Konar, F. P. Junqueira, AND B. Reed ZooKeeper: wait-free coordination for internet-scale systems. In *Proc. ATC*. June 2010.

[14] A. Kamsky Adapting tpc-c benchmark to measure performance of multi-document transactions in MongoDB. *PVLDB* 12, 12 (2019).

[15] T. Kraska, G. Pang, M. J. Franklin, S. Madden, AND A. Fekete MDCC: multi-data center consistency. In *Proc. EuroSys*. Apr. 2013.

[16] F. Lai, M. Chowdhury, AND H. Madhyastha To relay or not to relay for inter-cloud transfers? In *HotCloud 18*. 2018.

[17] L. Lamport Paxos made simple. *SIGACT* 32, 4 (2001).

[18] L. Lamport, D. Malkhi, AND L. Zhou Reconfiguring a state machine. *SIGACT* 41, 1 (2010), 63–73.

[19] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, AND A. El Abbadi Low-latency multi-datacenter databases using replicated commit. *PVLDB* 6, 9 (July 2013).

[20] Y. Mao, F. P. Junqueira, AND K. Marzullo Mencius: building efficient replicated state machines for WANs. In *Proc. OSDI*. Dec. 2008.

[21] *MongoDB Atlas*. https://www.mongodb.com/cloud/atlas.

[22] I. Moraru, D. G. Andersen, AND M. Kaminsky Paxos quorum leases: Fast reads without sacrificing writes. In *Proc. SoCC*. Nov. 2014.

[23] I. Moraru, D. G. Andersen, AND M. Kaminsky There is more consensus in egalitarian parliaments. In *Proc. SOSP*. Nov. 2013.

[24] B. M. Oki, AND B. H. Liskov Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proc. PODC*. June 1988.

[25] D. Ongaro Consensus: Bridging theory and practice. PhD thesis. Stanford University, 2014.

[26] D. Ongaro, AND J. K. Ousterhout In search of an understandable consensus algorithm. In *Proc. ATC*. June 2014.

[27] R. van Renesse, AND F. B. Schneider Chain replication for supporting high throughput and availability. In *Proc. OSDI*. Dec. 2004.

[28] *RethinkDB*. http://www.rethinkdb.com/.

[29] W. Schultz, T. Avitabile, AND A. Cabral Tunable consistency in MongoDB. *PVLDB* 12, 12 (Aug. 2019).

[30] A. Shamis, M. Renzelmann, S. Novakovic, G. Chatzopoulos, A. Dragojević, D. Narayanan, AND M. Castro Fast general distributed transactions with opacity. In *Proc. SIGMOD*. June 2019.

[31] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss, ET AL. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proc. SIGMOD*. June 2020.

[32] *TiDB*. http://www.pingcap.com/.

[33] *TLA+ Specification of MongoDB Replication Protocol*. https://github.com/mongodb/mongo/blob/master/src/mongo/db/repl/tla_plus/RaftMongo/RaftMongo.tla.

[34] M. Tyulenev, A. Schwerin, A. Kamsky, R. Tan, A. Cabral, AND J. Mulrow Implementation of cluster-wide logical clock and causal consistency in MongoDB. In *Proc. SIGMOD*. June 2019.

[35] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, AND X. Bao Amazon Aurora : Design Considerations for High Throughput Cloud-Native Relational Databases. In *Proc. SIGMOD*. May 2017.

[36] Z. Wang, C. Zhao, S. Mu, H. Chen, AND J. Li On the parallels between Paxos and Raft, and how to port optimizations. In *Proc. PODC*. July 2019.

[37] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, AND D. R. K. Ports Building consistent transactions with inconsistent replication. In *Proc. SOSP*. Oct. 2015.

Appendix A: Comparison between Raft and MongoDB Consensus

Raft Protocol

Minor changes are marked in *italics*, e.g., changing from index to timestamp, comparison of OpTimes (<term, timestamp> pairs) and introducing heartbeats; major behavioral changes are marked in red.

State

Persistent state on all servers:
(Updated on stable storage before responding to RPCs)

currentTerm	latest term server has seen (initialized to 0 on first boot, increases monotonically)
votedFor	candidateId that received vote in current term (or null if none)
log[]	log entries; each entry contains command for state machine, and term when entry was received by leader (first index is 1)

Volatile state on all servers:

commitIndex	index of highest log entry known to be committed (initialized to 0, increases monotonically)
lastApplied	index of highest log entry applied to state machine (initialized to 0, increases monotonically)

Volatile state on leaders:
(Reinitialized after election)

nextIndex[]	for each server, index of the next log entry to send to that server (initialized to leader last log index + 1)
matchIndex[]	for each server, index of highest log entry known to be replicated on server (initialized to 0, increases monotonically)

AppendEntries RPC

(Invoked by leader to replicate log entries; also used as heartbeat.)

Arguments:

term	leader's term
prevLogIndex	index of log entry immediately preceding new ones
prevLogTerm	term of prevLogIndex entry
entries[]	log entries to store (empty for heartbeat; may send more than one for efficiency)
leaderCommit	leader's commitIndex

Results:

term	currentTerm, for leader to update itself
success	true if follower contained entry matching prevLogIndex and prevLogTerm

Receiver implementation:

1. Reply false if term > currentTerm
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it
4. Append any new entries not already in the log
5. If leaderCommit < commitIndex, set commitIndex = min(leaderCommit, index of last new entry)

RequestVote RPC

Invoked by candidates to gather votes.

Arguments:

term	candidate's term
candidateId	candidate requesting vote
lastLogIndex	index of candidate's last log entry
lastLogTerm	term of candidate's last log entry

Results:

term	currentTerm, for candidate to update itself
voteGranted	true means candidate received vote

Receiver implementation:

1. Reply false if term < currentTerm
2. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote

Rules for Servers

All Servers

- If commitIndex > lastApplied: increment lastApplied, apply log[lastApplied] to state machine
- If RPC request or response contains term T > currentTerm: set currentTerm = T, convert to follower

Followers

- Respond to RPCs from candidates and leaders
- If election timeout elapses without receiving AppendEntries RPC from current leader or granting vote to candidate: convert to candidate

Candidates

- On conversion to candidate, start election:
  - Increment currentTerm
  - Vote for self
  - Reset election timer
  - Send RequestVote RPCs to all other servers
- If votes received from majority of servers: become leader
- If AppendEntries RPC received from new leader: convert to follower
- If election timeout elapses: start new election

Leaders

- *Upon election: send initial empty AppendEntries RPCs (heartbeat) to each server; repeat during idle periods to prevent election timeouts*
- If command received from client: append entry to local log, respond after entry applied to state machine
- If last log index ⩾ nextIndex for a follower: send AppendEntries RPC with log entries starting at nextIndex
  - If successful: update nextIndex and matchIndex for follower
  - If AppendEntries fails because of log inconsistency: decrement nextIndex and retry
- If there exists an N such that N > commitIndex, a majority of matchIndex[i] ⩾ N, and log[N].term == currentTerm: set commitIndex = N

MongoDB Consensus Protocol

State

Persistent state on all servers:
(Updated on stable storage before responding to RPCs)

currentTerm	latest term server has seen (initialized to 0 on first boot, increases monotonically)
votedFor	candidateId that received vote in current term (or null if none)
log[]	log entries; each entry contains command for state machine, *timestamp* and term when entry was received by leader

Volatile state on all servers:

lastCommitted	*OpTime* of highest log entry known to be committed (initialized to minimum, increases monotonically)
lastApplied	*OpTime* of highest log entry applied to state machine (*initialized to last log entry's OpTime*, increases monotonically)
lastPosition[]	for each server, OpTime of highest log entry known to be replicated on that server

RequestVote RPC (Omitted)

. . . The same as Raft's RequestVote RPC *except changing index to timestamp* . . .

PullEntries RPC

(Replicate log entries from its sync source.)

Arguments:

prevLogTimestamp
timestamp of last fetched log entry.

Results:

entries[]	log entries with a timestamp greater than or equal to prevLogTimestamp
commitPoint	sync source's lastCommitted

Receiver implementation:

1. Return the log entries with a timestamp greater than or equal to prevLogTimestamp. Could be empty if no such entry exists.

Sender implementation after RPC call:

1. If returned entries is empty or last OpTime in entries is less than last OpTime in log, select a new sync source and retry PullEntries.
2. If last log entry conflicts with the first of entries (*due to different OpTimes*)
   (a) Traverse the log on sync source backwards until a common entry is found
   (b) Delete all existing entries following the common entry
   (c) Roll back the data to the state right after the common entry
3. Append any new entries not already in the log
4. If commitPoint > lastCommitted, set lastCommitted = min(commitPoint, *OpTime* of last new entry)

UpdatePosition RPC

(Sending latest positions of all known nodes to sync source.)

Arguments:

term	currentTerm, for leader to update itself
position[]	lastPosition[] on the sender

Results: None

Receiver implementation:

1. Merge position and lastPosition to record the highest known position for each member
2. Send UpdatePosition RPC to sync source if the receiver has one

Heartbeat RPC

(Used for liveness monitoring, commit point propagation, and sync source selection.)

Arguments:

term	sender's term
senderId	sender's node Id
role	sender's role
position	sender's last log entry's OpTime
commitPoint	sender's lastCommitted

Receiver implementation

1. Record the role and the current time of the last heartbeat for senderId for liveness monitoring
2. Update lastPositions[senderId] to position if position is higher, for sync source selection
3. If commitPoint > lastCommitted, set lastCommitted = min(commitPoint, *OpTime* of last new entry)

Rules for Servers

All Servers

- Apply log entries speculatively when appending them to log
- If RPC request or response contains term T > currentTerm: set currentTerm = T, convert to follower

Followers

- Respond to RPCs from candidates and leaders
- If election timeout elapses without *receiving Heartbeat RPC from current leader*: convert to candidate

Candidates

- On conversion to candidate, start election:
  – Increment currentTerm
  – Vote for self
  – Reset election timer
  – Send RequestVote RPCs to all other servers
- If votes received from majority of servers: become leader
- If election timeout elapses: start new election

Leaders

- If command received from client: append entry to local log, respond after lastCommitted > entry's OpTime.
- *If there exists an entry such that entry.OpTime > lastCommitted, a majority of lastPosition[i] ⩾ entry.OpTime, and entry.term == currentTerm: set lastCommitted = entry.OpTime*

Appendix B: TPC-C Experiments

Setup. This section uses 3-way replication with 3 server VMs and 1 client VM. All VMs are deployed in the same Availability Zone. Each client thread continuously inserts documents of 1000 bytes in a closed-loop. We vary the number of client threads to control the offered load in the system.

	chaning disabled			chaining enabled		
	throughput (ops/s)	50% latency (ms)	95% latency (ms)	throughput (ops/s)	50% latency (ms)	95% latency (ms)
DELIVERY	58.16	144.14	257.72	60.21	129.94	247.22
NEW_ORDER	650.33	70.68	153.64	675.33	66.2	141.09
ORDER_STATUS	58.01	30.32	56.17	59.71	32.95	62.54
PAYMENT	628.51	37.04	169.28	651.46	37.91	171.52
STOCK_LEVEL	58.46	7.25	22.6	60.28	7.49	23.88
TOTAL	1453.47			1506.98		

Table 1: Results of an adapted TPCC benchmark. The benchmark denormalizes the data and leverages MongoDB query language and transaction semantics to be consistent with MongoDB best practices [14]. The test is run with 100 client threads and 100 warehouses on the same replica set setting as above without any bandwidth limit. Since the TPCC workload is CPU-bound, the performances of both chaining enabled and disabled settings are very close. Chaining-enabled case performs slightly better because it offloaded one secondary's oplog reading from the primary to a secondary and saved the CPU on the primary. The network was not saturated by replication: the chaining-disabled primary sent 15.56 MB/s in total over the network, including serving client requests and 4.05 MB/s to each secondary for oplog replication, while the chaining-enabled primary sent 11.70 MB/s in total. The gap of primary's network traffic is aligned well with the saved oplog replication traffic.

Mistify: Automating DNN Model Porting for On-Device Inference at the Edge

Peizhen Guo Bo Hu Wenjun Hu
Yale University

Abstract

AI applications powered by deep learning inference are increasingly run natively on edge devices to provide better interactive user experience. This often necessitates fitting a model originally designed and trained in the cloud to edge devices with a range of hardware capabilities, which so far has relied on time-consuming manual effort.

In this paper, we quantify the challenges of manually generating a large number of compressed models and then build a system framework, *Mistify*, to automatically port a cloud-based model to a suite of models for edge devices targeting various points in the design space. *Mistify* adds an intermediate "layer" that decouples the model design and deployment phases. By exposing configuration APIs to obviate the need for code changes deeply embedded into the original model, *Mistify* hides run-time issues from model designers and hides the model internals from model users, hence reducing the expertise needed in either. For better scalability, *Mistify* consolidates multiple model tailoring requests to minimize repeated computation. Further, *Mistify* leverages locally available edge data in a privacy-aware manner, and performs run-time model adaptation to provide scalable edge support and accurate inference results. Extensive evaluation shows that *Mistify* reduces the DNN porting time needed by over 10× to cater to a wide spectrum of edge deployment scenarios, incurring orders of magnitude less manual effort.

1 Introduction

AI-driven intelligent edge has already become a reality [9], where millions of mobile and IoT devices or edge servers analyze real-time data and transform those into actionable insights on user-facing devices. For example, real-time video analytics (e.g., traffic monitoring [43], security surveillance [5], and smart retail [3]), natural language understanding (e.g., virtual assistance, smart email composition [63]), visual assistance [48], and industrial automation (e.g., defect detection, assembly line management [1, 5]) are already everyday examples. It is projected that, by 2022, over 60% of the data locally generated by IoT, sensor, and mobile devices will drive real-time intelligent decisions; 80% of the IoT and mobile devices shipped will have on-device AI capabilities [6, 16].

Many AI functionalities today are powered by deep learning (DL), with a significant computation footprint. While edge devices used to primarily offload related computation to the cloud, increasingly inference workloads are run natively on the edge devices to provide better interactive user experience (e.g., $\sim$10 ms real-time response), data privacy, and reliability [60]. This often necessitates *porting* (i.e., *tailoring* and deploying) a deep neural network (DNN) model originally designed and trained on the cloud to edge settings.

Model porting is a non-trivial process even for a single target. From an *algorithmic* perspective, the core techniques involved are called *model tailoring* in the machine learning literature. There are two steps, adapting the architecture of a pre-trained model to fit a new resource specification, followed by fine-tuning the new model parameters. Although there have been numerous model tailoring *algorithms* [22, 25, 31, 82], the complete porting process additionally requires "executing" the algorithms by correctly annotating a source model, and then retraining the annotated model with the right data. By various estimates, there will be over 50 billions IoT devices [30] with very diverse hardware profiles. This creates a massive design space for optimizing the resource usage and performance of a new model.

Unfortunately, the current practice of porting relies on manual effort, which simply cannot scale with the sheer size of the design space. There are two issues: laborious manual annotations and the computational complexity. Even if porting is a one-time need, it takes time to meticulously annotate the original model to embed the correct model tailoring objectives. For instance, constructing the model tailoring logic for ResNet50 [35] requires around 30 lines of source code edits scattered around several files. Further, model adaptation incurs significant computational complexity. Existing algorithms can handle generating one model, but can not scale well to large batches of model generation. If many model variants are needed, either for different device hardware specifications or for different runtime conditions, manual tailoring incurs significant repeated efforts. The effort needed to tailor model variants could match that for training an original model. Therefore, app developers currently perform little platform-specific customization to the intractable target space [75], even though lack of customization results in suboptimal performance. (Section 2)

Fundamentally, the problem is the implicit coupling between model design and deployment currently. Model designers need to both improve the inference accuracy and minimize the memory and computation footprint for deployment. Model users need to both compress the model without degrading accuracy significantly and accelerate the inference. Both stages require significant expertise spanning deep learning algorithms and system runtime management.

In this paper, therefore, we build *Mistify*, a system *framework* to automate and scale the porting process from a pre-trained model to a suite of compact models tailored to diverse edge resource specifications (Section 3). *Mistify* does not propose any new model tailoring algorithm. Instead, it wraps over existing model tailoring algorithms and provides additional system services for scalability. This is analogous to a scheduling framework implementing common scheduling algorithms so that application developers can outsource scheduling considerations. With *Mistify*, model users (i.e., mobile app developers, often non-machine-learning experts) can outsource model adaptation instead of understanding the specifics of the model to adapt. In this way *Mistify* takes on the role of a provider of models for on-device inference execution. Meanwhile, model designers (i.e., machine learning experts) can use simple resource abstractions to evaluate the model instead of undergoing detailed resource profiling.

From a *system* perspective, we propose new abstractions to separate the model semantics from the execution characteristics and several techniques (*Collective adaptation*, *Privacy-aware knowledge distillation*, and *Downtime-free run-time model generation and switching*), all incorporated in an end-to-end framework. *Mistify* minimizes the need for "compile-time" code changes when generating a model statically. Instead of requiring the user to annotate the original model, *Mistify generates* model adaptation logic from the configuration file to correctly and scalably tailor to the resource budgets and performance requirements of each device while minimizing duplicate iterations (Section 4). Further, *Mistify* leverages implicitly correlated edge data in a privacy-aware manner to balance training data privacy and model accuracy (Section 5). During the run time of the inference task, *Mistify* employs a feedback mechanism to generate new models as needed to adapt to fluctuating application demands and resource availability (Section 6).

Note that we make a distinction between the end-to-end process (*porting*) and individual model compression techniques (*neural architecture search*, *layer pruning*, etc.). The latter do not always produce a readily usable compressed model. The (adapted) model still needs (re)training and that process incurs several practical difficulties. In contrast, *Mistify* automates the entire end-to-end process. Making it scalable and adaptive while keeping training data local are non-trivial efforts, and extend significantly beyond simply implementing known algorithms for each component in one system.

Mistify is implemented following a client-server model, built on TensorFlow [13]. We build example wrappers to adopt state-of-the-art model adaptation algorithms like MorphNet [31] and ChamNet [25], and evaluate *Mistify* using representative vision and natural language processing (NLP) models trained with widely used standard datasets. Extensive evaluation shows that *Mistify* reduces the DNN porting time needed by over 10× and incurs orders of magnitude less manual effort, all with little or up to 1% accuracy loss when compared to manually running the adaptation algorithms. This loss margin is well within the typical accuracy loss budget for on-device inference [64].

Mistify is far more than a tool for convenience. It serves as an intermediate layer that decouples the model design and deployment stages. Model designers can focus on model performance and advanced architecture design, without worrying about deployment difficulties, whereas edge users can focus on execution-centric issues such as optimizing the executable binaries, computation kernels, and job scheduling, without worrying about the inner workings of the model.

To summarize, this paper makes three contributions: First, we quantify the scalability challenges of porting pre-trained DNN models to edge settings to motivate framework support. Second, we design and implement *Mistify* as a framework for automated porting at scale. *Mistify* achieves scalability with collective adaptation and improves model quality with privacy aware knowledge distillation and run-time model adaptation. Third, *Mistify* provides a clean interface to separate DNN model design and deployment. This could lower the bar for wider usage of on-device deep learning at the edge.

2 Background and motivation

The lifecycle of a DNN model spans design and deployment, and the need for automating model porting arises from the complexity of the process. We discuss these in detail before outlining the challenges and solutions.

2.1 Current DNN lifecycle

The lifecycle of a DNN encompasses at least three stages: model design, publishing, and deployment. Publishing mainly requires adding a well-trained model to public repositories, while design and deployment are more involved.

DNN model design. Today's models are designed for either optimal inference quality or minimal resource footprint.

The former is typically assumed for workloads run on the cloud. Given increasing computation power, cloud-centric models employ advanced neural network topologies, millions of parameters and floating-point operations (FLOPs) to achieve the *highest accuracy*. For example, BERT [28] and ResNeXt [51] have 340 and 829 million parameters respectively, hence extremely computation intensive.

The latter goal is geared towards resource-constrained edge devices, including IoT nodes, smartphones and tablets. The desirable models (e.g., MobileNet [38] and SqueezeNet [40]) are exceedingly compact, requiring only a few MBs for storage and affordable computing budget, ready to run across diverse hardware. However, these DNNs sacrifice accuracy in exchange for super lightweight execution, aiming at *maximal deployment coverage*.

DNN deployment at the edge. Many DL inference engines

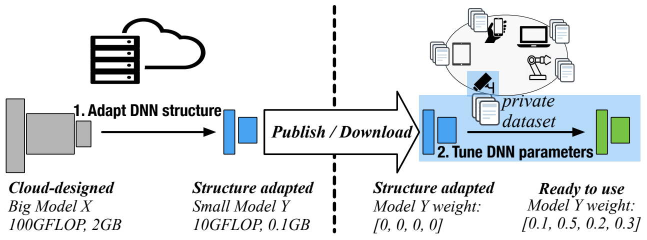


Figure 1: Steps to port a DNN model to an edge setting.

have been developed to serve DNN workloads on edge devices. They focus on deployment optimizations such as cross-platform compatibility, trimming executable size, and low-complexity operator kernels [2, 24]. Once DNN models are loaded (e.g., from model repositories or custom URLs), these engines can execute the inference tasks efficiently.

Transition from design to deployment. When a pre-trained model is ill-suited to a desirable deployment setting, it needs to be *tailored* to the new resource budget and performance goals. This requires *adapting* the model architecture (e.g., by trimming network connections, skipping layers, quantizing parameters) and then *fine-tuning* (i.e., retraining) the parameters with local datasets (Figure 1). However, the end-to-end model porting process is complex. The source model needs to be correctly annotated to have its architecture adapted to a desired setting. Fine-tuning also requires careful usage of the training data to balance training quality (i.e., effective specialization without overfitting) and data privacy.

2.2 The complexity of porting DNN models

As more edge devices adopt on-device inference, porting cloud-based models to edge settings becomes increasingly complex, facing several challenges: (i) the range of model adaptation targets is huge as a result of the diversity in the hardware specification; (ii) the porting process involves several stages, each requiring coordination between multiple parties; (iii) run-time dynamics and new deployment settings may necessitate frequent model re-adaptations.

Heterogeneous execution environment. Edge devices are incredibly diverse, ranging from embedded sensors, IoT devices, mobile phones/tablets, to edge servers, covering a full spectrum of hardware capability [75]. Table 1 lists the specifications of some GPU and ASIC accelerators and processors, from high-end to low-end, widely employed at the edge for DNN-based workloads. For the same DNN inference workload, the completion times for low-end (e.g., Jetson nano) and high-end (e.g., 2080) devices can differ by orders of magnitude (e.g., 229 ms vs 9.8 ms to run inference with ResNet).

Meanwhile, the DNN inference times for the same task can differ by up to 8×, and the quality (e.g., in *accuracy*, *F1 score*) varies by as much as 25% [18, 66]. It is therefore essential to match the desirable performance with the hardware capability.

These numbers outline a massive design space to explore different tradeoff points between inference accuracy and latency, where a sub-optimal choice could incur up to 10% accuracy loss (e.g., when running EfficientNet-BO unnecessarily on the latest iPhone model) or miss the latency requirement for real-time processing by over 100 ms (e.g., running ResNet on a low-end smartphone) [76].

Table 1: Popular DL hardware specifications.

GPU	Peak perf	Memory	Bandwidth
V100	112 TFLOP	32 GB	900 GB/sec
2080	11.7 TFLOP	11 GB	480 GB/sec
Edge GPU	**Peak perf**	**Memory**	**Bandwidth**
Jetson TX2	1.5 TFLOP	4 GB	58 GB/sec
Jetson nano	0.47 TFLOP	4 GB	25 GB/sec
ASIC	**Peak perf**	**Memory**	**Bandwidth**
Edge TPU [4]	4 TFLOP	-	-
Raspberry pi	6 GFLOP	2 GB	8.5 GB/sec

Clearly, one size does not fit all, but nor would a few sizes only. Instead, it is desirable to tailor to each target at a fine granularity. For instance, EfficientNet-B4 (a popular model occupying a sweet spot of computation complexity and prediction accuracy) is suitable for Samsung S9, achieving 83% accuracy and 50 fps real-time response rate. However, using the same DNN on its immediate predecessor (S8) and successor (S10) would reduce the response rate by 14 fps for S8 and the accuracy by nearly 1% for S10. These are significant to the model designers where even 0.1% accuracy improvement merits tremendous effort (both intellectually and computationally) into model design and training. Given the ever increasing size of this adaptation space, it is impractical to either cover all plausible operation points with a few DNN models, or manually exhaust the entire space to customize the adaptation tradeoff for each possible individual edge setting.

Multi-stage multi-party efforts. Tailoring a DNN model involves first adapting to the right model architecture, and then fine-tuning the model parameters (Figure 1).

The first stage is resource heavy and therefore takes place where the original models are trained (i.e., in the *cloud*).The second stage increasingly takes place at the *edge* given the push for on-device inference and private learning. Edge devices collect and maintain specialized data relevant to the local context for model training [19, 42] and local data are typically privacy sensitive [59]. However, smaller networks with less abundant datasets are well known to be much harder to train, as it is easy to overfit the model to the training data such that the model may not generalize well to unseen test data [37, 79]. Thus, it is also preferable for the edge to take advantage of relevant datasets available elsewhere (e.g., in the cloud or on other devices) to enhance the training dataset and improve training quality.

To sum up, both stages of model tailoring require coordination between the cloud and the edge, and resolving the conflict between data privacy and fine-tuning quality.

Fluctuating run-time characteristics. The run-time characteristics of deep learning inference tasks are highly dynamic, shown in two aspects. First, the performance requirements, e.g., accuracy and response time, of an inference task change frequently. For instance, the accuracy requirements of a vision-based security surveillance workload differ between crucial and trivial moments, while the latency requirements fluctuate across peak and off-peak hours (e.g., daytime and night) [41,42]. Further, the commonly used metric, FLOPS, is sometimes an inaccurate proxy to statically estimate run-time latency [72]. Second, the resource availability (e.g., memory space, CPU cycles, accelerator quotas), varies on the edge device due to other workloads competing for the same resource. For instance, when an edge device launches or completes a workload, or adjusts the resource allocation of the containers that serve the inference tasks, the perceived resource availability to the active workloads changes [23,74].

Frequent changes in the performance requirements and resource availability necessitate a mechanism to better serve combinations of the individual operation points, including a suite of models to switch to dynamically, and asynchronously tailoring new ones as the demand warrants.

2.3 The need to automate DNN porting

Current practice. To tailor DNNs towards heterogeneous deployment settings, currently either the model designers should generate different DNNs to cater to each possible resource budget and performance goal, or the model users should prepare the custom datasets, select and apply the algorithms to tailor already published DNNs towards their custom settings.

The latest adaptation algorithms, such as AutoML [82], EfficientNet [73], and others [14,22,25,31], all address target-specific adaptation *case by case* as an additional step in the *design* phase. While the techniques differ (e.g., gradient-based, evolutionary, and recurrent neural network based), they revise the model architecture to be closer to the required resource and performance targets with successive training iterations.

Problems with current porting practice. The overarching problems of the existing practice are they do not scale from a system perspective (e.g., hundreds of GPU hours for a single setting [71,82]) and largely rely on manual effort (e.g., thousands of lines of code spread across source files [10]). Such a manual tailoring process is not easily turned to a configuration style that is agnostic to the number of cases because distinct structure adapting terms have to be added to different DNN models/layers and at specific positions, which makes it difficult and error-prone. Furthermore, it is infeasible for the model designers to prepare for all possible deployment settings, or for the model users to be well versed in machine learning literature to run the right algorithm.

The need for an automated framework. The current model porting process implicitly couples DNN design and deployment, even though they are conceptually separate stages. This coupling introduces unnecessary complexity to both model designers and model users. This motivates adding a separate *model porting* stage to the model lifecycle, i.e., an intermediary to decouple design from deployment and automatically port pre-trained DNNs towards heterogeneous edge settings.

Mistify is therefore built as an intermediate framework to encapsulate diverse adaptation algorithms and address the end-to-end porting challenges outlined above, analogous to scheduler *frameworks* for distributed systems implementing scheduling *algorithms* and providing services.

2.4 System requirements

To address the challenges above, an automated model porting framework should meet the following requirements.

Avoiding deeply embedded and unscalable manual code changes. Since the existing model adaptation step is often coupled with model design itself, a side effect is that relevant code changes are embedded deep into the model design code. Therefore, the system challenge is to simplify the code modifications needed to specify the adaptation logic.

Mistify addresses this challenge in two steps (Section 4). First, we expose the right high-level abstractions of adaptation choices to users. This elevates per-model code edits (embedded in the particular script specifying the model) to framework level configuration parameter changes. Second, we parse the adaptation requirements from the configuration files and merge implicitly correlated model adaptation requests to reduce duplicate iterations and improve scalability.

Cloud-edge coordination. To automate the two-stage model tailoring process with the best training outcome, the main challenge is to simultaneously ensure that private data stay local but parameter tuning can benefit from the data distributed across devices. We address this by adopting mutual knowledge distillation. Our system implicitly coordinates multiple devices in the same tailoring batch to maximally "share" available training data in a privacy aware fashion, without explicitly exchanging and examining the raw data (Section 5).

Fast response to run-time dynamics. Fundamentally, the system challenge is to effectively handle the mismatch between a statically trained model and the dynamic execution environments during run time. Specifically, this requires generating new models as needed and switching to them with minimal downtime. We address the challenge with a feedback mechanism between the model deployment points (e.g., edge devices) and the model tailoring point (e.g., a central server or cloudlet) to perform real-time DNN re-adaptation (Section 6).

3 *Mistify* demystified

The overarching goal for *Mistify* is two-fold: (i) *Mistify* should separate the model design and deployment stages with a clean interface; and (ii) *Mistify* should bridge the two stages with a framework that automatically explores the design space at scale and generates models best suited to user-specified tradeoff points, hiding such complexity from both sides.

Therefore, *Mistify* is designed as an intermediate framework between DNN model design toolkits and deployment

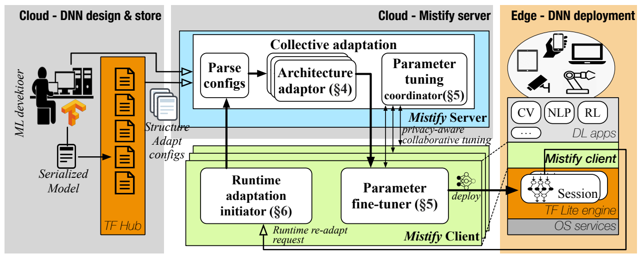


Figure 2: *Mistify* system architecture.

engines, as shown in Figure 2. The arrows across different shaded blocks show how *Mistify* interacts with model designers and users. *Mistify* exposes APIs to the model users and inference engines to specify their *porting configurations* (Figure 3), either in a *batch* mode during initialization or in a *streaming* mode incrementally during run time.

The primary challenge for *Mistify* is therefore how to generate a large number of adapted DNN models with minimal computation and manual intervention. In general, our approach is *collective adaptation*, i.e., parsing adaptation goals and harnessing the implicit correlation among the goals to reduce unnecessary computation (Section 4). *Mistify* parses a collection of individual adaptation goals into a dependency tree with each node corresponding to a distinct goal, so that each goal is adapted only from its immediate parent via a desirable, automatically selected adaptation algorithm. Next, the adapted models are distributed to the endpoints, where the *Mistify* client runtime will prepare the deployment of the adapted model by fine-tuning the parameters (Section 5). Finally, the models start running on edge devices, and the *Mistify* client monitors the execution environment (e.g., resource availability and desirable performance goals). The *Mistify* client will trigger on-demand model re-adaptation asynchronously when the environmental changes warrant a new model (Section 6).

Example deployment strategies. Following the common practice of on-device DL inference deployment [2, 5], *Mistify* can be deployed in two ways. The *Mistify* server can be deployed in the cloud by the DNN application developers, interfacing with the model repository (e.g., TF-Hub) and exposing APIs to the public. Alternately, the server can be maintained by the model users (e.g., edge device administrators) in their private clouds to serve local devices (e.g., IoT nodes). *Mistify* clients are simply deployed on the edge devices as a module extension to the native DL engine.

***Mistify* server.** The *Mistify* server consists of two functional modules: an *architecture adaptor* and a *parameter tuning coordinator*. Once the *architecture adaptor* receives the original DNN model and the adaptation settings from the model users and/or the *Mistify* client, it generates the adapted models and sends those to the corresponding clients (Section 4). The *parameter tuning coordinator* serves as the central point to coordinate the parameter fine-tuning process across the *Mistify* clients (Section 5.2), whereas the actual tuning logic is executed on each client locally (Section 5.1).

```
[ // start of all configurations
    {
        "id": "1",
        "model": "Resnet",
        "dataset": {
            "train": "/path/to/train",
            "test": "/path/to/test"
        },
        "algorithm": {
            "name": "Morphnet",
            "config": {
                "threshold": 0.1,
                "init_reg_strength": 1e-9
            }
        },
        "adaptation_goal": {
            "latency": "30ms",
            "accuracy": 0.80,
            "FLOP": "5G",
            "num_of_params": "20M"
        }
    },
    // more configurations ...
```

```
    // ... more configurations,
    {
        "id": "9",
        "model": "Efficientnet",
        "dataset": {
            "train": "/path/to/train",
            "test": "/path/to/test"
        },
        "algorithm": {
            "name": "Chamnet",
            "config": {
                "init_population": 10,
                "crossover_rate": 0.7,
                "mutation": 0.08
            }
        },
        "adaptation_goal": {
            "latency": "30ms",
            "accuracy": 0.80,
            "FLOP": "5G",
            "num_of_params": "20M"
        }
    }
] // end of all configurations
```

Figure 3: Example porting configurations.

***Mistify* client.** The *Mistify* client consists of a *run-time adaptation initiator*, a *parameter fine-tuner* and a *run-time performance monitor*. The *run-time adaptation initiator* intercepts the native DNN model loading path of the inference engine to automatically trigger model adaptation during initialization, and then listens for run-time re-adaptation requests. The *parameter fine-tuner* takes an adapted DNN model as the starting point, optimizes its parameters jointly based on the local (private) training data and the guidance from the correlated neighboring counterparts (coordinated by the *Mistify* server). This approach aims to overcome overfitting while maintaining data privacy. The *run-time monitor* tracks the current performance as well as resource availability. Once these profiles change significantly, it will trigger an online model switching as well as an offline re-adaptation request.

4 Scalable model architecture adaptation

Instead of requiring the user to manually annotate the source models, *Mistify* provides expressive configuration interfaces to specify adaptation goals and constraints (Section 4.1) and suitable abstractions to capture common algorithmic steps that meet these constraints (Section 4.2). To further scale to a large target space, *Mistify* merges adaptation instances to avoid duplicate efforts (Section 4.3) with *collective adaptation*.

4.1 Adaptation goal specification

An adaptation goal reflects the desirable inference performance given static and dynamic device conditions. We assume these goals are immutable, and any changes in the run-time conditions simply generate new goals. A user provides two sets of input: *hardware profiles* and *performance targets*. Hardware profiles mainly include *compute power* (GFLOP/s) and *memory bandwidth* (GB/s). Performance targets cover latency and accuracy requirements. The specification can be extended to support custom resource capability and performance metrics by adding the corresponding profiling libraries and tools (Section 4.2). We leverage a JSON-like format (Figure 3) to specify multiple goals in a single configuration file, which is parsed before adaptation.

We next formulate the *cost budgets* of a given DNN structure based on the specification of the adaptation goal provided by the user. In terms of computation cost, each layer contributes $C_{in} * C_{out} * S_{kernel} * S_{out}$ multiplications and additions. C_{in} and C_{out} denote the input and output channels;

S_{kernel} and S_{out} denote the convolution kernel and output size for Conv operations; for normal Matmul operations, S_{kernel} and S_{out} both equal 1 as they are equivalent to 1×1 convolution on 1×1 inputs. For memory cost, each layer contributes $C_{in} * C_{out} * S_{kernel}$ parameters (S_{kernel} is 1 for Matmul operations similarly). Combined with the quantization strategy, the total memory consumption of a neural network layer can be calculated. For latency cost, we first calculate the previous two costs C_{comp} and C_{mem} respectively. Then, we leverage the hardware specifications (peak computation power and the memory bandwidth) to translate these costs into the latency cost as $C_{mem}/mem\_bandwidth + C_{comp}/comp\_power$.

4.2 Adaptation Executor

Common DNN adaptation workflows. State-of-the-art DNN adaptation algorithms follow a similar process. They take a source DNN model and adaptation goals, and search for variants of the base model architecture that fits each scenario. The search explores a high-dimensional vector space, where each hyperparameter of the DNN (e.g., #layers, #filters, kernel size, and quantization) corresponds to a specific dimension. The search process runs iteratively until the costs of the current model optimally match the adaptation goal. Typical search strategies include evolutionary search [25,77], gradient descent [14,31], and RNN-based search [72,73].

Adaptation executor. In light of this common process, we design an abstraction, an `Adaptation Executor`, that collects all adaptation settings as a closure, and exposes three function APIs (`Init()`, `Measure()`, and `Adjust()`). `Init()` loads the adaptation settings, the model, and the constraints, and then instantiates the executor that runs the chosen adaptation algorithm (default or user specified). `Measure()` is called after each adaptation iteration to determine the *costs* of the current model (e.g., model size or accuracy). Custom metrics and profiling mechanisms can be incorporated by implementing the `Measure()` API. `Adjust()` will then tune the control knobs of the algorithms (e.g., dimension-wise step size, threshold, or learning rate) to steer the cost refinements towards the adaptation goals in an optimal direction. These APIs abstract away the inner workings of heterogeneous adaptation algorithms in a universal approach, obviating the need to directly annotate the models to embed the adaptation logic. A new adaptation algorithm can interface with *Mistify* by implementing the above APIs, and the user can specify the preferred algorithm in the configuration.

Case study: Running MorphNet via an adaptation executor. The vanilla DNN training starts with defining an accuracy loss function ($\mathcal{L}_{output}$) based on the difference between the model outputs and the ground-truth labels. The loss is back-propagated to each layer i (with parameter θ_i) as $\mathcal{L}_i(\theta_i)$. Each layer calculates the gradient of the loss, and optimizes the parameters (θ_i) iteratively by minimizing the loss via gradient descent. Namely, $\theta_i^{new} = \theta_i^{old} - \eta \cdot \nabla_{\theta_i} \mathcal{L}_i(\theta_i)$.

MorphNet (a recent gradient based search algorithm [31]) converts the resource costs of DNNs as additional penalty terms of the loss function. This way, the DNN architecture is iteratively optimized via gradient descent along with the vanilla DNN training. For instance, the "useless" weight parameters will be suppressed to zero and trimmed during training when minimizing the overall loss, as they do not contribute to reducing the accuracy loss but increase the architectural loss. The adaptation process completes when each structure-related cost (e.g., number of FLOPs) satisfies the corresponding constraint, or when the pre-defined maximal running time is reached for the non-converged cases.

Manually adapting a model using MorphNet requires several steps: (i) selecting the penalty term for each operator appearing in the DNN (e.g., the *Gamma* regularizer for `BatchNorm`), (ii) specifying the input and output operators of the model, (iii) instantiating the penalty terms with the right arguments such as the trimming threshold and the learning rate, and adding them to the overall training loss, and (iv) adding the cost monitoring operators and the termination conditions. All these steps are needed for each adaptation target, and require modifying the source code of the DNN definition and training scripts. In contrast, *Mistify* only requires users to specify the high-level configurations (e.g., the adaptation algorithm, the trimming threshold) and adaptation goals (e.g., memory usage, number of FLOPs) in a single JSON file.

To encapsulate the MorphNet algorithm in a *Mistify* adaptation executor, we implement the APIs as follows. For `Init()`, we additionally implement the operations of deriving the positions (e.g., `Conv` layers) to add architectural loss terms, essentially by first finding the input layers, and then traversing the whole DNN graph topologically along the dependencies to insert the loss terms into the corresponding layers until the outputs. `Measure()` simply calculates the resource and performance costs of a given DNN independent of the adaptation algorithms. For `Adjust()`, we implement the logic of setting the learning rate of the loss term corresponding to each resource and/or performance constraint. The implementation of these APIs is lightweight (Section 7).

4.3 Collective adaptation

Multiple adaptation goals often share similar initial steps or training iterations. Handling each adaptation goal independently is very inefficient when deploying the same model to a range of devices. Therefore, *Mistify* provides a mechanism to "merge" adaptation goals to avoid duplicating the same steps. We parse the adaptation goals into an n-ary tree structure following certain rules. Goals along a branch are fulfilled one by one serially in a single pass. We also design a tree traversal mechanism to meet the constraints (e.g., time and space usage) of all goals simultaneously.

Adaptation goals compilation. As mentioned above, each single goal consists of several resource and performance constraints, and can be abstracted as a multi-dimensional

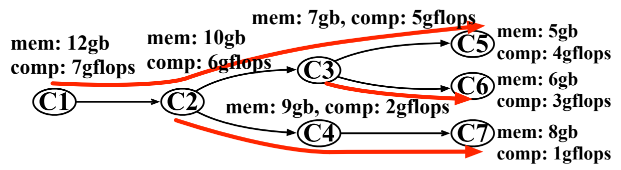


Figure 4: Example adaptation goals parsed into a tree.

vector. Combining the hardware specifications and performance constraints, we generate a partial order of all adaptation goal vectors, from the least demanding to the most. Figure 4 shows an example of 7 goals ($C1$ to $C7$), each with two constraints (memory usage `mem` and computation complexity `comp`). Goals C_i and C_j follow a strict order, $C_i < C_j$, only when $C_i.mem < C_j.mem$ and $C_i.comp < C_j.comp$.

Following the partial order between goals, we further generate a tree structure, with each node representing a goal, and each edge leading to one of its immediate more demanding goals. Hence, each branch of the tree corresponds to an independent adaptation path (marked with a red arrow in Figure 4). Along each path, every two goals are consistently ordered on all constraints. This ensures that they can be collectively adapted in one pass without conflicts. Note that the accuracy does not strictly increase over the path. When *Mistify* starts to traverse a path from one point to the next, the accuracy will first drop to a certain level, and then climb back while the training continues. Meanwhile, the resource profiles will move to the most desirable positions.

Given the tree structure, we first uniformly expand the architecture of the original DNN so that, for each constraint dimension, its actual cost value is larger than that of the root node (the least demanding goal). Then, starting from the root, we run the encapsulated adaptation algorithm to trim the DNN architecture iteratively along each adaptation path. Every time a goal is satisfied, the corresponding version of DNN is stored as a checkpoint for future use.

Note that even though the mapping between partially ordered goals to a tree structure is usually not unique, we find that there is only marginal difference in the overall adaptation time between different mappings. Hence, it is not worth optimizing the mapping given it is NP-hard.

Structure loss scheduling. When executing the adaptation along a path, an essential question is how to control the adaptation towards the optimal direction (via `Adjust()`), i.e., how to meet multiple desirable constraints simultaneously. Although this can be achieved by forking a new adaptation schedule for each change of the adaptation "direction" [69], constant forking does not scale to a large number of adaptation goals and can not achieve fine-grained continuous control.

Instead, we adjust the control knobs based on the weighted combination of the corresponding architecture losses. The overall DNN loss function is the sum of the normal loss ($\mathcal{L}$) and architecture losses corresponding to a set of constraints $\{\mathcal{G}_i\}$. For each $\mathcal{G}_i$, their control knob (e.g., learning rate for gradient-based algorithms) can be viewed as a weight parameter w_i. Hence, the overall loss $\mathcal{L}_{all} = \mathcal{L} + \sum_i w_i \cdot \mathcal{G}_i$. To adjust the adaptation "direction" towards a specific constraint f_i, we only need to increase the weight w_i of the loss $\mathcal{G}_i$.

Initially, all the weights w_i are equal and sum to 1. Suppose for the loss of constraint f_i we have the initial value $\mathcal{G}_i^{(0)}$ and the target value $\mathcal{G}_i^{(+)}$. Then, for every k training iterations (empirically set to 200), we reschedule the weights once. The n-th iteration weight $w_i^{(n)}$ is calculated as $Share_i^{(n)} / \sum_i Share_i^{(n)}$, where $Share_i^{(n)} = \frac{\mathcal{G}_i^{(+)} - \mathcal{G}_i^{(n-1)}}{\mathcal{G}_i^{(+)} - \mathcal{G}_i^{(0)}}$. In essence, we proportionally assign the next value of the weight $w_i^{(n)}$ according to how far the corresponding loss value $\mathcal{G}_i^{(n-1)}$ deviates from the target, and finally normalize these weights.

5 Privacy-aware fine-tuning at the edge

After adjusting the model architecture with respect to the resource and performance constraints, the weight parameters need to be fine-tuned before actual deployment. If all training data are collected and stored in the cloud, parameter tuning simply follows the standard training process for the adapted DNN. The challenge arises when specializing the DNNs only using the local contexts of edge devices.

Recall (Section 2.2) that DNNs are hard to train with a small dataset, usually the case for individual edge device, and can easily overfit. On the other hand, the data local to each device is often more relevant but private, making it difficult or infeasible to aggregate the data from different devices into a larger dataset for centralized training. Therefore we need to balance protecting edge data privacy and ensuring training quality (in terms of how well individual models generalize). While many works (e.g., federated learning [19] and others [20, 52, 70]) address decentralized private DNN training, they assume different endpoints train the *same DNN structure* with different local datasets. The situation is different for *Mistify*, where the models on different devices have *different architectures* to meet specific adaptation goals.

Knowledge distillation (KD). To tackle the aforementioned dilemma, we need a mechanism for DNN "knowledge" sharing between distinct peer models and without explicitly exchanging private data between devices. Fortunately, *mutual knowledge distillation* [15, 81] comes to the rescue. When training a DNN model (M_1, the *student* model) from scratch, leveraging additional help from another similar but independently trained model (M_2, the *peer teacher* model) can significantly improve the validation accuracy of M_1.

Specifically, the optimization of parameter θ_i follows: $\theta_i = \theta_i - \eta \nabla_{\theta_i} \{\phi(y, M_1(x)) + \varphi(M_2(x), M_1(x))\}$, where ∇_θ denotes taking derivatives with respect to the variable θ, ϕ and φ denote the loss functions (e.g., cross-entropy) respectively defined for the ground-truth labels and the teacher model M_2's outputs, and η denotes the learning rate as usual. The corresponding parameter values in M_2 are incorporated as added constraints. This way, the student model receives extra supervision from the teacher model during training, beyond optimizing for conventional learning objectives like the cross-entropy loss subject to the ground-truth training labels.

5.1 Client: KD-enhanced parameter tuning

Observe that a DNN trained locally on an edge device encapsulates the "knowledge" extracted from the private local data. Therefore, to take full advantage of the edge data distributed across devices without exchanging the private data, our algorithm instead shares the DNN models trained independently on each device. The ensemble of DNNs from other devices serves as the "teacher" to guide the current device's model just like in standard mutual knowledge distillation.

Our algorithm proceeds as follows.

(i) Each participating endpoint device (E_i) first tunes their adapted version of DNN model (M_i) with locally available training data until convergence.

(ii) Each endpoint sends its current model along with its loss and accuracy statistics to the central coordinator and waits for a response, namely a set of models (M_1 to M_n) trained on the other devices. An operator is added over the n outputs of these models (M_1 to M_n), taking their average as the final output of the model ensemble.

(iii) KD-enhanced tuning is then invoked to optimize the parameters θ_i of model M_i: $\theta_i = \theta_i - \eta\nabla_{\theta_i}\{\phi(y, M_i(x)) + \varphi(\frac{1}{n-1}\sum_{j\neq i} M_j(x), M_i(x))\}$. Namely, the outputs of each local model M_i are compared with both the ground-truth labels y and the outputs of the assembled teacher model to calculate loss. We follow similar hyperparameter settings as in [37], using cross-entropy loss for ϕ and Kullback-Leibler (KL) divergence [45] for φ to measure the distance between the teacher and local models, and a default value 0.001 for η.

(iv) Now loop over steps (i) to (iii) until the model finally converges. Noticeably, to improve generalization and avoid being skewed by some poor performing models, we randomly skip $max(n/10, 1)$ of the models used for each round of KD-enhanced tuning in step (iii).

Privacy-aware tuning. Although less privacy-sensitive than the training data, DNN models can still leak information from the private training data. To overcome this privacy leak, we can add noise to the fine-tuning process to achieve differential privacy [39, 56]. The noise can be added to either the training data, or the model parameters sent to the *Mistify* server. However, the latter provides less privacy protection, is easier to "denoise", and does not provide fine-grained control easily.

Therefore, we augment the algorithm above with an optional step after (i). Specifically, we add Laplacian noise to the local training data, and train the model (M_i) for additional epochs until convergence. Then, this noisy model (M_i') is sent to the central coordinator in step (ii). This provides differential privacy to the model parameters and reduces information leakage from the private data. The level of noise added is chosen empirically according to existing privacy-preserving machine learning practice (e.g., PATE [56] and Myelin [39]) with the same level of privacy loss preference (e.g., $\varepsilon < 5$).

Mistify is amenable to this differentially private approach by design. As *Mistify* aims to scale to a large batch of end devices (hundreds or more), potentially there is a large number of peer models to draw from during the intermediate steps. Even if the individual noisy intermediate model (M_i') is less accurate than its noiseless counterpart, the accuracy loss is compensated for by the ensemble of other peer models [55].

5.2 Server: Client model coordination

One particular concern of our aforementioned algorithm is whether the models used for KD-enhanced tuning indeed add knowledge rather than noise. Fortunately, our approach is supported by the evidence of correlation between training data and the models. First, datasets from nearby edge devices exhibit spatio-temporal correlation [32, 33, 78]. Second, given training datasets sufficiently similar in their semantic contexts (e.g., types of objects, hidden feature occurrence frequencies), the models thus trained perform semantically equivalent functionality and can provably generalize to achieving the same capability [53, 67, 80].

In practice, we use common spatio-temporal hints (e.g., location, time, view angle) sent by each client along with their models as a coarse-grained mechanism to estimate data correlation. There are myriad alternative lightweight approaches to measure dataset similarity without piece-wise comparison of the actual raw data (e.g., by calculating dataset feature summaries [46, 57]). They are easily pluggable into *Mistify* with the corresponding API implementations. Regardless of the exact metrics used to measure correlation, they are represented as multi-dimensional vectors. The central coordinator on the *Mistify* server maintains a Locality-Sensitive Hashing (LSH [26]) structure to index the vectors for large-scale nearest neighbor lookup at a sublinear complexity [21].

6 Run-time model adaptation

Existing algorithms and libraries only port DNN models statically in a batch mode. Instead, *Mistify* further provides a *streaming* mode, where the client actively monitors runtime changes of the resource and performance constraints and requests new model generation in response to such dynamics.

Constructing a multi-branch model. To support on-the-fly adaptation to fluctuating resource constraints, each DNN model is further constructed in a multi-branch form (Figure 5) during the architecture adaptation process. First, the aforementioned adaptation algorithm is triggered as usual until the constraints specified in the configuration are satisfied. Now, besides continuing to adapt to other configurations, a new adaptation thread is spawned. This thread separately adapts the current DNN into a k-branch DNN. For instance, a 5-branch DNN is built by freezing the first few layers and adapting the remaining layers towards 5 different configurations, whose resource budgets range from $\frac{1}{3}$ of to 3 times that of the original DNN model. The branches share the same base,

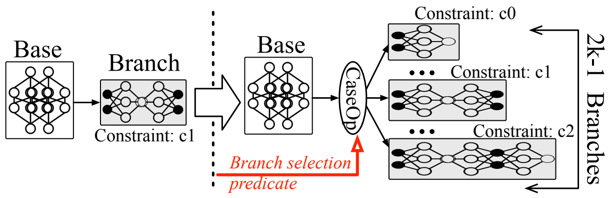


Figure 5: Multi-branch model construction.

achieve the same inference task, but satisfy different resource budgets and performance goals. In practice, *k* is set based on the extent of fluctuation typically observed in the resource availability and performance targets. After fine-tuning the parameters of the multi-branch DNN, we add a case conditional operator (e.g., `tf.case` for Tensorflow) between the base and different branches.

Foreground path: downtime free branch switching. The foreground path is tightly coupled with the user-facing inference serving logic, performing real-time adjustments of the current DNN model based on the dynamic constraints. To achieve this, *Mistify* picks a branch from the multi-branch DNN with the closest resource and performance profiles. The branch switching is done on the fly by setting the corresponding value of the conditional variable (the red arrow in Figure 5) of the case operator in the DNN, avoiding additional overhead such as memory allocation and runtime resource instantiation.

Background path. Meanwhile, in the background, the *Mistify* client will send the new adaptation configuration to the *Mistify* server. The server compares the new configuration with existing ones in terms of their resource constraints, based on the partial ordering explained in Section 4, in order to retrieve the immediate predecessor configuration of this new incoming one. Then, the new DNN model is incrementally adapted from the corresponding "predecessor" DNN, until the constraints of the new configuration are met.

7 Implementation

We implement *Mistify* on TensorFlow (TF) 1.13 [13] (Figure 2), consisting of around 8.5K lines of Python code for both the server and client modules. The source code will be available later at [8].

Interfacing with the native environment. Recall that *Mistify* can be activated at two stages (Section 3), when initializing inference serving or during the run time. For the former, the function `tfhub.load()` is intercepted to trigger the model porting process (when fed the special argument). For the latter case, the *Mistify* runtime monitor is by default registered with the live `Session` of the TensorFlow engine to collect runtime statistics (`tf.RunMetadata`), and invoke the *Mistify* client to initiate the re-adaptation process on demand. The foreground branch switching is implemented by assigning a suitable value to the predicate variable of the `tf.case` op.

Encapsulating adaptation algorithms. *Mistify* implements wrappers over two representative, state-of-the-art adaptation algorithms, MorphNet [31] (using sparsifying regularizers) and ChamNet [25] (using evolutionary algorithms). Adding new adaptation algorithms to *Mistify* is fairly easy, following the process outlined in the MorphNet case study in Section 4.2. Each wrapper implementation around these algorithms for *Mistify* requires around 100 lines of code (LoC), which is fairly modest compared to the thousands of LoC in the original codebases of these algorithms.

8 Evaluation

Hardware setup. Following Figure 2, a Linux server with 8-core 2.1 GHz Intel Xeon CPU, and NVIDIA 2070 GPU acts as the server side of *Mistify*. For the client-side operations of *Mistify*, we use a server with a low-end NVIDIA P600 GPU, a Google Edge TPU [4], and a Samsung S9 smartphone, to represent diverse types of edge hardware.

Application benchmarks. Computer Vision (CV) and Natural Language Processing (NLP) tasks almost dominate deep learning use scenarios. We select one workload each, *Object Recognition* and *Question & Answering* corresponding to the two application categories, as the representative benchmarks. While there are numerous other CV and NLP applications, for example, *scene segmentation* for CV and *machine translation* for NLP, these are based on DNN models derived from the same base structures as those used for our benchmarks (e.g., ResNet blocks for *object recognition* and *detection*, Transformer blocks for *Q&A* and *named entity recognition*). Therefore, the results obtained for our benchmarks are representative of a wide range of scenarios.

Specifically, we select three carefully designed, state-of-the-art DNNs, MobileNet [38], ResNet50 [35], and ResNeXt101 [51], with increasing computation complexity (0.5 to 16 GFLOPs), parameter size (16 to 320 MB), and accuracy (68% to 79%) for *object recognition*. MobileNet is originally designed for mobile devices, whereas the other two mostly run in the cloud. For *Q&A*, the input is a question along with a context paragraph containing the answer to the question. The "accuracy" metric for this is the Exact Match (EM) score, i.e., whether the output answer exactly matches the question. We prepare two DNNs, BiDAF [68], and BERT [28]. The former is lightweight but task-specific (customized for *Q&A*) (10× MB), whereas BERT is much larger, generically supporting various downstream tasks.

Datasets. We use domain specific standard datasets to adapt network architectures, fine-tune their parameters, and validate their performance. Specifically, ImageNet [27] and Cifar100 [44] are used for *object recognition*, whereas SQuADv1.1 [61] is used for *Q&A*.

8.1 Collective architecture adaptation

Collective adaptation time. We generate 128 different adaptation configurations based on four DNNs (MobileNet, ResNet50, ResNeXt101, and BERT). Among these, the least

Table 2: Accuracy - collectively adapted models (*Mistify*) vs individually adapted models (Per-case)

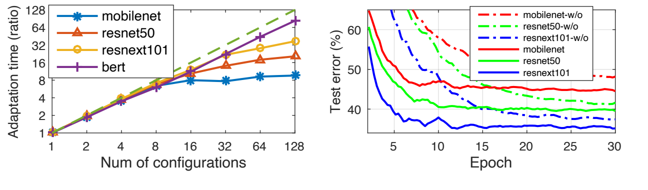
DNN	Per-case (%)	*Mistify* (%)	Relative diff (%)
MobileNet	55.8	54.7	-2.0%
	69.4	69.5	+0.1%
ResNet50	68.2	68.0	-0.3%
	72.9	72.5	-0.6%
ResNeXt101	74.0	74.3	+0.4%
	77.6	77.9	+0.3%
BERT	71.4	70.6	-1.2%
	79.1	78.8	-0.4%

and most demanding configurations respectively constrain the adapted DNNs to 2× and 0.5× the default DNN memory usage and computation complexity. Then, we select different subsets of these 128 configurations, adapt all of them with and without *Mistify*, and compare their overall time consumption to evaluate our collective adaptation approach (Section 4.3). Figure 6(a) shows the relative time needed without *Mistify* over with *Mistify*. *Mistify* accelerates the overall adaptation time almost linearly with the number of configurations when it is less than 10, consistently achieving around 10x acceleration even for DNNs as small as MobileNet. For large DNNs, such as BERT, that are structurally more amenable to adaptation (i.e., easier to prune a subset of the network without affecting validation accuracy), the acceleration scales well with over 100 configurations.

Adaptation quality. Next, we examine the quality of the DNNs collectively adapted by *Mistify* versus those adapted individually. Table 2 shows two rows for each network, corresponding to compression and expansion by a factor 4 with respect to the complexity and memory consumption of the original DNN. This spans the range from low- to high-end hardware [75]. For instance, the inference times of the compressed and expanded ResNet50, running on a Google Nexus 5 (low-end, 2013 model) and a Samsung Galaxy 10 (high-end, 2019 model), are both around 30 ms, low enough for practical usage. "Accuracy" corresponds to the EM score (exactly matching the ground-truth answer) for NLP. To avoid being affected by the parameter tuning quality, all adapted DNNs are trained with the whole datasets, and without considering any device-specific constraints. *Mistify*'s collective adaptation achieves almost the same accuracy compared to the case-by-case strategy, with less than 0.5% accuracy loss for most cases and only 1% for the worst scenario (e.g., when adaptation configurations are incompatible with total ordering, causing the overall adaptation path to detour substantially). These are within the typical range of accuracy loss in exchange for resource efficiency [64].

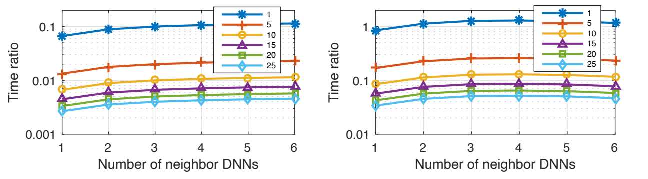
8.2 Parameter tuning

We use a more specialized dataset Cifar100 to evaluate parameter tuning on the edge. The whole dataset is partitioned into subsets, mimicking the local data of each edge device.



(a) DNN apdatation time. (b) Parameter tuning time.

Figure 6: Speed and performance improvements for architecture adaptation and parameter tuning with *Mistify*.



(a) MobileNet (b) BERT

Figure 7: The ratio of communication time over training time, reflecting the scalability of fine-tuning in *Mistify*.

Convergence speed and quality. We compare the convergence speed and test accuracy for three different networks (MobileNet, ResNet50, and ResNeXt101), with and without *Mistify*'s support for parameter tuning (Section 5.1). Figure 6(b) shows that, even without additional data, KD-enhanced parameter tuning (solid lines) already achieves over 3× faster convergence as well as better accuracy.

Scalability. We assess the scalability of the parameter tuning algorithm (Section 5.1) in terms of the ratio of the communication time over the training time given different network bandwidths. We consider two model extremes, MobileNet (very compact) and BERT (very sophisticated). In Figure 7, each line corresponds to a specific network bandwidth in MB/s. When the network bandwidth exceeds 5 MB/s, our algorithm is consistently scalable, with communication merely taking less than 15% of the time relative to training. Further, the lines almost flatten when more than three neighbors' DNNs are used, so using more peer DNNs for our tuning does not impact scalability.

Accuracy of parameter tuning. We randomly partition Cifar100 and SQuAD each into 5 subsets, each used by an edge device for local training. Then, we compare the fine-tuning accuracy using different approaches. Table 3 shows that knowledge distillation (KD) improves parameter tuning accuracy by 40% over local training alone. Compared to the ideal distillation case where an exceptionally accurate teacher network is available (a pre-trained, cloud version), the ensemble of 4 peer DNNs achieves within 10% of the optimal KD, despite using half the training data and adding differential privacy to the model parameters.

8.3 Run-time model re-adaptation overhead

The foreground path. Switching DNNs in response to the run-time dynamics (Section 6) incurs two types of overhead:

Table 3: The accuracy of parameter tuning with *Mistify*.

Scenario	DNNs (%)		
	MobileNet	ResNet50	BERT
Local training	39.7	43.9	22.5
KD	66.4	75.3	78.8
1-peer tuning	53.8	61.5	51.9
2-peer tuning	58.1	67.2	65.6
4-peer tuning	59.8	69.0	71.8

Table 4: Additional number of parameters and DNN switching time overhead.

DNN	Addi. / orig. params (M)	Time (s)
MobileNet-b3	2.67 / 3.43	2.11
MobileNet-b5	4.57 / 3.43	
ResNet50-b3	18.2 / 23.9	3.34
ResNet50-b5	31.7 / 23.9	
BERT-b3	92.4 / 110	21.84
BERT-b5	171 / 110	

i) additional memory to store alternate DNNs; and/or ii) downtime for loading the new DNN and on-demand preparation of the resource runtime. The model size corresponds to the in-memory size, not the on-disk size of the serialized form. Table 4 illustrates the trends of additional memory or time consumption for different DNNs. The suffix "-bk" means adding k branches to the adapted DNN. Holding 75% to 1.5× more parameters in memory is an affordable cost for modern hardware, but can save us 2 to 20 seconds by avoiding loading new DNNs on the critical path of inference serving.

Latency of modifying the configuration tree. We also measure the latency of generating a configuration tree (Section 4.3) given various numbers of configurations. Using 1000 configurations, each consisting of 4 constraints, the whole tree is built in 37 ms, and inserting into an existing configuration tree only takes microseconds.

8.4 End-to-end performance

Based on the device specifications in Table 1 and typical latency requirements for vision and NLP tasks [1, 63], we generate different combinations of memory, complexity, and latency constraints as the execution settings, and evaluate the quality of the DNN models tailored by *Mistify*. We further compare the manual overhead involved in *Mistify* with running MorphNet [31] and ChamNet [25] directly.

Balancing performance and resource usage. We set the memory budget to range from 0.1 GB to 10 GB, covering embedded IoT devices to edge servers. The computation complexity constraints for running inference on a DNN vary between 0.1 to 100× GFLOPs, roughly equivalent to achieving 10s of microseconds of inference latency for resource-constrained devices and powerful edge servers alike.

Figure 8 shows the three-way trade-offs between accuracy, latency, and resource consumption. The top three plots correspond to *recognition*, the lower three to *Q&A*. *Mistify* reduces the compute requirements by over 20× with less than 5% accuracy loss for the CV workloads, and could achieve 50× reduction of complexity in exchange for 12% relative quality-of-result degradation. Note that the accuracy loss is due to the adaptation algorithms, not *Mistify* itself. Similarly, *Mistify* consistently achieves a near-optimal and practically usable accuracy (comparable to existing hand-tuned on-device models in production [25, 49]) with between 0.5 to 10 GB of run-time memory usage, hence significantly decreasing the deployment complexity for state-of-the-art DNN models on the edge. Mapping resource consumption to inference time, *Mistify* consistently achieves near-optimal accuracy performance even when the latency requirements vary by 8 to 10×, corresponding to using accelerator hardware ranging from advanced, datacenter grade to low-power, lightweight devices.

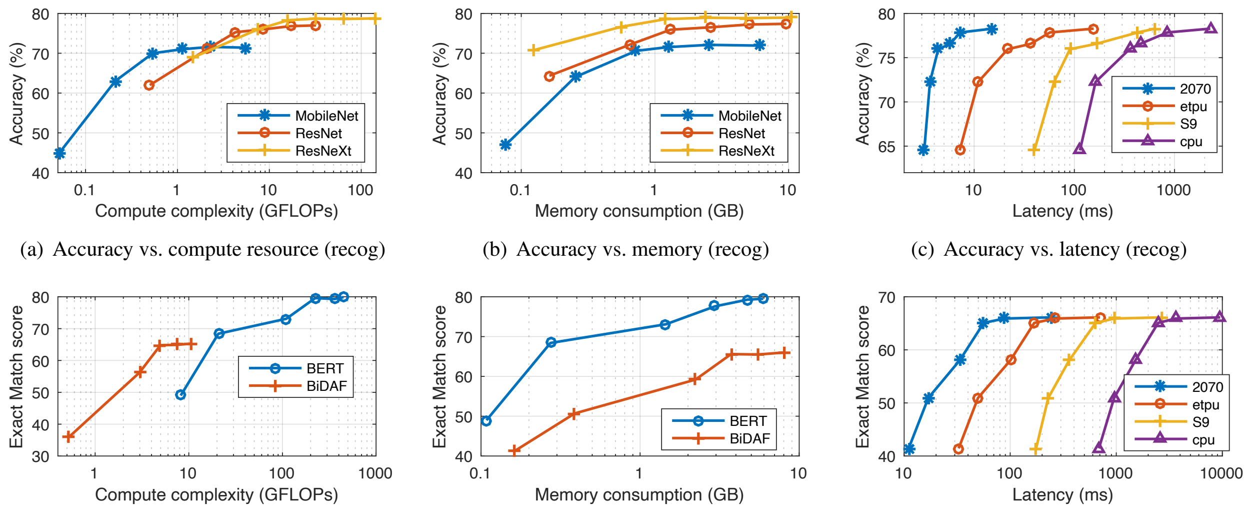
Reducing manual overhead. We further assess the end-to-end manual effort and time overhead needed to port a pre-designed DNN to different edge devices. The manual overhead is quantified with two metrics: lines of code (LoC) needed for code addition or modification, and number of files (NoF) touched. The former depicts the overall overhead, and the latter one captures the scatteredness of the modifications, which correlates with the probability of making mistakes. For NoF, we follow a typical file organization [12], i.e., model definition, training, evaluation, and other stages are separated into different files or folders.

Table 5 demonstrates that *Mistify* reduces the overall modification needed in LoC by 7 to 10×. More importantly, *Mistify* exposes high-level configuration files to users, obviating the need for source script modifications. *Mistify* only requires editing one file. Thus, it can reduce the number of files users need to access by orders of magnitude (over 100×). Finally, *Mistify* can batch-adapt to 100 execution settings using less than 3% of the time needed for the other approaches, highlighting the enormous potential of harnessing the correlation among configurations to optimize the overall porting efficiency.

9 Related work

We are not aware of any prior work that aims at providing an automatic porting *service* bridging DNN design and seamless edge deployment. The most related work revolves around model adaptation and knowledge distillation *algorithms*.

Model adaptation. Production DNN models hand-tuned by experts can run fast and accurately on mobile devices [38, 40, 49, 62]. The essential techniques include quantization, sparsification, and neural block optimization. Recently, Distiller [11], AMC [36], MorphNet [31], OFA [22], ChamNet [25], and many neural architecture search (NAS) works [17, 50, 65, 83, 84] systematically explore the search space for the optimal neural network structure, obviating the hand-tuning by experienced experts. However, none of them is directly usable like *Mistify*, because all are still *algorithms*, requiring manually annotating source code to construct the



(a) Accuracy vs. compute resource (recog) (b) Accuracy vs. memory (recog) (c) Accuracy vs. latency (recog)

(d) Accuracy vs. compute resource (Q&A) (e) Accuracy vs. memory (Q&A) (f) Accuracy vs. latency (Q&A)

Figure 8: The dynamic tradeoff between latency, accuracy, and resource consumption with *Mistify*.

Table 5: Comparison of overhead for porting DNN to edge with/without *Mistify*.

Metrics	2 configurations				10 configurations				100 configurations			
	Manual	Mor.	Chm.	*Mistify*	Manual	Mor.	Chm.	*Mistify*	Manual	Mor.	Chm.	*Mistify*
Lines of Code	>0.2k	55	97	**6**	>1k	138	159	**14**	>10k	782	511	**104**
Num of Files	6	4	5	**1**	30	12	32	**1**	300	102	302	**1**
Total time (%)	100			**54.2**	100			**12.5**	100			**2.86**

adaptation logic, hence not scalable to edge scenarios with multiple adaptation instances. None supports on-device fine-tuning or considers run-time adjustments. *Mistify* is orthogonal as an automated *system framework* and can incorporate them as pluggable algorithmic modules.

While frameworks like TF-Lite [2], PyTorch [58], and MCDNN [34] provide some model *compression* and *switching* support, *Mistify* differs in the techniques supported and the level of manual efforts needed. To generate a good model, careful model architecture design is essential, which normally requires significant expertise. *Mistify* abstracts away the model architecture searching process with the configurable APIs to make it accessible to non-experts, automating the end-to-end process and optimizing for batch model generation.

Knowledge distillation. Initially proposed as an optimization for model training, *knowledge distillation* transfers "knowledge" (i.e., parameter values) from a *teacher* network to a *student* network [37]. The idea is then extended to *mutual distillation* among peer models [15, 47, 81]. *Mistify* adopts and revises the general idea in a *selective distillation* manner to improve edge training accuracy while enhancing privacy.

Edge-centric deep learning inference engines. Emerging frameworks such as TF-Lite [2] and more [7, 29, 54] are optimized for inference serving on mobile and IoT devices, aiming to hide the deployment complexity from developers and device users. However, the interface exposed by existing engines only permits model download from the cloud (or the central server), without tailoring to edge runtime requirements and constraints, proactively or reactively. In contrast, *Mistify* provides an interface for two-way state exchange and a feedback loop between the cloud and the edge, facilitating targeted model design and efficient execution on the edge.

10 Conclusion

Deep learning models today are typically trained in the cloud and then ported to edge devices manually. Not only is manual porting unscalable, it indicates a lack of separation between model design (optimized for accuracy) and deployment (optimized for resource efficiency).

In this paper, we design and implement *Mistify*, a framework to automate this porting process, which reduces the DNN porting time needed to cater to a wide spectrum of edge deployment scenarios by over 10×, incurring orders of magnitude less manual effort. *Mistify* not only provides a useful service to complete the transition from DL workload design to deployment on the edge, but cleanly separates these two stages. We believe the system will further facilitate advanced model design and seamless model deployment.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Eric Schkufza, for their insightful comments. Julia McClellan helped with early exploration of the work. This work is supported by the National Science Foundation under Grant No. CNS-1815115 and a Google Faculty Research Award.

References

[1] Aetina: dedicated AI computing solution at edge. https://www.aetina.com/index.php.

[2] Deploy machine learning models on mobile and IoT devices. https://www.tensorflow.org/lite.

[3] Driving intelligent retail with AI. https://www.nvidia.com/en-us/industries/retail/.

[4] Edge tpu: Google's purpose-built asic designed to run inference at the edge. https://cloud.google.com/edge-tpu.

[5] Foghorn: The Original Edge-Native AI Platform. https://www.foghorn.io/edge-ai-platform/.

[6] Gartner highlights 10 uses for ai-powered devices. https://www.gartner.com/en/newsroom/press-releases/2018-03-20-10-uses-for-ai-powered-devices.

[7] Integrate machine learning models into your app. https://developer.apple.com/documentation/coreml.

[8] Mistify github repo. https://github.com/PatrickGuo/Mistify.

[9] MobiSys'19 IoT Day. https://www.sigmobile.org/mobisys/2019/iot_day_program/.

[10] MorphNet: a library of learning deep network structure during training. https://github.com/google-research/morph-net.

[11] Neural Network Distiller by Intel AI Lab: a Python package for neural network compression research. https://github.com/NervanaSystems/distiller.

[12] TensorFlow Official Models. https://github.com/tensorflow/models.

[13] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[14] J. M. Alvarez and M. Salzmann. Learning the number of neurons in deep networks. In *Advances in Neural Information Processing Systems*, pages 2270–2278, 2016.

[15] R. Anil, G. Pereyra, A. Passos, R. Ormandi, G. E. Dahl, and G. E. Hinton. Large scale distributed neural network training through online distillation. *arXiv preprint arXiv:1804.03235*, 2018.

[16] Avnet. Ai at the edge: The next frontier of the internet of things, 2018.

[17] B. Baker, O. Gupta, N. Naik, and R. Raskar. Designing neural network architectures using reinforcement learning. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[18] S. Bianco, R. Cadene, L. Celona, and P. Napoletano. Benchmark analysis of representative deep neural network architectures. *IEEE Access*, 6:64270–64277, 2018.

[19] K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. Kiddon, J. Konecny, S. Mazzocchi, H. B. McMahan, et al. Towards federated learning at scale: System design. *arXiv preprint arXiv:1902.01046*, 2019.

[20] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth. Practical secure aggregation for privacy-preserving machine learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1175–1191, 2017.

[21] J. Buhler. Efficient large-scale sequence comparison by locality-sensitive hashing. *Bioinformatics*, 17(5):419–428, 2001.

[22] H. Cai, C. Gan, and S. Han. Once for all: Train one network and specialize it for efficient deployment. *arXiv preprint arXiv:1908.09791*, 2019.

[23] N. Chen, Y. Chen, Y. You, H. Ling, P. Liang, and R. Zimmermann. Dynamic urban surveillance video stream processing using fog computing. In *2016 IEEE second international conference on multimedia big data (BigMM)*, pages 105–112. IEEE, 2016.

[24] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, et al. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 578–594, 2018.

[25] X. Dai, P. Zhang, B. Wu, H. Yin, F. Sun, Y. Wang, M. Dukhan, Y. Hu, Y. Wu, Y. Jia, et al. Chamnet: Towards efficient network design through platform-aware model adaptation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 11398–11407, 2019.

[26] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262. ACM, 2004.

[27] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[28] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, 2019.

[29] M. Dukhan, Y. Wu, and H. Lu. Qnnpack: open source library for optimized mobile deep learning.

[30] D. Evans. The internet of things: How the next evolution of the internet is changing everything. *CISCO white paper*, 1(2011):1–11, 2011.

[31] A. Gordon, E. Eban, O. Nachum, B. Chen, H. Wu, T.-J. Yang, and E. Choi. Morphnet: Fast & simple resource-constrained structure learning of deep networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1586–1595, 2018.

[32] P. Guo, B. Hu, R. Li, and W. Hu. Foggycache: Cross-device approximate computation reuse. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, pages 19–34, 2018.

[33] P. Guo and W. Hu. Potluck: Cross-application approximate deduplication for computation-intensive mobile applications. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 271–284, 2018.

[34] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy. Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, pages 123–136, 2016.

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

[36] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han. Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 784–800, 2018.

[37] G. Hinton, O. Vinyals, and J. Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[38] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[39] N. Hynes, R. Cheng, and D. Song. Efficient deep learning on multi-source private data. *CoRR*, abs/1807.06689, 2018.

[40] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[41] S. Jain, G. Ananthanarayanan, J. Jiang, Y. Shu, and J. Gonzalez. Scaling video analytics systems to large camera deployments. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*, pages 9–14. ACM, 2019.

[42] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia. Noscope: optimizing neural network queries over video at scale. *Proceedings of the VLDB Endowment*, 10(11):1586–1597, 2017.

[43] N. V. Kim and M. A. Chervonenkis. Situation control of unmanned aerial vehicles for road traffic monitoring. *Modern Applied Science*, 9(5):1, 2015.

[44] A. Krizhevsky, V. Nair, and G. Hinton. Cifar-10 and cifar-100 datasets. *URl: https://www. cs. toronto. edu/kriz/cifar. html*, 6, 2009.

[45] S. Kullback and R. A. Leibler. On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86, 1951.

[46] A. B. L. Larsen, S. K. Sønderby, H. Larochelle, and O. Winther. Autoencoding beyond pixels using a learned similarity metric. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, pages 1558–1566. JMLR.org, 2016.

[47] T. Li, J. Li, Z. Liu, and C. Zhang. Knowledge distillation from few samples. *arXiv preprint arXiv:1812.01839*, 2018.

[48] R. LiKamWa, Y. Hou, J. Gao, M. Polansky, and L. Zhong. Redeye: analog convnet image sensor architecture for continuous mobile vision. In *ACM SIGARCH Computer Architecture News*, volume 44, pages 255–266. IEEE Press, 2016.

[49] C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy. Progressive neural architecture search. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 19–34, 2018.

[50] H. Liu, K. Simonyan, and Y. Yang. DARTS: differentiable architecture search. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[51] D. Mahajan, R. Girshick, V. Ramanathan, K. He, M. Paluri, Y. Li, A. Bharambe, and L. van der Maaten. Exploring the limits of weakly supervised pretraining. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 181–196, 2018.

[52] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Artificial Intelligence and Statistics*, pages 1273–1282. PMLR, 2017.

[53] M. Mirman, T. Gehr, and M. Vechev. Differentiable abstract interpretation for provably robust neural networks. In *International Conference on Machine Learning*, pages 3578–3586, 2018.

[54] S. Mittal. A survey on optimized implementation of deep learning models on the nvidia jetson platform. *Journal of Systems Architecture*, 2019.

[55] N. Papernot, M. Abadi, Ú. Erlingsson, I. J. Goodfellow, and K. Talwar. Semi-supervised knowledge transfer for deep learning from private training data. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[56] N. Papernot, S. Song, I. Mironov, A. Raghunathan, K. Talwar, and Úlfar Erlingsson. Scalable private learning with pate. In *ICLR*, 2018.

[57] S. Parthasarathy and M. Ogihara. Exploiting dataset similarity for distributed mining. In *International Parallel and Distributed Processing Symposium*, pages 399–406. Springer, 2000.

[58] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pages 8026–8037, 2019.

[59] V. Popov, M. Kudinov, I. Piontkovskaya, P. Vytovtov, and A. Nevidomsky. Distributed fine-tuning of language models on private data. 2018.

[60] W. Qualcomm. We are making on-device ai ubiquitous, 2018.

[61] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang. Squad: 100,000+ questions for machine comprehension of text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2383–2392, 2016.

[62] S. Ravi. Projectionnet: Learning efficient on-device deep networks using neural projections. *arXiv preprint arXiv:1708.00630*, 2017.

[63] S. Ravi and Z. Kozareva. Self-governing neural networks for on-device short text classification. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 887–893, 2018.

[64] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 267–278. IEEE, 2016.

[65] E. Real, C. Liang, D. R. So, and Q. V. Le. Automl-zero: Evolving machine learning algorithms from scratch. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 8007–8019. PMLR, 2020.

[66] J. Redmon and A. Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[67] A. Ruoss, M. Baader, M. Balunović, and M. Vechev. Efficient certification of spatial robustness. *arXiv preprint arXiv:2009.09318*, 2020.

[68] M. Seo, A. Kembhavi, A. Farhadi, and H. Hajishirzi. Bidirectional attention flow for machine comprehension. *arXiv preprint arXiv:1611.01603*, 2016.

[69] A. Shin, D. Y. Kim, J. S. Jeong, and B.-G. Chun. Hippo: Taming hyper-parameter optimization of deep learning with stage trees. *arXiv preprint arXiv:2006.11972*, 2020.

[70] R. Shokri and V. Shmatikov. Privacy-preserving deep learning. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, pages 1310–1321, 2015.

[71] E. Strubell, A. Ganesh, and A. McCallum. Energy and policy considerations for deep learning in nlp. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 3645–3650, 2019.

[72] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2820–2828, 2019.

[73] M. Tan and Q. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pages 6105–6114, 2019.

[74] J. Wang, J. Pan, and F. Esposito. Elastic urban video surveillance system using edge computing. In *Proceedings of the Workshop on Smart Internet of Things*, page 7. ACM, 2017.

[75] C.-J. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia, et al. Machine learning at facebook: Understanding inference at the edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 331–344. IEEE, 2019.

[76] C. Xia, J. Zhao, H. Cui, X. Feng, and J. Xue. Dnntune: Automatic benchmarking dnn models for mobile-cloud computing. *ACM Transactions on Architecture and Code Optimization (TACO)*, 16(4):1–26, 2019.

[77] Y. Xiong, R. Mehta, and V. Singh. Resource constrained neural network architecture search: Will a submodularity assumption help? In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1901–1910, 2019.

[78] M. Xu, M. Zhu, Y. Liu, F. X. Lin, and X. Liu. Deepcache: Principled cache for mobile deep vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, pages 129–144, 2018.

[79] J. Yim, D. Joo, J. Bae, and J. Kim. A gift from knowledge distillation: Fast optimization, network minimization and transfer learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4133–4141, 2017.

[80] J. Zhang, Y. Wang, P. Molino, L. Li, and D. S. Ebert. Manifold: A model-agnostic framework for interpretation and diagnosis of machine learning models. *IEEE transactions on visualization and computer graphics*, 25(1):364–373, 2018.

[81] Y. Zhang, T. Xiang, T. M. Hospedales, and H. Lu. Deep mutual learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4320–4328, 2018.

[82] B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. 2017.

[83] B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[84] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. Learning transferable architectures for scalable image recognition. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pages 8697–8710. IEEE Computer Society, 2018.

Elastic Resource Sharing for Distributed Deep Learning

Changho Hwang, KAIST
Taehyun Kim, KAIST
Sunghyun Kim, MIT
Jinwoo Shin, KAIST
KyoungSoo Park, KAIST

Abstract

Resource allocation and scheduling strategies for deep learning training (DLT) jobs have a critical impact on their average job completion time (JCT). Unfortunately, traditional algorithms such as Shortest-Remaining-Time-First (SRTF) often perform poorly for DLT jobs. This is because blindly prioritizing only the short jobs is suboptimal and job-level resource preemption is too coarse-grained for effective mitigation of head-of-line blocking.

We investigate the algorithms that accelerate DLT jobs. Our analysis finds that (1) resource efficiency often matters more than short job prioritization and (2) applying greedy algorithms to existing jobs inflates average JCT due to overly optimistic views toward future resource availability. Inspired by these findings, we propose Apathetic Future Share (AFS) that balances resource efficiency and short job prioritization while curbing unrealistic optimism in resource allocation. To bring the algorithmic benefits into practice, we also build CoDDL, a DLT system framework that transparently handles automatic job parallelization and efficiently performs frequent share re-adjustments. Our evaluation shows that AFS outperforms Themis, SRTF, and Tiresias-L in terms of average JCT by up to 2.2x, 2.7x, and 3.1x, respectively.

1 Introduction

Deep learning has become a key technology that drives intelligent services based on machine learning such as face recognition [51,56,57], voice assistant [28,29,46], self-driving cars [11, 16, 40], and medical image processing [35, 43]. Practical deep learning models often require training with a large number of samples for high accuracy, so it is common practice to accelerate deep learning training (DLT) with parallel execution using multiple GPUs. Thus, today's GPU cluster for DLT accommodates running multiple distributed DLT jobs that multiplex the shared GPUs.

Minimizing the average job completion time (JCT) is often a desirable goal, but existing approaches are ill-suited for two reasons. First, existing DLT schedulers [24,34] tend to prioritize only "early-finishing" jobs, but we observe that prioritizing "late-finishing but scalable" jobs is key to average JCT reduction. Algorithms such as Shortest-Remaining-Time-First (SRTF) [44] that prioritize early-finishing jobs have been shown optimal when the throughput scales linearly with the allocated GPUs [48]. However, this does not apply to DLT jobs in general as evidenced by the fact that even a simple fairness scheme such as Max-Min [20] often achieves better average JCT than SRTF. Second, existing algorithms are typically coupled with non-elastic resource allocation [24, 55], which always allocates the requested number of resources to each job and rarely regulates it. This is inherently inefficient as it disallows share re-adjustment at runtime even when some GPUs are underutilized or idle. Also, job-level resource preemption is often too coarse-grained for effective mitigation of head-of-line (HOL) blocking.

In this work, we investigate scheduling algorithms that accelerate DLT jobs. Our rigorous analysis and experiments with real-world DLT traces reveal the following. First, it is critical to consider both resource efficiency and short job prioritization for average JCT reduction. This is because real-world DLT workloads typically include relatively short jobs whose throughput scales poorly as well as highly-scalable jobs that run longer than others. In such scenarios, it is detrimental to prioritize only short jobs as it would reduce the aggregate resource efficiency that ends up with average JCT blowup. Second, designing an optimal algorithm is infeasible as the optimal *past* decisions would highly depend on the *future* jobs. In fact, repeatedly applying a greedy algorithm leads to grossly poor behavior due to its overly optimistic view toward future resource availability.

Incorporating the above findings, we draw a rule-of-thumb of elastic resource sharing for average JCT reduction: more resources to efficient jobs if the resource contention is heavy in the future, otherwise more resources to short jobs. This indicates that the scheduler should proactively prepare for the contention in the future by utilizing current resources more efficiently (see Section 3.2 for details). However, the future is typically unknown. Thus, of other possible ways to achieve it, we propose a scheduling algorithm which assumes that the future will be similar to the past, hence the name Apathetic Future Share (AFS). By assuming that the future load still exists, and that the level of it will be similar to the past, we do not rush into overly biasing to either efficiency or shortness. Instead, we gracefully adapt to the change of contention by re-adjusting shares of all jobs at each churn event. Our evaluation shows that this heuristic is highly effective in real-world DLT traces. AFS outperforms existing algorithms like Tiresias [24] and Themis [34] by 1.9x to 3.1x and 1.2x to 2.2x in average JCT reduction.

To deliver the algorithmic benefits to the real world, we also implement CoDDL, a DLT system framework that efficiently supports elastic share adaptation. Users of CoDDL simply submit a model based on a single GPU, and the system transparently parallelizes it with an arbitrary GPU share determined by the scheduler. The system handles frequent share re-adjustments efficiently via fast cancellation of outdated in-flight re-adjustment commands, which avoids potential thrashing on a burst of reconfigurations. It also overlaps job execution and slow initialization of a newly-allocated GPU, which effectively minimizes the idle time of GPUs during reconfiguration.

Our contribution is three-fold. (1) We show the importance of considering both resource efficiency and short job prioritization for average JCT minimization. We present empirical evidence with real-world traces and an analytical model that considers both to reduce average JCT. (2) We show that handling future jobs requires proactive preparation in current share calculation. We demonstrate that a simple heuristic like AFS brings significant benefits to average JCT reduction. (3) We design and implement CoDDL, which enables efficient realization of elastic share adaptation.

2 Background and Motivation

We begin by describing the DLT job scheduling problem with our underlying assumptions. We then present the limitations of existing schemes and discuss an alternative as well as a set of new challenges it poses.

2.1 Problem and Challenges

We investigate the problem of scheduling multiple DLT jobs in a GPU cluster. A DLT job trains a deep neural network (DNN) that can utilize multiple GPUs for parallel execution. We assume that neither the arrivals of future jobs nor their lengths (training durations) are known to the GPU cluster.[1] We seek to develop efficient algorithms and systems support to enhance overall cluster performance: minimize average JCT, enhance cluster efficiency, and alleviate job starvation.

[1]For the sake of presentation, we consider the case where job lengths are available to the GPU cluster in earlier parts of the paper.

Algorithms	Prioritize short job	Prioritize efficient job	Elastic sharing
SRTF [44]	✓		
Max-Min [20]		△	✓
Optimus [38]		✓	✓
SRSF, Tiresias [24]	✓	△	
Themis [34]	✓	△	✓
AFS	✓	✓	✓

Table 1: Comparison of the algorithmic gain with existing algorithms. △ indicates that it is handled implicitly.

Our approach to improving cluster performance is to design a job scheduler that leverages *elastic resource sharing* among DLT jobs. As opposed to non-elastic sharing [24, 55], which only selects the jobs to run and allocates the requested number of resources (job-level coarse-grained scheduling), elastic sharing decides how many resources to allocate to each job and regulates it during runtime to better achieve the performance goal (resource-level fine-grained scheduling). This approach opens up the possibility of further optimization, but it is feasible only when jobs in the workload can adapt to their changing resource usage with a high degree of freedom. DLT jobs nicely fit into the category; they can scale-in/out automatically [1, 4, 45] to utilize more or fewer GPUs, as they have a common scale-out pattern for data-parallel training [12, 13]. In a greedy multi-tenant environment, this implies that DLT jobs always want more GPUs (as long as it improves training throughput), thus the scheduler should focus on better distribution of GPUs across jobs rather than sticking to a fixed (or non-elastic) amount for each job.

Even when jobs can freely change their resource usage, elastic sharing is not always effective for average JCT reduction. As regulating the resources of a running job incurs an overhead, even migrating a job to avoid resource fragmentation produces little gain unless the job's runtime is long enough. Also, the fine-grained scheduling via elastic sharing does not provide any extra benefit for average JCT reduction if the job throughput scales linearly to the given resources, where SRTF is proven to be optimal [44, 48]. However, we find that the DLT workload is one example that could benefit from elastic sharing. DLT jobs typically run for a long time in the cluster – the real-world workloads we use (see Section 5.1) show up to 2.8 *days* of average JCT, while a typical big-data job completes within 30 minutes [19, 26, 39]. Also, the DLT job throughput is known to scale *sublinearly* to the number of allocated GPUs due to inter-GPU communication overhead for parameter updates.

Putting elastic resource sharing into practice poses a new set of challenges on two fronts. On the algorithms side, we find that minimizing average JCT of sublinearly-scaling jobs requires fine-grained resource preemption that simultaneously considers job lengths and resource efficiency. Existing DLT schedulers either adopt non-elastic sharing that only conducts a coarse-grained *job-level* resource preemption [24, 55], or handles preemption in a less aggressive

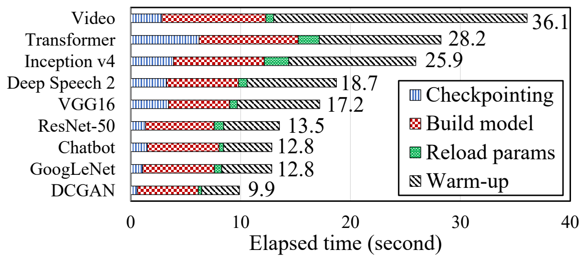


Figure 1: Breakdowns of DLT restart overhead in TensorFlow. Details of models are found in Table 4. Warm-up indicates the execution overhead seen at the first training iteration.

manner to reduce average JCT [34, 38] (see Table 1). We investigate these challenges in greater detail in Section 3.

On the systems side, we find that the overhead of existing DLT auto-scaling [1, 4, 45] and inter-GPU communication APIs [2, 3] is too large to support elastic-share schedulers because they require a complete restart of a DLT job when it updates the resource share. A full restart of a DLT job often takes tens of seconds (see Figure 1), and elastic sharing would only aggravate the situation as it tends to incur more frequent share re-adjustments. To bring the algorithmic benefit to real-world DLT jobs, we build the CoDDL system to address these challenges. We investigate these challenges in greater detail in Section 4.

2.2 Resource Efficiency Matters

Before we present our results in the following sections, let us look deeper into one central concept: resource efficiency. Cluster resource schedulers are often coupled with job preemption to curb HOL blocking that inflates average JCT. For linearly-scaling jobs, the preemption needs to prioritize short jobs (i.e. SRTF), which can be achieved with job-level resource preemption alone. In this case, resource efficiency[2] is a non-issue as all jobs have the identical efficiency gain for the same amount of extra resources. However, sublinearly-scaling jobs (e.g., DLT jobs) tend to have a different efficiency gain, so fine-grained balancing is required in prioritizing either short or efficient jobs for HOL blocking mitigation. Failing to consider both leads to suboptimal behavior.

Let us demonstrate that the existing algorithms fail to explicitly consider either efficiency or short job prioritization, thus exhibit inconsistent performance across two distinct workload scenarios. We run two non-elastic sharing algorithms (SRTF and Shortest-Remaining-Service-First (SRSF) [24]) and two elastic sharing ones (Max-Min and AFS-L) using two traces.[3] SRSF is similar to SRTF in that it performs job-level resource preemption, but differs in that it prioritizes jobs with a smaller remaining *service* time (i.e., a product of remaining time and the number of requested

[2]We refer to resource efficiency of a GPU as the marginal throughput in ratio that it contributes to its job, which drops as a job is given more GPUs.

[3]Except for Max-Min, all other algorithms require knowledge on job lengths in this particular experiment. Max-Min is at a disadvantage.

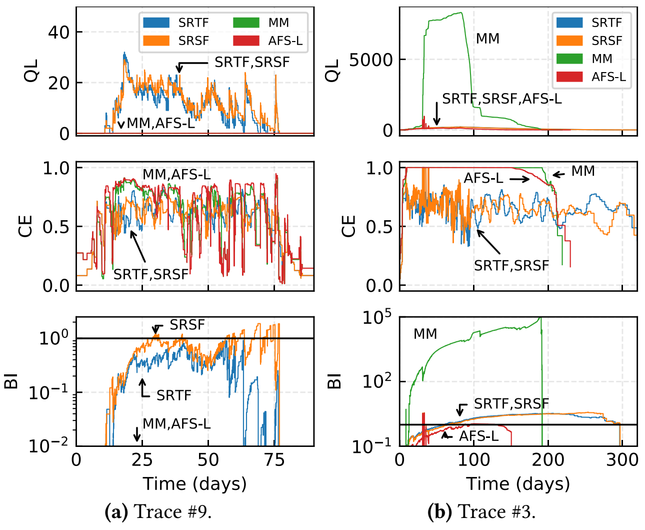


Figure 2: QL, CE, and BI (log-scale) during runtime for two traces in Table 3. MM is Max-Min. Note the different scale of Y-axis in (a) and (b). Experiment setup is in Section 5.1.

resources), in a way to penalize a job that requires many resources. Max-Min distributes available resources across all jobs in max-min fairness, hence performs finer-grained resource preemption than SRTF and SRSF. AFS-L is our algorithm whose details are deferred to Section 3. It is important to note that (1) none of the existing algorithms (except AFS-L) is the best in both traces, and more importantly, (2) Max-Min outperforms both SRTF and SRSF in some cases (5 out of 11 traces in Table 3 in Appendix).

To make our discussion clear, we define three metrics:

- *Queue Length (QL)*: number of pending jobs (yet to be allocated resources) in the cluster. This metric roughly measures the busyness of the cluster.
- *Cluster Efficiency (CE)*: aggregate resource efficiency of the cluster. A larger CE leads to a smaller makespan (elapsed time to complete all jobs). Let us denote by J_R the set of running jobs on an M-GPU cluster. Then,

$$CE := \frac{1}{M}\sum_{j\in J_R}\frac{\text{Current Throughput of } j}{\text{Throughput of } j \text{ with 1 GPU}}.$$

- *Blocking Index (BI)*: average fractional pending time to remaining time of pending jobs, i.e. it increases as pending jobs that can finish shortly pend for a long time. Let us denote by J_P as the set of pending jobs on an M-GPU cluster. Then,

$$BI := \frac{1}{|J_P|}\sum_{j\in J_P}\frac{\text{Pending Time of } j}{\text{Remaining Time of } j \text{ with 1 GPU}}.$$

Figure 2 illustrates two distinct workload scenarios with the above three metrics. Figure 2a presents moderate contention with a relatively few jobs that are submitted to the cluster, whereas Figure 2b shows heavy contention with a relatively large number of jobs.

Moderate contention. The average JCTs for SRTF, SRSF, Max-Min, and AFS-L are 31.1, 32.8, 18.3, and 15.2 hours, respectively. Note that Max-Min achieves lower JCT than SRTF and SRSF. This case is where job-level resource preemption, being too coarse-grained, results in unnecessary job blocking, and in turn poor JCT performance. To see this, let us consider Max-Min. As the number of submitted jobs is relatively small, Max-Min enables the cluster to accommodate nearly all jobs concurrently with *individual resource-level* preemption. Max-Min does not explicitly aim to deal with job blocking to reduce average JCT, but its behavior results in effective mitigation (or even elimination) of it.

On the other hand, SRTF and SRSF impose restrictions on the number of GPUs each job can utilize (either requested amount or nothing) with job-level resource preemption. Due to a small pool of jobs, SRTF and SRSF find it hard to select a good combination of jobs to utilize GPUs. In Figure 2a, Max-Min shows better QL and BI than SRTF and SRSF as it accommodates all jobs in contrast to SRTF and SRSF.

Heavy contention. The average JCTs for SRTF, SRSF, Max-Min, and AFS-L are 3.53, 3.32, 63.20, 2.40 days, respectively. SRTF and SRSF achieve lower JCT than Max-Min in this case as failing to prioritize short jobs aggravates job blocking, which in turn leads to poor JCT. To see this, let us consider SRTF and SRSF. Unlike the previous case, the number of submitted jobs is relatively large, so it is unavoidable to leave some jobs to starve. SRTF and SRSF mitigate job blocking to curb it. As the pool of jobs is diverse in terms of the requested number of GPUs due to its large size, SRTF and SRSF find it easy to select a good combination of jobs to do so.

On the other hand, Max-Min aims to maximize the fairness across submitted jobs rather than to explicitly prioritize short jobs. As a result, it leaves many (and a growing number of) short jobs under-served, causing severe job blocking. In Figure 2b, QL and BI show that Max-Min cannot deal with a large number of submitted jobs while SRTF and SRSF are good at prioritizing short jobs and thus able to keep it at bay.

3 Scheduling Algorithm

Our elastic resource sharing scheme strives to balance prioritizing between short and efficient jobs. At first glance, finding the optimal allocation strategy (possibly, by evaluating all possible allocation candidates) is an NP-complete problem [17]. Also, future job arrivals (which are unknown at the time of scheduling) can wreak havoc on previous resource allocation decisions that would have been optimal otherwise. Instead, we first gain insight through rigorous analysis on a simplified problem, and then factor in practical concerns of the original problem.

3.1 Overview

Problem. We formally define the DLT scheduling problem as follows. Let n denote the total number of jobs including all unknown future jobs. Each DLT job j_k ($1 \le k \le n$) is submitted at an arbitrary time to a cluster with a fixed number of GPUs (M) where each job trains a DL model in a bulk-synchronous-parallel (BSP) [12, 13] fashion.[4] Every job in its lifetime incurs two scheduling events (i.e., arrival and completion)[5] under which the scheduler re-adjusts the GPU shares of all jobs to minimize average JCT. Specifically, it strives to find the optimal value of n-dimensional vector $R_u = \{r_{1,u}, r_{2,u}, \cdots, r_{n,u}\}$, where $r_{k,u}$ is the number of GPUs allocated to j_k after the u-th event (either arrival or completion, $1 \le u \le 2n$).[6] For simplicity, we assume all GPUs have the identical computing/memory capacity that is accessed with the identical network latency.

Approach. As aforementioned, one cannot find the optimal R_u without knowledge on the future jobs.[7] Thus, our goal is to find a clever heuristic that helps improve overall JCT. Our high-level intuition is that repeatedly applying greedy optimization to existing jobs will be "overly optimistic" in the future when a new job arrives, as it rests on the greedy assumption that all resources released by finishing jobs will be used solely by the existing jobs. This implies that the scheduler assumes all active jobs in the cluster will have a non-decreasing number of resources at every scheduling event (i.e. $r_{k,u} \le r_{k,u+1}$), which is far from reality except when there is no more job in the future.

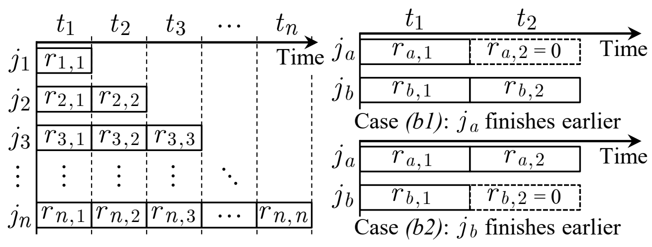
Key assumption. We propose the Apathetic Future Share (AFS) assumption, which predicts that the resource usage of each job (except the finishing one) would be the same in the future, and find the optimized shares based on that. This is not only simple but it also closely approximates the real cluster environment where the level of resource contention does not change dramatically during most of its runtime.

Organization. In what follows, we explain AFS in detail and discuss its corner cases. First, in a two-job case with the knowledge on their job length, we gain insight by presenting a greedy optimization. Next, we extend it to an n-job case without the knowledge on the job length by incorporating the AFS assumption.

3.2 Insight from Two-Job Analysis

Time slot-based analysis. Let us consider a problem with n DLT jobs submitted to an M-GPU cluster all at once in the beginning. Assume that we know the optimal algorithm to allocate the GPUs and schedule the jobs, and that Figure 3a shows the allocation result over time. The jobs are listed in the order of completion where j_1 finishes first and j_n last. t_k ($k > 1$) represents a time slot k which denotes the time interval between the completions of j_{k-1} and j_k; t_1 represents the interval from the beginning to the completion of j_1. j_k is allocated $r_{k,t}$ GPUs at time slot t. Its share is released and re-distributed to other jobs at its completion.

[4]Discussion on asynchronous training [32, 42] is found in Appendix D.
[5]We omit other kinds of events (e.g. eviction timeout) for brevity.
[6]Completed or not arrived jobs are allocated zero GPU.
[7]Several simple examples are shown in Appendix A.


t_1 t_2 t_3 $\cdots$ t_n
Time
j_1 $r_{1,1}$
j_2 $r_{2,1}$ $r_{2,2}$
j_3 $r_{3,1}$ $r_{3,2}$ $r_{3,3}$
j_n $r_{n,1}$ $r_{n,2}$ $r_{n,3}$ $\cdots$ $r_{n,n}$
j_a $r_{a,1}$ $r_{a,2}=0$
j_b $r_{b,1}$ $r_{b,2}$
Case (b1): j_a finishes earlier
j_a $r_{a,1}$ $r_{a,2}$
j_b $r_{b,1}$ $r_{b,2}=0$
Case (b2): j_b finishes earlier

(a) Time slots of n jobs. **(b)** Time slots of 2 jobs.

Figure 3: Example timelines of jobs.

Optimal allocation for two jobs. Let us consider the simplest scenario in which we have one GPU and two jobs (j_a and j_b). We need to allocate the GPU to the shorter job first. Thus, we compute the lengths of both jobs to obtain the solution, which Figure 3b illustrates: if j_a turns out to be shorter, $(r_{a,1}, r_{a,2}, r_{b,1}, r_{b,2}) = (1,0,0,1)$; otherwise, $(r_{a,1}, r_{a,2}, r_{b,1}, r_{b,2}) = (0,1,1,0)$.

To extend the case to $M > 1$, without loss of generality, let us assume j_a finishes earlier than j_b. Suppose $M-1$ GPUs have been optimally allocated for t_1 and t_2. How should one determine which job to allocate the extra M-th GPU to? More concretely, let us denote by w_k (k is either a or b) the total training iterations required for j_k to complete and by $p_{k,u}$ the throughput of j_k with $r_{k,u}$ GPUs at time slot u. One can express t_1 and t_2:

$$t_1 = \frac{w_a}{p_{a,1}}, \quad t_2 = \frac{w_b - p_{b,1}t_1}{p_{b,2}}.$$

We have two cases to consider. (a) j_a earns the GPU first. In this case, $r_{a,1}$ and $r_{b,2}$ increase by one (hence $p_{a,1}$ and $p_{b,2}$ will increase accordingly). (b) j_b earns the GPU first. In this case, two possibilities arise depending on which job finishes first as shown in Figure 3b:

Case (b1): j_a finishes earlier (i.e., $\frac{w_a}{p_{a,1}} < \frac{w_b}{p'_{b,1}}$, where $p'_{k,u}$ is the throughput of j_k with $(r_{k,u}+1)$ GPUs at time slot u). $r_{b,1}$ increases by one and $r_{b,2} = M$.

Case (b2): j_b finishes earlier (i.e., $\frac{w_a}{p_{a,1}} \geq \frac{w_b}{p'_{b,1}}$). $r_{b,1}$ increases by one and $r_{a,2} = M$.

We obtain (1) and (2) by subtracting the average JCTs for cases (b1) and (b2) from the average JCT for case (a), respectively:

$$\frac{w_a}{p'_{a,1}} - \frac{w_a}{p_{a,1}} + \frac{w_a}{2p'_{b,2}}\left(\frac{p'_{b,1}}{p_{a,1}} - \frac{p_{b,1}}{p'_{a,1}}\right), \tag{1}$$

$$\frac{w_a}{p'_{a,1}} - \frac{w_b}{p'_{b,1}} + \frac{p_{b,1}}{2p'_{b,2}}\left(\frac{w_b}{p_{b,1}} - \frac{w_a}{p'_{a,1}}\right) - \frac{p_{a,1}}{2p'_{a,2}}\left(\frac{w_a}{p_{a,1}} - \frac{w_b}{p'_{b,1}}\right). \tag{2}$$

Here, if either (1) or (2) is positive, then one can allocate the extra GPU to j_b first and further minimize average JCT. Otherwise, one should allocate it to j_a first.

For an arbitrary number of GPUs, one needs to repeat the above procedure starting with one GPU. As long as one knows the workload (w_k) and throughput ($p_{k,u}$) information in advance, she can determine the optimal resource allocation for the simple two-job case.

Notation	Description
M	Total # of GPUs in the cluster
n	# of jobs
j_k	Job k
$r_{k,u}$	# of GPUs assigned to j_k after the u-th event
R_u	$\{r_{1,u}, r_{2,u}, \cdots, r_{n,u}\}$
w_k	# of training iterations of j_k
t_u	Length of time slot u
$p_{k,u}$	Throughput of j_k using $r_{k,u}$ GPUs
$p'_{k,u}$	Throughput of j_k using $r_{k,u}+1$ GPUs

Table 2: Description of mathematical notations.

Handling future jobs. We attain valuable insight when we add a third job j_c. For the sake of presentation, let us assume j_c is submitted right after j_a finishes in case (b1).[8] With the new job present, j_b is likely to be allocated fewer GPUs and consequently achieves a lower throughput ($p'_{b,2}$). This could flip the sign of (1) from negative to positive, which would prioritize the longer job (j_b) over the shorter job (j_a). Likewise, a lower throughput of j_a ($p'_{a,2}$) could flip the sign of (2) from positive to negative in case (b2), which would also prioritize the longer job (j_a) over the shorter job (j_b). This example clearly shows that (1) the presence of a future job at time slot 2 may impact the optimal decision at time slot 1, which demonstrates the *infeasibility of an optimal resource allocation*, and (2) if future jobs increase resource contention, it is often beneficial to allocate *more GPUs to longer but efficient jobs*, which is in contrast with SRTF.[9] This implies that we can prepare for future contention by assigning more resources to efficient jobs, which would be difficult for greedy optimization to achieve as it does not consider future job arrivals.

Apathetic Future Share. We have learned that future jobs may interfere with greedy decisions in the past. We can avoid this pitfall by shifting from the optimistic view from greedy decisions to our AFS assumption that the future share of any existing job will be the same as the current share even if some jobs finish and release their shares. Thus, we set $p'_{a,2} = p_{a,1}$ and $p'_{b,2} = p'_{b,1}$ in (1) and (2).[10] Then, evaluating if either (1) or (2) is positive is translated into a simple inequality:

$$\frac{p'_{b,1} - p_{b,1}}{p'_{b,1}} > \frac{p'_{a,1} - p_{a,1}}{p_{a,1}}. \tag{3}$$

If (3) is true, j_b has priority over j_a for the extra GPU, and vice versa.

AFS implicitly prepares for future job arrivals by continuously adapting to the change of resource contention, which is done by re-evaluating (3) to re-adjust the shares of current jobs at each churn event. AFS tends to give higher priority to longer-but-efficient jobs when the contention level increases, refraining from over-committing resources to shorter-but-inefficient jobs (i.e., greedy approaches). Our evaluation in Section 5.2 shows that this assumption is highly effective in

[8]This analysis is similarly applied regardless of when j_c arrives.

[9]Appendix B provides more rigorous details.

[10]$p'_{a,2}$ is not $p'_{a,1}$ as $p'_{a,2}$ is used when j_b earns the GPU first (case (b2)).

real-world DLT workloads as AFS achieves the best average JCT reduction in all our traces.

The AFS assumption may prove ineffective in a few corner cases. One such case is when the future contention level decreases; jobs start simultaneously but no future jobs arrive until they finish. In this case, strategies acting more greedily (exploiting decreasing contention) may perform better than AFS. Although it is not uncommon in DLT workloads that a slew of jobs start at the same time, e.g. parameter sweeping [9], they typically run with other background jobs as we observe in the real cluster traces, rendering the decreasing contention assumption invalid. The other such case is when the future contention level increases; jobs start at different times and finish all at the same time. In this case, strategies acting more altruistically (exploiting increasing contention) may perform better than AFS, but we believe it is a rare case in real-world clusters. We finally note that if contention levels or their statistical characteristics are available, more sophisticated strategies than AFS can be devised to take advantage of them. For the time being, we focus on the more usual case where no such information is easily available.

3.3 AFS Algorithm for Multi-Job

Extending to n jobs. One may have noticed that extending the two-job case to the n-job case is highly non-trivial as it would be prohibitive to analyze all cases that potentially bring JCT reduction. To avoid the impasse, we directly apply our heuristic, AFS-L, to pick the job with the highest priority among n jobs for allocating each GPU. We determine the relative priority between any two jobs by evaluating (3) and apply the transitivity of priority comparison (see the proof in Appendix C) to find the job with the highest priority. We repeat this process M times, which results in $O(M \cdot n)$ steps for M GPUs with n jobs.

Algorithm 1 formally describes AFS-L, our resource allocation algorithm for n jobs under the assumption that workload size information for the jobs is known. At each churn event, Algorithm 1 determines the per-job GPU shares. The jobs with a positive share run until the next churn event. More specifically, AFS-L simply finds the job with the top priority (j^*) via TopPriority() and allocates one GPU to it, and repeats the same process M times. TopPriority() starts by picking two jobs at random. Given the current GPU shares, j_a is the shorter job and j_b is the longer one. If the current GPU share is zero, its length is considered infinite. Then it evaluates (3) to find the higher priority job and marks it as j^*. It repeats for all jobs to find the top choice.

AFS-L is a two-stage operation. In stage one, it assumes all jobs run with a single GPU and allocates one GPU in the increasing order of the job lengths. If M is smaller than the current number of jobs, the algorithm stops here. Otherwise, it moves into stage two, where it distributes the remaining GPUs by considering both job length and resource efficiency, which is achieved by evaluating (3).

Algorithm 1: AFS-L Resource Sharing

```
Function TopPriority(Jobs $J$)
    $j^* \leftarrow$ any job in $J$
    for $j \in J$ do
        $j_a \leftarrow j^*, j_b \leftarrow j$
        if $j_a.cnt = 0$ and $j_b.cnt = 0$ then
            if $j_a.len(1) < j_b.len(1)$ then $j^* \leftarrow j_a$
            else $j^* \leftarrow j_b$
        else
            if $j_a.len(j_a.cnt) \geq j_b.len(j_b.cnt)$ then
                Swap $j_a$ and $j_b$
            if (3) is true then $j^* \leftarrow j_b$
            else $j^* \leftarrow j_a$
    return $j^*$
Procedure AFS-L(Jobs $J$, Total Resources $M$)
    for $j \in J$ do
        $j.cnt \leftarrow 0$
    $m \leftarrow M$
    while $m > 0$ do
        $j^* \leftarrow$ TopPriority($J$)
        $j^*.cnt \leftarrow j^*.cnt + 1$
        $m \leftarrow m - 1$
    for $j \in J$ do
        Allocate $j.cnt$ resources for $j$
```

Handling unknown job lengths. AFS-L performs well, but it requires job length information to compare lengths of j_a and j_b in TopPriority(). As job length information is often unavailable, we modify TopPriority() to function without it by evaluating (3) for both cases: ($j_a = j^*$ and $j_b = j$) and ($j_a = j$ and $j_b = j^*$). (3) being true in either case indicates that prioritizing j_b is better for average JCT. If it evaluates to false in both cases, the real priority would depend on the finishing order of the jobs, which cannot be decided without knowing job lengths. In such a case, we give a higher priority to either one at random. Note that (3) cannot be true in both cases.

AFS-P is our job-length-unaware algorithm that modifies AFS-L by adopting the traditional *processor sharing* [31] approach to mimic the SRTF-like behavior of stage one in AFS-L. Specifically, the algorithm maintains a counter for each job that tracks the amount of time for which it is scheduled. The counter is zero at job arrival and increases by one whenever the job is executed for one unit of time.[11] If there are fewer jobs than M, the algorithm ignores the counters and uses modified TopPriority() to determine the shares. If the number of jobs exceeds M (the algorithm stays in stage one), it allocates one GPU in the increasing order of the job counters in a non-preemptive manner and evicts a job whenever it is executed for one unit of time, which triggers the scheduler to re-schedule all jobs. This policy mitigates job blocking if the current number of jobs exceeds M, but it is unnecessary otherwise as every job would run with at least one GPU. Appendix D provides more discussion on AFS.

[11] AFS-P set the unit time as 2 hours for all full-scaled traces used in this work. Like existing practices of processor sharing, we should scale the unit time if the workload is in a different scale.

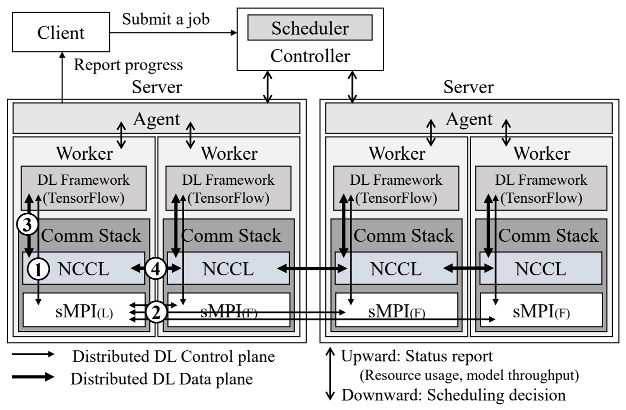

Client
Submit a job
Scheduler
Controller
Report progress
Server
Agent
Worker
DL Framework (TensorFlow)
Comm Stack
NCCL
sMPI(L)
sMPI(F)
Distributed DL Control plane
Distributed DL Data plane
Upward: Status report (Resource usage, model throughput)
Downward: Scheduling decision

Figure 4: Overview of CoDDL system architecture. There are two servers for the cluster and two GPUs for each server. sMPI (L) and (F) refer to the leader and the follower stacks, respectively.

4 Systems Support for Elastic Sharing

Elastic share adaptation requires every job to reconfigure to an arbitrary number of GPUs at any time. To support this efficiently, we present CoDDL, our DLT system framework that transparently handles automatic job parallelization.

4.1 CoDDL System Architecture

CoDDL is a centralized resource management system for a dedicated DLT cluster. It enables the system administrator to specify a scheduling policy and also parallelizes DLT jobs automatically. The key requirement of this system is to minimize the overhead incurred by frequent reconfiguration of DLT jobs to their elastically adapted resource shares.

CoDDL is divided into a front-end and a back-end. The front-end interacts with users (DLT job owners). It accepts a DL model to train from a user and reports its progress back periodically to the user. The current user interface is based on TensorFlow [6], although we can easily support other frameworks as well. A user specifies her model via native TensorFlow APIs and provides the batch size and her model graph (automatically extracted via native TensorFlow APIs) to a CoDDL client. The client submits these to the back-end system that initiates a DLT job.

The back-end determines resource shares for the submitted jobs and allocates them. It consists of a central controller (which includes the scheduler), per-server agents, and per-GPU workers as shown in Figure 4. To clarify, a GPU cluster consists of multiple servers, each of which consists of multiple GPUs. The controller invokes the scheduler to determine the shares and notify the agents of the updated per-job shares. The agents in turn order their local workers to reflect the changes. The agents manage the intra-server workers and the workers carry out actual DLT with their own GPU.

4.2 Automated Parallel Training

A CoDDL user does not need to write a model for multiple GPUs considering parallel execution. Rather, she writes it only for a single GPU and the system automatically parallelizes it to run with an arbitrary number of GPUs. The basic approach to the parallelization is similar to existing works [1, 4, 30, 45] while it seamlessly supports a large batch size that exceeds the physical memory size of a GPU.

Auto-parallelization. The key to job parallelization is to enable each worker to exchange gradients for model update with an arbitrary number of co-workers. To achieve this, we insert special vertices for every pair of gradient and updater vertices such that these new vertices work as a messenger between the DLT framework and our custom communication stack (see Section 4.3 for details) in Figure 4. Whenever a gradient is calculated, the newly added vertex notifies the communication stack of the availability of the gradient and registers a callback function. Then, the communication stack reduces the gradients, and triggers the callback function so that the DLT framework starts updating the parameters.

Accumulative gradient update. Per-GPU batch size of a job is determined by its total batch size and GPU share. The smaller the GPU share of the job is, the larger the per-GPU batch size gets. Hence, the memory footprint often exceeds the physical memory budget of a single GPU. To prevent memory overflows, CoDDL transparently performs accumulative gradient update by adding extra vertices to the graph. It selects the batch size small enough to fit the memory budget of one GPU and repeatedly adds up the gradient results until it reaches the originally-set per-GPU batch size.

4.3 Efficient Share Re-adjustment

Elastic resource sharing schedulers tend to perform frequent share re-adjustments. In fact, we observe that AFS-P executes 6.3x on average or up to 22x more reconfigurations than Tiresias-L for real-world traces. To address the overhead, CoDDL optimizes the reconfiguration process and avoids potential thrashing due to a burst of reconfigurations that arrive in a short time.

Custom communication stack. CoDDL implements its own communication stack so that workers can dynamically join or leave an on-going DLT job without checkpointing and restarting it. It consists of a data-plane (NCCL [2]) stack and a control-plane (sMPI) stack. The data plane is responsible for efficient reduce operations with a static set of workers. The control plane reconfigures NCCL whenever there is a GPU share re-adjustment order from the controller (①). It also communicates with co-workers to monitor the availability of intermediate data (e.g., gradients and parameters) (②), and invokes NCCL APIs to exchange it (③, ④). One of the control-plane stacks operates as a leader, which periodically polls all other followers if it is ready to join or leave the job, and shares this information with all followers.

Concurrent share expansion with job execution. CoDDL mitigates the overhead for resource share expansion by continuing job execution during the reconfiguration. This would minimize the period of inactivity caused by frequent reconfigurations. The key enabler for this is to separate the independent per-GPU operations from the rest that communicates with remote GPUs in distributed DLT. The independent operations (e.g., forward/backward propagation, gradient calculation, updating parameters) can be executed on their own regardless of the number of remote GPUs. Only the operations that gather the gradients would depend on the communication with remote GPUs. So, when a job is allocated more GPUs, the job continues the execution with the old share while newly-allocated GPUs are being prepared; the new GPUs are loaded with the graph, and receive the latest parameters from one of the existing workers in the job. When the new GPUs are ready to join, they tell other workers to update the communication stack to admit the new GPUs. We find the overhead for this operation is as small as 4 milliseconds.

Zero-cost share shrinking. The process of share shrinking is even simpler. When the GPU share of a job decreases, the job only needs to tell its communication stack to reflect the shrunk share and continues the execution with the remaining GPUs. Then, the kernels on the released GPUs are stopped and tagged as idle. Again, it takes only 4 milliseconds regardless of a model.

Handling a burst of reconfigurations. While running our system, we often observe a burst of reconfiguration orders from the controller as multiple churn events happen simultaneously. Naïvely carrying them out back to back would be inefficient as the older configuration would be readily nullified by the newer one. There are two approaches to efficiently handling them. One approach is to coalesce multiple consecutive reconfiguration orders into one while the current reconfiguration is going on. While it is simple to implement, the controller would need to synchronize with all agents and workers before initiating the next reconfiguration. Instead, we implement "cancelling" the current reconfiguration on the individual worker level. In this approach, a new reconfiguration order will be delivered to the agents as soon as it is available, but the agent can cancel the on-going reconfiguration with its workers. Unless careful, this could create a subtle deadlock as one worker may leave a job on a new reconfiguration order while the rest of the workers wait for it indefinitely in a blocking call (e.g., initializing, all-reduce, etc.). To avoid the deadlock, CoDDL allows a worker to leave the job only when all others are aware of it.

Failure handling. Efficient share re-adjustment of CoDDL makes it easy to handle failures efficiently as it is similar to handling churn events. The controller and each agent are responsible for monitoring the status of all agents and the workers of its own, respectively. If any of them stops responding or returns a fatal error, it is reported to the controller, and it simply excludes the failed entities from the available resource list and re-runs the scheduler to reconfigure all jobs. Unlike ordinary churn events, it is deadlock-free for a failed worker to exit without informing to its co-workers since it is done by agents and the controller instead.

4.4 Network Packing and Job Migration

Depending on the DL model, the placement of GPUs may affect the training performance. It is generally more beneficial to use as fewer machines as possible for the same number GPUs to minimize network communication overhead [34,55]. Also, the performance could be unpredictable if multiple jobs compete for network bandwidth on the same machine.

CoDDL adopts a simple yet effective GPU placement mechanism called *network packing*, which enforces the GPU share of each job to be a factor or a multiple of the number of GPUs per machine. For instance, in a cluster with 4 GPUs per machine, the share of a job should be one of $1/2/4n$ GPUs. This regulation can be proven to eliminate the "network sharing", which satisfies the following two conditions – (1) every job is allocated GPUs scattered on the minimum number of machines and (2) at most one job on a machine communicates with remote GPUs on other machines (while all other jobs on the machine use only local GPUs). When the GPU share is determined by the scheduling algorithm, the scheduler applies the regulation to come up with an allocation plan that minimizes the difference from the original plan. More details on the algorithm and the proof are found in Appendix E.

Despite the network packing, we still need job migration as it does not prevent resource fragmentation. Since a job can continue training without migration unless the newly allocated GPU set does not include any GPU that the job was previously using, the scheduler finds a placement decision that maximizes the number of jobs which reuse at least one GPU, and then maximizes the total number of reused GPUs as the secondary goal. It conducts migration as the final resort when it is unavoidable.

4.5 Throughput Measurement

Algorithms like Optimus [38] and AFS require the throughput (or throughput scalability for AFS-P) with an arbitrary number of GPUs for determining the GPU share. While Optimus estimates the throughput scalability with a few sampled measurements for iteration, gradient calculation, and data transmission, we find it unreliable due to the heterogeneity of computing hardware, network topology, and DL system frameworks. Instead, CoDDL takes the *"overestimate-first-and-re-adjust-later"* approach. It over-allocates the number of GPUs first but re-adjusts the share from real throughput measurements later. This exploits the fact that GPU share shrinking incurs a small overhead of only a few milliseconds.

Overestimate first and re-adjust later. When a new job arrives, the scheduler allocates an initial share to it by assuming the linear throughput scalability. We limit the initial share to be within its fair share (1/n). When the job gets its GPU share, it takes the throughput measurement with all allocatable units that are smaller than the given share. Note that this measurement is done efficiently as shrinking the share incurs little cost (Section 4.3) and all iterations used for the measurement are part of training. In most cases, it requires only two reconfigurations for a new job arrival. In a rare case where a job needs to predict the throughput beyond the currently allocated share, it assumes the linear scalability from the last allocatable unit. Then, the re-adjustment is made again when the real throughputs are reported to the controller.

5 Evaluation

We evaluate AFS if it improves average JCT on a diverse set of real-world DLT workloads against existing scheduling policies. We evaluate it on simulation as well as on real-world cluster with CoDDL.

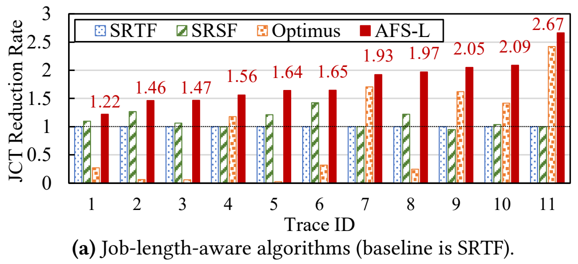
5.1 Experiment Setup

Cluster setup. We use a 64-GPU cluster for experiments. Each server is equipped with four NVIDIA GTX 1080 GPUs, two 20-core Intel Xeon E5-2630 v4 (2.20GHz), 256 GB RAM, and a dual-port 40 Gbps Mellanox ConnectX-4 NIC. Only one NIC port of each server is connected to a 40 GbE network switch and the servers communicate via RDMA (RoCEv2), leveraging NCCL 2.4.8 [2]. All DLT jobs run TensorFlow r2.1 [6] with CUDA 10.1 and cuDNN 7.6.3.

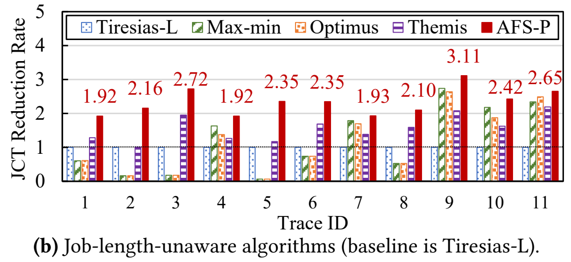
DLT workload. We use 137-day real-world traces from Microsoft [5, 27] (Table 3 in Appendix). We evaluate with *all* traces except those that have fewer than 100 jobs. From the traces, we use submission time, elapsed time, and requested (allocated) numbers of GPUs of each job. Since the traces do not carry training model information, we submit a random model chosen from a pool of nine popular DL models (Table 4 in Appendix) whose training throughput scales up to the requested number of GPUs. Each DLT job submits the chosen model for training in the BSP manner with the same batch size throughout training, even though the GPU share changes during the training. The number of training iterations of the job is calculated so that its completion time becomes equal to the total executed time from the trace, based on the use of the requested number of GPUs.

Simulator. We implement a simulator to evaluate the algorithms for large traces. All experimental results without explicit comments are from the simulator. The simulator is provided with the measured throughput of each model on our real cluster for job length calculation. We confirm that simulation results conform to those on real execution for all algorithms with scaled-down traces. Figure 15 in Appendix shows examples of our scheduler conformance tests. The error rate is low (<1%) for algorithms that do not require throughput measurements but slightly high for others (2.7~5.2%) due to the measurement overhead on real cluster.

Algorithms. Scheduling algorithms are divided into two categories: job-length-aware and job-length-unaware. The former consists of SRTF, SRSF [24], Optimus [38], and AFS-L while the latter consists of Tiresias-L [24], Optimus, Max-Min, Themis [34], and AFS-P. Optimus is counted into both as real job lengths differ from the estimated ones by Optimus. We just assume that their estimation is always correct.



(a) Job-length-aware algorithms (baseline is SRTF).



(b) Job-length-unaware algorithms (baseline is Tiresias-L).

Figure 5: Average JCT reduction rates with respect to baselines for scheduling algorithms.

Metrics. We evaluate average JCT, makespan, QL, CE, and BI, and GPU utilization during runtime for real experiments. While makespan is often used to evaluate resource efficiency of scheduling algorithms [22, 24, 38], we believe CE is a better metric as makespan would depend on the last job submission time, which is orthogonal to resource efficiency. CE does not suffer from such an issue as it reports the reduction rate of makespan per unit time instead.

5.2 JCT Evaluation on Simulation

Figure 5 compares average JCT reduction rates for each category of the algorithms. Values larger than 1 mean that their average JCT is better than the baseline algorithms (SRTF or Tiresias-L). Interestingly, we observe that Max-Min and Optimus show a similar trend on the same set of traces (1,2,3,5,6,8). These traces launch many jobs in a short interval while many of them run longer than other traces. In such cases, it is important to serve shorter jobs first to avoid HOL blocking, but neither does so. Not surprisingly, SRSF outperforms SRTF on these traces as penalizing jobs with more GPUs helps reduce HOL blocking beyond favoring short jobs. AFS-L and AFS-P consistently outperform all others in their category by 1.2x to 2.7x over SRTF or by 1.9x to 3.1x over Tiresias-L. This shows that resource efficiency-aware HOL blocking mitigation is effective in practice.

One may argue that non-elastic sharing algorithms would achieve better JCT if we allow them to use more GPUs as we do for elastic-share algorithms. However, we find that it only marginally improves average JCT or even performs

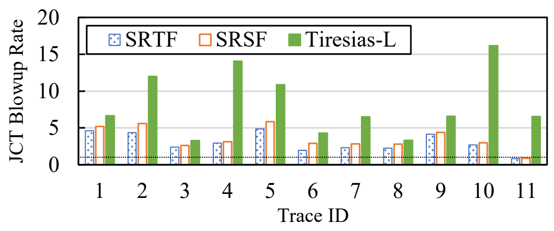


Figure 6: Average JCT blowup rates when we allow SRTF, SRSF, and Tiresias-L [24] to use the maximum number of GPUs each job can utilize. Dotted line indicates unity.

worse. For example, if we allow SRTF, SRSF, and Tiresias-L to allocate enough GPUs for peak throughput of each job, Figure 6 shows that their average JCTs rapidly grow by up to 4.85x, 5.84x, and 16.14x, respectively. Only one trace improves by 21% and 17% for SRTF and SRSF. This is because non-elastic allocation with a larger GPU share reduces the cluster efficiency as it would increase the inefficiency of individual resource. What matters more is to flexibly adjust the GPU share according to job lengths and resource efficiency, which is impossible with non-elastic allocation.

5.3 Evaluation on Real Cluster

Algorithm behavior on real cluster. Figure 7 compares job-length-unaware algorithms running on our GPU cluster. As full-scale execution would take months to finish, we scale down the job submission times and total iterations to 1% and 0.2% of the original values for traces #9 and #3, respectively.

We observe similar trends with QL, CE, and BI as those in the full-scale simulations in Figure 2. AFS-P achieves (a) 3.54x and (b) 2.93x better average JCT over Tiresias-L, (a) 1.12x and (b) 1.66x better than Optimus, and (a) 1.22x and (b) 1.30x better than Themis. Note that the CEs of Optimus and AFS-P fluctuate heavily in Figure 7a because they complete most of the jobs in the cluster rather quickly, which decreases the CE due to underutilization (see the low QL). Optimus, Themis, and AFS-P show higher GPU utilizations than that of Tiresias-L in Figure 7b as they tend to assign fewer GPUs to each job than Tiresias-L especially when there are many jobs in the cluster.

Efficient share re-adjustment. We evaluate the effectiveness of CoDDL's share re-adjustment over a baseline system without concurrent share expansion and zero-cost shrinking (ES) or without reconfiguration cancelling (RC). We monitor the training progress of a specific job (video prediction model) along with other jobs on a full-scale trace (#10) extracted from day 50 and 116. Figure 8 compares job progress, allocated GPU shares, and # of jobs submitted over time.

In Figure 8a, we observe that the job finishes 2.12x and 2.82x faster than "No RC" and "No RC/ES", respectively. On this day, 19 short jobs enter the system in a short time interval of 2 to 70 seconds for the first few minutes. During this time, the job with ES and RC is allocated a much larger GPU share than the one with full restart. This is because ES enables

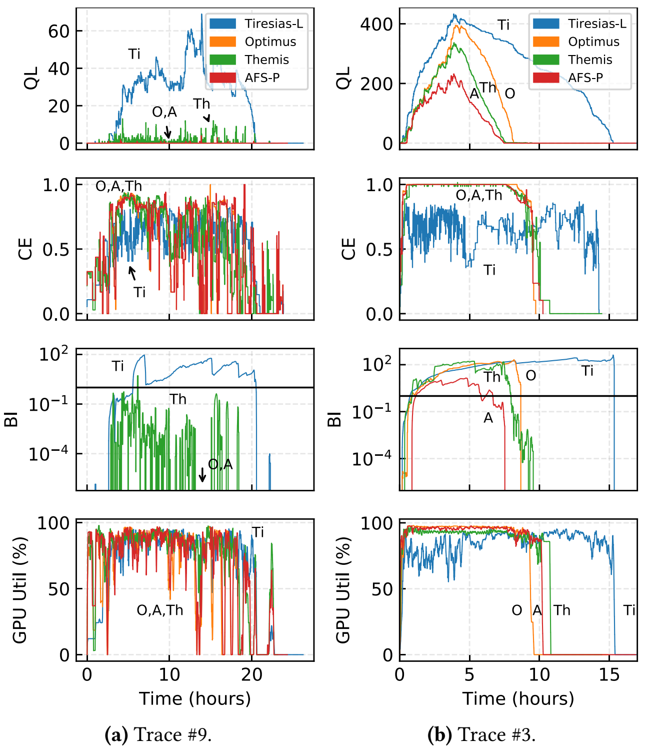


Figure 7: Real-cluster execution over CoDDL using Tiresias-L (Ti), Themis (Th), Optimus (O), and AFS-P (A) schedulers. Average JCTs are (a) 1.45, 0.50, 0.46, and 0.41 hours (b) 5.65, 2.51, 3.21, and 1.93 hours, respectively.

the short jobs to finish more quickly as they can continue job execution even during reconfiguration. This enables the monitored job to earn more GPUs. In addition, RC helps the job adapt to newly-assigned resources quickly even when the controller updates the resource share frequently. In the graph, we see that all short jobs that arrive at ④ finish within 1.50 minutes (①) while "No RC" and "No RC/ES" take 4.10 (②) and 5.06 minutes (③), respectively.

Figure 8b shows results on day 116 where there are a slew of job submissions (68 jobs) at start, commonly seen in DLT clusters (e.g. parameter sweeping [9]). The monitored job finishes 1.19x and 1.30x faster than "No RC" and "No RC/ES", respectively. The AFS-P scheduler performs more stably at heavy contention as each job would employ a small share (one GPU per job if the number of jobs exceeds that of GPUs). In this case, a churn would rarely affect other jobs as most of them are likely to keep their previous share. This explains the smaller benefit compared to the scenario in Figure 8a.

5.4 Evaluation of Tail JCTs

DLT jobs in a multi-tenant cluster tend to be heavy-tailed (see Appendix F), so we evaluate tail JCTs of schedulers. Unlike other systems whose tail latency directly measures the worst-case system (or scheduler) performance caused by the congestion, tail JCT in DLT clusters is a fairness metric that evaluates the trade-off between average and tail JCTs.

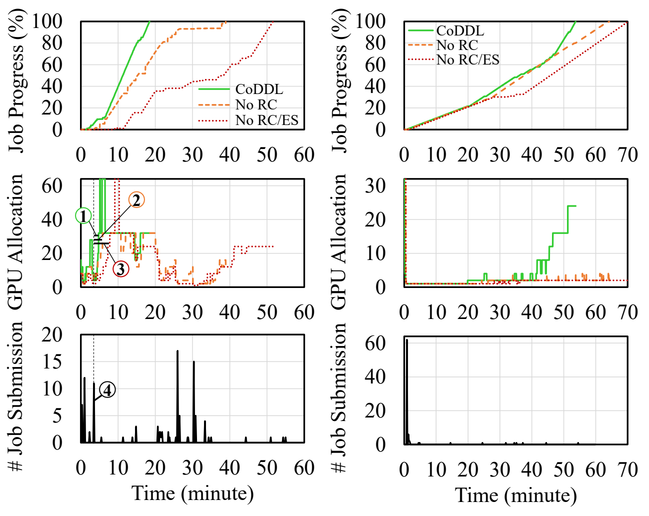
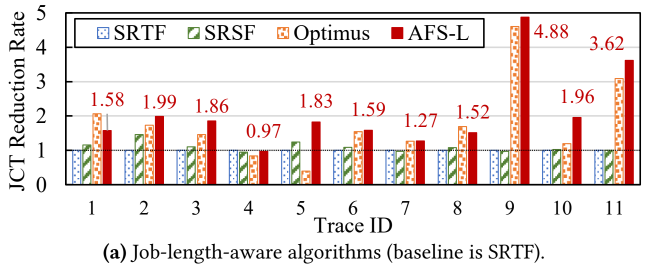

(a) Day 50. **(b)** Day 116.

Figure 8: Real-cluster execution of parts of full-scale trace #10 over CoDDL using AFS-P scheduler. X-axis indicates the elapsed time since the arrival of the monitoring job. "No RC/ES" means that we disable both RC and ES.

The difference arises as the intrinsic characteristic of a DLT job (i.e. how much resource it can utilize and how long it runs) often dominates the JCT rather than the congestion itself. Consequently, it is often hard to blame the system for a long tail when the major causes are (1) the user's request to run for a long time and (2) the scheduler's decision that avoids allocating more resources to those jobs that incur the average-tail trade-off (or fairness) issue.

Figure 9 shows that AFS reduces tail JCTs over existing schedulers in most cases in addition to the average JCTs, as demonstrated in Figure 5 as well. Actually, other scheduling algorithms targeting for reducing average JCT such as SRTF often suffer from severe job starvation. In contrast, it is interesting that AFS experiences little starvation (thus shows good tail JCTs) while achieving low average JCT for online jobs. As explained, AFS allocates at least one GPU for all jobs when $n < M$ (stage one), and distributes remainders when $n \geq M$ (stage two), which helps avoid job starvation. The low BI values of AFS in Figure 7 also implies that it effectively avoids starvation.

In some traces, Max-Min or Optimus shows better tail JCTs over AFS. This is an expected behavior because Max-Min and Optimus do not prioritize short jobs at all, which is relatively more beneficial to tail JCT reduction. However, this does not necessarily mean that they are fairer than AFS because the tail JCT reduction is often achieved by sacrificing the average JCT substantially, as is already shown in Figure 5. To clarify this, Figure 10 compares the JCT reduction rates over varying job lengths. The figures group all jobs into 100 bins in the increasing JCT order on the X-axis (a longer job comes more rightward) and plot the average reduction rate of each bin on the Y-axis (a more prioritized job shows up higher).


(a) Job-length-aware algorithms (baseline is SRTF).

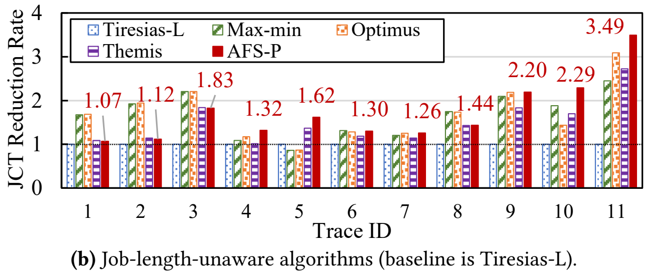

(b) Job-length-unaware algorithms (baseline is Tiresias-L).

Figure 9: 99th%-ile JCT reduction rates with respect to baselines for scheduling algorithms.

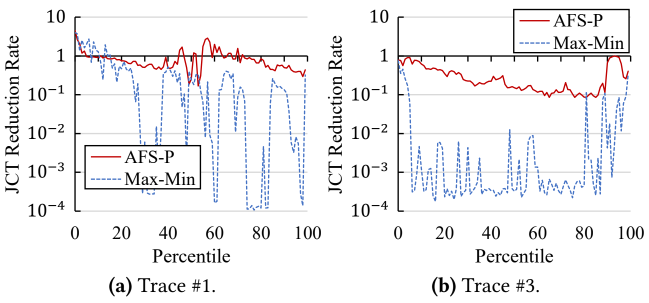

(a) Trace #1. **(b)** Trace #3.

Figure 10: JCT reduction rates over varying job lengths. Results of other traces are omitted as they show similar results.

The result indicates that Max-Min not only performs more poorly on average but it also shows uneven performance gain compared to AFS-P. Especially when the cluster is heavily contended (trace #3), it tends to prioritize very long jobs by its algorithmic design.[12] While we do not present here, Optimus also shows the similar behavior as Max-Min. In contrast, AFS-P demonstrates relatively more even performance gain regardless of the job length.

5.5 Benefit over Altruistic Approach

One low-hanging fruit of leveraging elastic resource sharing is that it can enhance cluster efficiency by distributing idle resources to running jobs. Altruistic scheduling [22] is a straightforward approach to achieving this: it first schedules jobs to achieve the primary goal (in this case, assigning the user-requested number of GPUs) and then distributes leftover resources for the secondary goal (in this case, assigning more GPUs to achieve the largest throughput). Figure 11 shows

[12]Figure 10 shows that Max-Min also prioritizes very short jobs. This is a common feature of job-length-agnostic schedulers (including AFS-P) rather than being specific to Max-Min, because very short jobs are less affected by the algorithmic detail of the scheduler as they typically finish as soon as they are first scheduled.

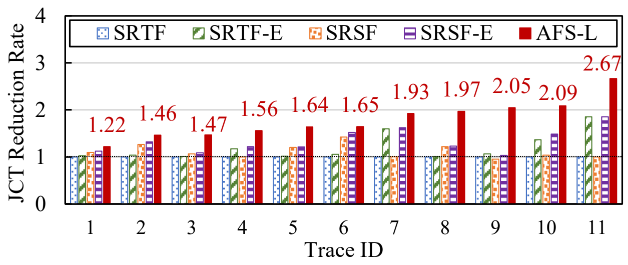


Figure 11: Benefit of altruistic approaches (SRTF-E and SRSF-E) to reduce average JCT.

the performance of this approach when applied to SRTF and SRSF to make their elastic versions, SRTF-E and SRSF-E, respectively. Even though these reduce average JCT from their non-elastic versions, AFS-L still shows a substantial benefit over these, which indicates the importance of fine-grained individual resource-level adjustment.

6 Related Work

Leveraging optimality of SRTF. SRTF, also known as Shortest-Remaining-Processing-Time discipline (SRPT) or preemptive Shortest-Job-First (SJF), has been proven to minimize average JCT in cases where a single resource is available [18,44,48]. In case of multiple resources, SRTF is optimal only when the throughput of all jobs scales linearly to the given amount of resources, which does not hold for DLT jobs due to inter-GPU communication overhead. In general, optimal scheduling of online jobs is NP-complete [17].

Even though SRTF does not perform optimally in real-world systems, many practical cluster schedulers [21–24] leverage it as a good heuristic for reducing average JCT. Among them is Tiresias [24] designed for DLT job scheduling similar to this work. It suggests a variation of SRSF that considers remaining time multiplied by requested number of GPUs instead of remaining time alone. This approach penalizes jobs that request many GPUs. Compared to SRTF, SRSF improves average JCT because such jobs typically utilize GPUs less efficiently, thus it is often better to execute jobs using fewer GPUs instead for resource efficiency. Tiresias also focuses on relaxing the constraint of both SRTF and SRSF that job lengths should be known in advance. In comparison, our algorithms handle more general cases beyond penalizing the jobs with a large share. Also, our algorithms benefit from more fine-grained resource allocation and preemption, which substantially improves both average and tail JCTs.

Elastic sharing algorithms. Optimus [38], OASiS [8], and Themis [34] are DLT job schedulers that adopt elastic sharing similar to this work. Optimus estimates finishing time of a DLT job via modeling loss degradation speed and performing online fitting of the model during training, and designs a heuristic algorithm to minimize average JCT. However, the approach raises two issues. First, it is unclear whether it is always possible to reliably estimate loss degradation speed, which is also questioned by a follow-up work [24]. Second, Optimus has no mechanism to prioritize short jobs, which would often suffer from HOL blocking that results in JCT blowup. OASiS solves an integer linear program to maximize the system throughput. As maximizing system throughput often incurs severe starvation, it does not fit the goal of our work. Themis suggests a resource auction algorithm mainly motivated for fairness, but it actually achieves the closest JCT performance to AFS-P among all existing works. This is because it leverages elastic resource sharing for its partial auction algorithm and random distribution of leftover resources, prioritizes short jobs by setting a static lease duration of assigned GPUs, and also tends to prioritize efficient jobs since it is fairness-motivated, which prevents greedy jobs (which are typically resource-inefficient) from monopolizing resources. However, its optimization target is maximizing fairness, which is not always ideal for reducing average JCT. To be specific, if a job shows its largest throughput with m GPUs and the job has been running with other $n-1$ jobs on average over time, Themis's fairness metric tries to assign GPUs to this job until it achieves at least m/n times of throughput speedup.[13] Compared to AFS, this could give much higher priority to inefficient jobs which is fairer but it could degrade overall cluster performance.

Systems for elastic sharing. While there are several works which suggest an elastic sharing algorithm for DLT job scheduling [8,34,38], none of them fully cover the necessary systems support for elastic sharing: automatic scaling, efficient scale-in/out, and efficient reconfiguration handling. A recent work [36] implement an efficient method for scaling a single job without stop-and-resume similar to the ES in Section 5.3 of our work. However, it focuses only on scaling a single job, and does not cover handling churn events efficiently in multi-job scheduling. We believe the latter is the key to evaluating an elastic resource sharing system since its overhead hugely depends on the frequency of churn events as evidenced by the different amount of gain of either ES or RC in Figure 8a and 8b.

7 Conclusion

Existing scheduling policies have been clumsy at handling the sublinear throughput scalability inherent in DLT workloads. In this work, we have presented AFS, an elastic scheduling algorithm that tames this property well into average JCT minimization. It considers both resource efficiency and job length at resource allocation while it amortizes the cost of future jobs into current share calculation. This effectively improves the average JCT by bringing 2.2x to 3.1x reduction over the start-of-the-art algorithms. We have also identified essential systems features for frequent share adaptation, and have shown the design with CoDDL. We hope that our efforts in this work will benefit the DLT systems community.

[13]This does not cover all features of Themis. Refer to [34] for details.

Acknowledgments

We appreciate the feedback by our shepherd, Minlan Yu, and anonymous reviewers of NSDI '21, and thank Kyuho Son for early discussion on the CoDDL system. This work is in part supported by the National Research Council of Science & Technology (NST) grant by the Korean government (MSIP) No. CRC-15-05-ETRI, and by the ICT Research and Development Program of MSIP/IITP, under projects, [2016-0-00563, Research on Adaptive Machine Learning Technology Development for Intelligent Autonomous Digital Companion], and [2018-0-00693, Development of an ultra low-latency user-level transfer protocol].

References

[1] DeepSpeed. https://www.deepspeed.ai/.

[2] NVIDIA Collective Communications Library (NCCL). https://developer.nvidia.com/nccl.

[3] Open MPI. https://open-mpi.org.

[4] PaddlePaddle. https://github.com/PaddlePaddle/Paddle.

[5] Philly traces. https://github.com/msr-fiddle/philly-traces.

[6] TensorFlow. https://tensorflow.org.

[7] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Erich Elsen, Jesse H. Engel, Linxi Fan, Christopher Fougner, Awni Y. Hannun, Billy Jun, Tony Han, Patrick LeGresley, Xiangang Li, Libby Lin, Sharan Narang, Andrew Y. Ng, Sherjil Ozair, Ryan Prenger, Sheng Qian, Jonathan Raiman, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Chong Wang, Yi Wang, Zhiqian Wang, Bo Xiao, Yan Xie, Dani Yogatama, Jun Zhan, and Zhenyao Zhu. Deep speech 2 : End-to-end speech recognition in english and mandarin. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2016.

[8] Yixin Bao, Yanghua Peng, Chuan Wu, and Zongpeng Li. Online job scheduling in distributed machine learning clusters. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, 2018.

[9] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13:281–305, 2012.

[10] Mauro Cettolo, Niehues Jan, Stüker Sebastian, Luisa Bentivogli, Roldano Cattoni, and Marcello Federico. The IWSLT 2016 evaluation campaign. In *Proceedings of the International Conference on Spoken Language Translation (IWSLT)*, 2016.

[11] Yuntao Chen, Chenxia Han, Yanghao Li, Zehao Huang, Yi Jiang, Naiyan Wang, and Zhaoxiang Zhang. Simpledet: A simple and versatile distributed framework for object detection and instance recognition. *CoRR*, abs/1903.05831, 2019.

[12] Henggang Cui, Hao Zhang, Gregory R. Ganger, Phillip B. Gibbons, and Eric P. Xing. GeePS: scalable deep learning on distributed GPUs with a GPU-specialized parameter server. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2016.

[13] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew W. Senior, Paul A. Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2012.

[14] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Fei-Fei Li. ImageNet: A large-scale hierarchical image database. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2009.

[15] Chelsea Finn, Ian J. Goodfellow, and Sergey Levine. Unsupervised learning for physical interaction through video prediction. *CoRR*, abs/1605.07157, 2016.

[16] Lex Fridman, Benedikt Jenik, and Jack Terwilliger. Deeptraffic: Driving fast through dense traffic with deep reinforcement learning. *CoRR*, abs/1801.02805, 2018.

[17] Michael R. Garey, David S. Johnson, and Ravi Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research*, 1(2):117–129, 1976.

[18] Natarajan Gautam. *Analysis of Queues: Methods and Applications*. CRC Press, 1 edition, 2017.

[19] Ahmad Ghazal, Todor Ivanov, Pekka Kostamaa, Alain Crolotte, Ryan Voong, Mohammed Al-Kateb, Waleed Ghazal, and Roberto V. Zicari. Bigbench V2: the new and improved bigbench. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2017.

[20] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Proceedings of the USENIX Symposium on*

Networked Systems Design and Implementation (NSDI), 2011.

[21] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2014.

[22] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[23] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. GRAPHENE: packing and dependency-aware scheduling for data-parallel clusters. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[24] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Harry Liu, and Chuanxiong Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[25] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[26] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy H. Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[27] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2019.

[28] Ye Jia, Yu Zhang, Ron J. Weiss, Quan Wang, Jonathan Shen, Fei Ren, Zhifeng Chen, Patrick Nguyen, Ruoming Pang, Ignacio Lopez-Moreno, and Yonghui Wu. Transfer learning from speaker verification to multispeaker text-to-speech synthesis. In *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2018.

[29] Nal Kalchbrenner, Erich Elsen, Karen Simonyan, Seb Noury, Norman Casagrande, Edward Lockhart, Florian Stimberg, Aäron van den Oord, Sander Dieleman, and Koray Kavukcuoglu. Efficient neural audio synthesis. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2018.

[30] Soojeong Kim, Gyeong-In Yu, Hojin Park, Sungwoo Cho, Eunji Jeong, Hyeonmin Ha, Sanha Lee, Joo Seong Jeong, and Byung-Gon Chun. Parallax: Sparsity-aware data parallel training of deep neural networks. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2019.

[31] Leonard Kleinrock. Time-shared systems: a theoretical treatment. *Journal of the ACM*, 14(2):242–261, 1967.

[32] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[33] Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. Deep learning face attributes in the wild. In *Proceedings of the International Conference on Computer Vision (ICCV)*, 2015.

[34] Kshiteej Mahajan, Arjun Singhvi, Arjun Balasubramanian, Varun Batra, Surya Teja Chavali, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient GPU cluster scheduling for machine learning workloads. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[35] Ozan Oktay, Jo Schlemper, Loïc Le Folgoc, Matthew C. H. Lee, Mattias P. Heinrich, Kazunari Misawa, Kensaku Mori, Steven G. McDonagh, Nils Y. Hammerla, Bernhard Kainz, Ben Glocker, and Daniel Rueckert. Attention u-net: Learning where to look for the pancreas. *CoRR*, abs/1804.03999, 2018.

[36] Andrew Or, Haoyu Zhang, and Michael J. Freedman. Resource elasticity in distributed deep learning. In *Proceedings of the Conference on Machine Learning and Systems (MLSys)*, 2020.

[37] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. Librispeech: An ASR corpus based on public domain audio books. In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2015.

[38] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2018.

[39] Meikel Poess, Tilmann Rabl, and Hans-Arno Jacobsen. Analysis of TPC-DS: the first standard benchmark for sql-based big data systems. In *Proceedings of the Symposium on Cloud Computing (SoCC)*, 2017.

[40] Tong Qin and Shaojie Shen. Online temporal calibration for monocular visual-inertial systems. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2018.

[41] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *CoRR*, abs/1511.06434, 2015.

[42] Benjamin Recht, Christopher Ré, Stephen J. Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2011.

[43] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *Proceedings of the Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, 2015.

[44] Linus E. Schrage and Louis W. Miller. The queue M/G/1 with the shortest remaining processing time discipline. *Operations Research*, 14(4):670–684, 1966.

[45] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *CoRR*, abs/1802.05799, 2018.

[46] Joan Serrà, Santiago Pascual, and Carlos Segura. Blow: a single-scale hyperconditioned flow for non-parallel raw-audio voice conversion. In *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2019.

[47] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[48] Donald R. Smith. Technical note - A new proof of the optimality of the shortest remaining processing time discipline. *Operations Research*, 26(1):197–199, 1978.

[49] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[50] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[51] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. Deepface: Closing the gap to human-level performance in face verification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2014.

[52] Jörg Tiedemann. News from OPUS - A collection of multilingual parallel corpora with tools and interfaces. In *Recent Advances in Natural Language Processing*, volume V, pages 237–248. 2009.

[53] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2017.

[54] Oriol Vinyals and Quoc V. Le. A neural conversational model. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2015.

[55] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[56] Mengjia Yan, Mengao Zhao, Zining Xu, Qian Zhang, Guoli Wang, and Zhizhong Su. Vargfacenet: An efficient variable group convolutional neural network for lightweight face recognition. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV) Workshops*, 2019.

[57] Yujie Zhong, Relja Arandjelovic, and Andrew Zisserman. Ghostvlad for set-based face recognition. In *Proceedings of the Asian Conference on Computer Vision (ACCV)*, 2018.

Appendix

A Future Dependency of Job Scheduling

At first glance, it may appear as if we could find the optimal allocation of resource shares by taking into account all factors we discussed. We would simply need to know the lengths of all jobs that vary according to their GPU usage, enumerate all possible allocation candidates, and compare the resultant average JCTs to obtain the smallest.

Not only would such an attempt be infeasible due to the exponential growth of possible candidates,[14] a fundamental challenge underlies the usage of a GPU cluster in practice: churn by future jobs. They can wreak havoc on previous resource allocation decisions that otherwise would have been optimal.

To see this, let us consider a toy example. Suppose we have two jobs, A and B, initially submitted. With 1 GPU, each job takes 2 hours to finish. With 2 GPUs, they take 1 and 1.5 hours, respectively. Let us assume that we run them on a 2-GPU cluster. The optimal strategy to achieve the minimal average JCT of 1.25 hours would be SRTF: A is scheduled first (1 hour) with 2 GPUs and then B (1.5 hours). A naïve alternative that allocates all available resource shares equally to all pending jobs (assuming for the sake of simplicity that the number of pending jobs does not exceed the total number of shares) would achieve the average JCT of 2 hours. What if an unforeseen arrival of future jobs comes into the picture?

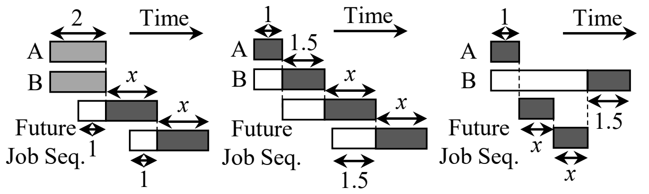
Suppose a sequence of identical future jobs were to arrive. They take 2 hours with 1 GPU and x hours ($1 < x < 2$) with 2 GPUs. Let us consider a scenario in which the first future job arrives 1 hour later since the beginning and all subsequent jobs arrive x hours later than the preceding one. Regardless of the value of x, the naïve alternative achieves smaller average JCT than SRTF (Figure 12a). SRTF would perform as in Figure 12b when $1.5 < x < 2$ and as in Figure 12c when $1 < x < 1.5$. This example demonstrates that even an optimal allocation, if it does not account for future job arrivals, can turn into a mediocre one. Thus, to design an algorithm that performs well in practice, one must incorporate the possibility of future job arrivals into her design.

B Impact of Resource Efficiency

To present how resource efficiency impacts the evaluation of (1) and (2), which decides which job (either j_a or j_b) gets the extra GPU, let us first define f_k as *fractional throughput gain* of j_k:

$$f_k = \frac{p'_{k,1}}{p_{k,1}}.$$

[14]The optimal online job scheduling is NP-complete [17]. A brute force search would require examining 10^{209} cases if we run 20 concurrent jobs on a 60-GPU cluster.



(a) Naïve. (b) SRTF, $1.5 < x < 2$. (c) SRTF, $1 < x < 1.5$.

Figure 12: Examples in which the optimal allocation of resource shares depends on the arrival pattern of future jobs. Dark gray, light gray, and white boxes indicate that the corresponding jobs use two, one, and zero GPUs, respectively.

This metric measures the marginal throughput j_k earns with respect to its current throughput when it is allocated an extra GPU. Intuitively, a larger value indicates a larger resource efficiency. This metric is constrained by $1 < f_k \leq 2$, as we consider regimes where an extra GPU increases training throughput and the throughput scales sublinearly to the allocated GPUs.

We rewrite (1) (j_b earns the extra GPU if it is positive) as follows:

$$\left(\frac{p_{b,1}}{p'_{b,2}}f_b - 2\right)f_a - \left(\frac{p_{b,1}}{p'_{b,2}} - 2\right). \qquad (4)$$

If no future job arrives, the share of an existing job would monotonically increase. Hence, $p_{b,1} \leq p'_{b,2}$ holds. Combined with $f_b \leq 2$, the terms in the parentheses are negative. This means that smaller f_a makes (4) more likely to be positive. Also, one can check that larger f_b makes (4) more likely to be positive. Put together, it concludes that j_b is more likely to get the extra GPU if its resource efficiency is relatively higher than that of j_a.

If future jobs do arrive, the share of an existing job may decrease (i.e., $p_{b,1} > p'_{b,2}$ may hold) to the point where (4) is always positive, hence j_b always gets the extra GPU. Aside from this case, similar arguments apply. Also, evaluating whether (2) is positive follows a similar line of arguments and leads to the same conclusion: resource efficiency matters.

C Transitivity of (3)

Since we use (3) as a job-to-job comparison function to select a single job j^* with the top priority in Algorithm 1, (3) should be transitive, otherwise we cannot guarantee that such j^* exists. As the transitivity of (3) looks non-trivial, we provide a simple proof here.

Problem. Given that:

$$\frac{p'_{b,1} - p_{b,1}}{p'_{b,1}} > \frac{p'_{a,1} - p_{a,1}}{p_{a,1}},$$

$$\frac{p'_{c,1} - p_{c,1}}{p'_{c,1}} > \frac{p'_{b,1} - p_{b,1}}{p_{b,1}},$$

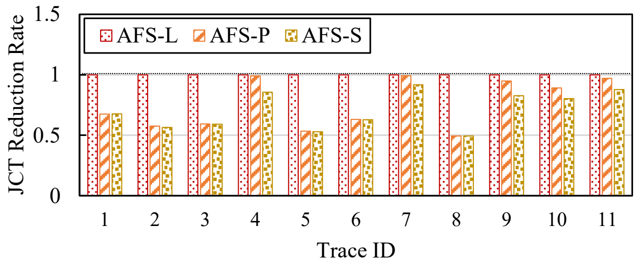


Figure 13: Rate of average JCT reduction from AFS-L with different scheduling algorithms using traces in Table 3 over a 64-GPU cluster.

prove:

$$\frac{p'_{c,1} - p_{c,1}}{p'_{c,1}} > \frac{p'_{a,1} - p_{a,1}}{p_{a,1}}.$$

Proof. As $p'_{b,1} \geq p_{b,1}$, the following holds true.

$$\frac{p'_{c,1} - p_{c,1}}{p'_{c,1}} > \frac{p'_{b,1} - p_{b,1}}{p_{b,1}} \geq \frac{p'_{b,1} - p_{b,1}}{p'_{b,1}} > \frac{p'_{a,1} - p_{a,1}}{p_{a,1}}.$$

□

D Discussion on AFS

AFS-L vs. AFS-P. Figure 13 compares the performance of AFS-L and AFS-P. We observe that AFS-P performs especially worse than AFS-L when there are many unfinished jobs in the cluster, which makes it more difficult for processor sharing of AFS-P to mimic the SRTF-like behavior of AFS-L. However, even in such cases, the average JCT blowup of AFS-P is curbed at twice the value from AFS-L in our experiments. It still outperforms other existing algorithms as demonstrated in Section 5.

Sensitivity to throughput measurement. Unlike legacy scheduling algorithms, AFS-P relies on throughput measurement to determine the resource shares. To evaluate the performance sensitivity to throughput measurement, we design AFS-S as a proof-of-concept algorithm. AFS-S is the same as AFS-P except that it only obtains the number of GPUs that lead to the maximum throughput for a job, rather than precisely estimating the throughput in relation to the number of GPUs. As it does not evaluate job throughputs, AFS-S ends up distributing the resources by the max-min fairness rather than enforcing fine-grained resource distribution of AFS-P. Figure 13 shows that AFS-S performs similarly to AFS-P in many cases. This implies that the performance of AFS-P is insensitive to detailed throughput measurement, rather contingent on figuring out the number of GPUs that gives rise to the highest performance.

Asynchronous training. As CoDDL automatically scales a DLT job instead of letting users do it manually, we need to select a default strategy for distributed training. We select bulk-synchronous-parallel (BSP) training over asynchronous training because its system throughput is equal to algorithmic training throughput. With asynchronous training, it is common that the system throughput increases linearly while the algorithmic training throughput does not, partly due to exchanging outdated parameter updates. This can be viewed as wasting GPU resources (requiring extra training iterations), similar to BSP's wasting GPU time on GPU-to-GPU communication. However, in case of asynchronous training, such a waste is difficult to be detected by the system, as it additionally requires evaluating algorithmic training performance (i.e. how fast training and/or evaluation accuracy improves, what is the final accuracy of the trained model, etc.). Optimus [38] suggests a method for measuring algorithmic performance by predicting the training loss curve, but it is unclear whether predicting the loss curve is always possible in theory, as is also questioned by a follow-up work [24].

E Removing Network Sharing

This section provides the details and the proof of the guarantee mentioned in Section 4.4 – the network packing regulation removes network sharing across the cluster, while providing a small constant bound of the difference from the originally-determined share to the regulated share for all jobs. We formulate the problem as follows. We assume a homogeneous GPU cluster where each machine is equipped with 2^k GPUs ($k \geq 0$). Given n jobs to share m machines ($n > 0$ and $m > 0$), say the scheduling algorithm originally assigns s_i GPUs to job i ($i \in \{1, \cdots, n\}$), where $\sum_{i=1}^{n} s_i = 2^k m$. Then, the controller applies the network packing regulation, adapting s_i to r_i that requires r_i to be one of zero, 2^ℓ ($0 \leq \ell < k$), or a multiple of 2^k, for all i. We prove two propositions as follow.

Proposition 1: removal of network sharing. Given r_i GPUs to job i for all i, there exists at least one GPU placement decision that the following conditions hold true. First, for all i, job i uses exactly $\lceil r_i/2^k \rceil$ machines, which is the minimum number required. Second, at most one job on any machine uses two or more machines.

Proof. We describe one of the feasible placement algorithms that makes this proposition hold true. Initially, no machines are assigned to any jobs. First of all, we discard all jobs that $r_i = 0$. For all i where r_i is a multiple of 2^k, assign $r_i/2^k$ machines to job i. Then, all those jobs and assigned machines already satisfy the two conditions of the proposition, so now we only consider the remaining jobs and machines. Since all remaining jobs are assigned less than 2^k GPUs each, to satisfy the first condition, we need to let each remaining job use only GPUs on the same machine – this is a bin-packing problem where each machine is a bin of size 2^k and each job is an item of size r_i. Fortunately, since r_i of all remaining jobs is a power of two less than 2^k, the bin size is always divisible by the item size, thus the simple first-fit-decreasing algorithm can always fit all remaining jobs to the remaining machines. This also satisfies the second condition at the same time because all remaining jobs do not use inter-machine networking and all remaining machines do not run any multi-machine jobs. □

Proposition 2: feasibility and difference bound. Regardless of the value of $\{s_1, \cdots, s_n\}$, there always exists at least one $R = \{r_1, \cdots, r_n\}$ that the following conditions hold true. First, the total number of assigned GPUs do not change due to the regulation, i.e. $\sum_{i=1}^{n} r_i = \sum_{i=1}^{n} s_i = 2^k m$.[15] Second, for all i, we guarantee the minimum value of r_i, say x_i, which is as follows:

$$r_i \geq x_i = \begin{cases} \left\lfloor \frac{s_i}{2^k} \right\rfloor 2^k, & \text{if } s_i > 2^k, \\ 2^{\lfloor \log_2 s_i \rfloor}, & \text{if } 0 < s_i \leq 2^k, \\ 0, & \text{if } s_i = 0. \end{cases}$$

Note that x_i is the maximum value that satisfies the network packing regulation while not exceeding s_i.

Proof. Say that we assign x_i GPUs to all job i, i.e. $r_i = x_i$. Then, R satisfies both the network packing regulation and the second condition of this proposition. However, since $x_i \leq s_i$, this may not satisfy the first condition of this proposition, i.e. this may remain unassigned idle GPUs and the number is $d = \sum_{i=1}^{n} s_i - \sum_{i=1}^{n} r_i$. Thus, we will prove that in any cases where $d > 0$, we can always reduce d by assigning more GPUs while always satisfying the network packing regulation and the second condition of this proposition.

If $d \geq 2^k$, we can reduce d to be lower than 2^k using two methods. First, assign 2^k more GPUs to any job i with $r_i \geq 2^k$. Since r_i is already regulated by the network packing (i.e. it is a multiple of 2^k), adding 2^k more GPUs to r_i will keep it to be a multiple of 2^k. Second, assign more GPUs to any job i with $r_i = 2^\ell$ $(0 \leq \ell < k)$ and $d \geq 2^\ell$ so that r_i increases to $2^{\ell+1}$. Note that at least one of these two methods is always usable unless d becomes less than 2^k.

If $d < 2^k$, we can only use the second method aforementioned to reduce d. Note that we cannot use the second method if $r_i > d$ for all job i. Thus, we need to prove that d becomes zero if $r_i > d$ for all job i. Actually, it is self-conflicted if we say $d > 0$ in that case. The total number of GPUs is $d + \sum_{i=1}^{n} r_i$ and it should be equal to $2^k m$, but it cannot be a multiple of 2^k if $0 < d < r_i$ for all job i because r_i itself is a multiple of 2^k or a power of 2 less than 2^k. □

ID	Virtual Cluster ID	#Machines	#GPUs	#Jobs
1	`0e4a51`	398	1846	2465
2	`6c71a0`	409	1856	14791
3	`b436b2`	387	1668	9033
4	`e13805`	389	1750	938
5	`6214e9`	412	1868	51288
6	`7f04ca`	389	1714	1461
7	`103959`	226	470	2677
8	`ee9e8c`	412	1868	5781
9	`2869ce`	384	1668	956
10	`11cb48`	408	1860	19070
11	`ed69ec`	301	1454	1401

Table 3: Summary of Philly DNN workload traces [5, 27]. Jobs which have incomplete information such as finished time or requested GPU count are excluded. Among 15 total traces, 4 traces which contain few jobs (less than 100) are excluded, so we make use of 11 remaining traces.

Name	Dataset	Batch Size	Max #GPUs
VGG16 [47]	ImageNet [14]	256	8
GoogLeNet [49]	ImageNet [14]	128	20
Inception-v4 [50]	ImageNet [14]	256	52
ResNet-50 [25]	ImageNet [14]	128	28
DCGAN [41]	Celeb-A [33]	256	20
Video Prediction [15]	Push [15]	64	28
Chatbot [54]	OpenSubtitles [52]	256	4
Deep Speech 2 [7]	LibriSpeech ASR corpus [37]	64	20
Transformer [53]	IWSLT 2016 English-German corpus [10]	256	44

Table 4: Description of real-world DL models for experiments.

F DLT Workload Details

All experiments carried out in this paper use the Philly DNN workload [5, 27], 137-day real-world DNN traces from Microsoft. It consists of 15 traces from different virtual clusters, and we use 11 of them as shown in Table 3 that contain 100 or more jobs. From the traces, we use submission time, elapsed time, and requested (allocated) numbers of GPUs of each job. Figure 14 shows that the DLT jobs in the workload tend to be heavy-tailed.

Since the traces do not carry training model information, we submit a random model chosen from a pool of nine popular DL models shown in Table 4 whose training throughput scales up to the requested number of GPUs. Each DLT job submits the chosen model for training in the BSP manner with the same batch size throughout training, even though the GPU share changes during the training. The number of training iterations of the job is calculated so that its completion time becomes equal to the total executed time from the trace, based on the use of the requested number of GPUs.

[15]This condition is needed to make sure that the regulation avoids embarrassingly inefficient adaptation that makes extra idle GPUs.

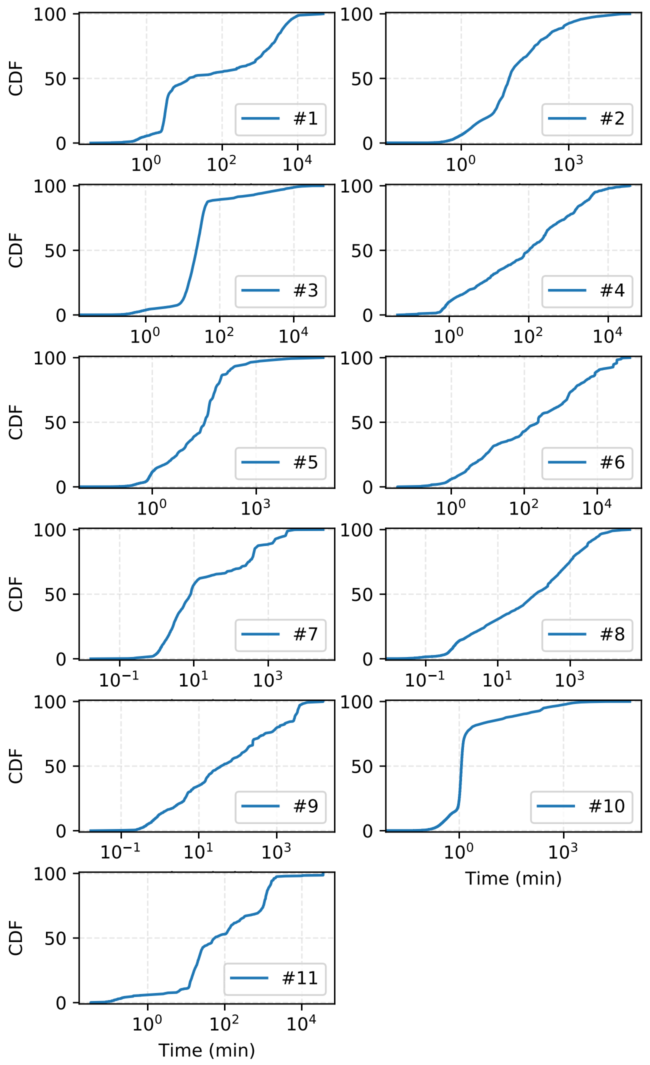
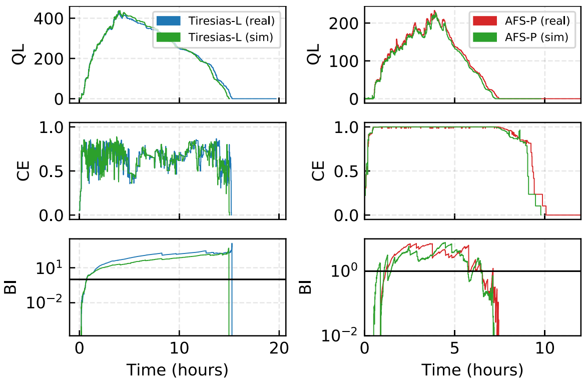


Figure 14: CDF of job sojourn time of traces in Table 3. X-axis is log-scaled and the unit is minutes.



(a) Tiresias-L. **(b)** AFS-P.

Figure 15: Comparison of our simulator (sim) with real execution (real) for (a) Tiresias-L and (b) AFS-P with trace #3 (down-scaled to 0.2%) in Table 3 on our 64-GPU cluster. Average JCTs are (a) sim: 5.62 and real: 5.65 hours (0.5% error) (b) sim: 1.83 and real: 1.93 hours (5.2% error), respectively.

ATP: In-network Aggregation for Multi-tenant Learning

ChonLam Lao ††, Yanfang Le†, Kshiteej Mahajan†, Yixi Chen††,
Wenfei Wu††, Aditya Akella†, Michael Swift†*

Tsinghua University ††, University of Wisconsin-Madison †

Abstract

Distributed deep neural network training (DT) systems are widely deployed in clusters where the network is shared across multiple tenants, i.e., multiple DT jobs. Each DT job computes and aggregates gradients. Recent advances in hardware accelerators have shifted the the performance bottleneck of training from computation to communication. To speed up DT jobs' communication, we propose ATP, a service for in-network aggregation aimed at modern multi-rack, multi-job DT settings. ATP uses emerging programmable switch hardware to support in-network aggregation at multiple rack switches in a cluster to speedup DT jobs. ATP performs *decentralized, dynamic, best-effort* aggregation, enables efficient and equitable sharing of limited switch resources across simultaneously running DT jobs, and gracefully accommodates heavy contention for switch resources. ATP outperforms existing systems accelerating training throughput by up to 38% - 66% in a cluster shared by multiple DT jobs.

1 Introduction

Traditional network design relied on the end-to-end principle to guide functionality placement, leaving only common needs implemented within the network, primarily routing and forwarding. However, datacenter networks and workloads have evolved, and there is a strong case to support common application functionality within the network [22, 41, 71].

Deep Neural Networks (DNN) are emerging as a critical component of more and more enterprise applications such as computer vision [33], natural language processing [26, 67], databases [65], compilers [66], systems [68], and networking [54]. These applications all require distributed DNN training (DT) to iteratively train better DNNs for improved prediction performance. Enterprises typically run DT on multi-rack clusters [12] shared by other applications. Each DT job has several workers and parameter servers (PS) spread across several machines. Workers compute gradients and send these gradients to the PS(s) over the network for aggregation. *Gradient aggregation*, which combines partial results from multiple workers and returns a single aggregated result, is commonly used in DT, and contributes substantially to overall training time [48]. Recent advances in special hardware [6, 12] have shifted the performance bottleneck of distributed training from computation to communication [48, 56]: VGG16 training can be $4X$ faster without network communication [56].

Further, datacenter networks are becoming feature-rich with the introduction of new classes of programmable network devices such as programmable switches (e.g., Intel's FlexPipe [8], Cavium's XPliant [13], Barefoot Tofino [4]) and network accelerators (e.g., Cavium's OCTEON and LiquidIO products [9], Netronome's NFP-6000 [10], and FlexNIC [43]). Together, they offer in-transit packet processing and in-network state that can be used for *application-level stateful computation* as data flows through the network.

Current DT stacks implement gradient aggregation purely in the application. However, the emergence of DT as a common application and its reliance on gradient aggregation, as well as the emergence of application-level stateful computation as a network feature, suggests an opportunity to reduce training time by moving gradient aggregation inside the network. This reduces network bandwidth consumption from workers to the PS(s). For both single DT and multiple DT jobs (i.e., multi-tenant settings) this bandwidth allows pushing more gradients through the network, and increases the total throughput of gradient flows thereby reducing training times.

Recent proposals show the initial promise of such in-network aggregation: e.g., SwitchML [56] increases training throughput for VGG16 by $2X$ via in-network aggregation on a programmable top-of-rack switch. However, the general problem of making aggregation a true in-network service to be leveraged by multiple DT tenants in a multi-rack/multi-switch cluster has not received systematic attention. Realizing such a service calls for mechanisms to share limited multi-switch aggregation resources across multiple tenants.

The key goal of our work is to speed up multiple DT jobs running simultaneously in a cluster by maximizing the benefits from in-network multi-switch aggregation, and distributing these benefits across multiple DT jobs in an equitable manner. To do so, we propose a new network service for multi-rack clusters called Aggregation Transmission Protocol, i.e., ATP. ATP supports *dynamic aggregation* at rack switches. DT jobs go through 'on' and 'off' gradient aggregation phases, and ATP uses decentralized mechanisms to ensure that switch resources used by a DT job entering its off phase can be dynamically reused by a DT job in its on phase. ATP supports *best-effort* aggregation. This enables DT jobs to gracefully fall back to end-host aggregation under heavy contention from many tenants without extra overhead.

*ChonLam Lao and Yanfang Le are co-primary authors, and Wenfei Wu is the corresponding author.

ATP chunks gradients for each DT job into fixed size fragments that we refer to as *gradient fragment packets* and partitions programmable switch resources into the same fixed size fragments called *aggregators*. As these gradient fragment packets flow through the network, ATP opportunistically aggregates them by accumulating results at the earliest available programmable switch, or in the worst-case at the PS end-host.

ATP proposes a decentralized aggregator allocation mechanism that supports aggregation at line rate for multiple jobs by dynamically allocating free aggregators when gradient fragment packets arrive at a switch. A key issue with an in-network aggregation service is that traditional end-to-end protocols do not work when gradient fragment packets are consumed in the network due to aggregation, as that may be misinterpreted as packet loss. Thus, ATP co-designs the switch logic and end host networking stack specifically to support reliability and effective congestion control.

We opensource ATP's implementation [2]. Our implementation works atop clusters using P4-programmable switches. Such switches expose a limited set of in-network packet processing primitives, place ungenerous memory limits on network state, and have a constrained memory model restricting reads/writes. We overcome these constraints, and show how ATP can support highly effective dynamic, best-effort aggregation that can achieve 60Gbps. Our implementation also has mechanisms that improve state-of-the-art floating point value quantization to support limited switch computation. ATP's implementation adopts a kernel bypass design at the end-host so that existing protocol stacks are not replaced by ATP's network stack and non-ATP applications can continue to use existing protocol stacks.

We run extensive experiments on popular DNN models to evaluate ATP in a single rack testbed with multiple jobs. Our evaluation shows that in multi-tenant scenarios, dynamic, best-effort in-network aggregation with ATP enables efficient switch resource usage. For example, the performance only decreases by $5-10\%$ when only half of the desired aggregators are available, and outperforms current state-of-the-art by 38% when there is heavy contention for on-switch resources. We simulate multi-rack cluster experiments with a typical topology and show a 66% reduction in network traffic with ATP. We benchmark loss-recovery and congestion control algorithms proposed in ATP. The loss recovery mechanism of ATP outperforms the state-of-the-art (SwitchML) by 34% and an ATP job with congestion control speeds up $3X$ compared to one without congestion control.

2 Background and Motivation

2.1 Preliminaries

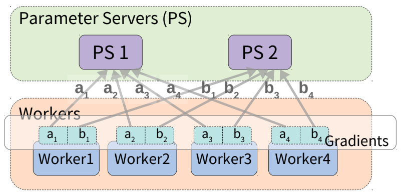
PS Architecture. This design [39, 51, 62] as shown in Figure 1 enables data-parallel training, where training data is partitioned and distributed to workers. There are two phases: gradient computation, where workers locally compute gradients; and gradient aggregation, where workers' gradients are transmitted over the network to be aggregated (which involves the addition of gradients) at one or more end-hosts called parameter servers (PSs). The aggregated parameters are then sent back to the workers. Gradients are tensors, i.e., arrays of values. With multiple PSs, each PS has a distinct partition of parameters.

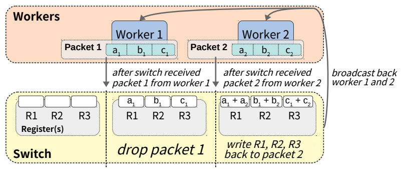
Programmable Switch. The recent emergence of programmable switches provides opportunities to offload application-level stateful computation [41, 47, 71]. A popular example is the Tofino switch [4], which we use. Programmable switches expose memory as stateful and stateless objects. Stateless objects, *metadata*, hold the transient state for each packet, and the switch releases this object when that packet is dropped or forwarded. Stateful objects, *registers*, hold state as long as the switch program is running. A *register* value can be read and written in the dataplane, but can only be accessed once, either for read or write or both, for each packet. A register is an array of values. In the context of in-network aggregation, each packet has a subset of gradient values and needs a set of registers to aggregate them. We call this set of registers an *aggregator*.

Programmable switches have constrained compute resources, memory($\sim$ 10MB [53]), and programmability for application-level processing. Register memory can only be allocated when the switch program launches. To change memory allocation, users have to stop the switch, modify the switch program and restart the switch program. The computation flexibility is limited by the number of stages, the payload parsing capability, and the time budget at each stage: only independent computation primitives can be placed in the same stage and the number of register accessed in the same stage is also limited. These limits lead to small packet sizes for in-network computation and storage applications: the payload size of SwitchML and NetCache is 128B [40, 41, 46, 56] [1].

In-Network Aggregation. Gradients can be seen as a sequence of fragments (each fragment has a subset of gradient values), and aggregation (addition of gradients) of all the gradients is the aggregation of each of these fragments. In-network aggregation for each fragment is done in a specific aggregator. Figure 2 exemplifies this for a DT job with two workers using one programmable switch. Workers 1 and 2 create packets having a fragment with 3 tensor values and send them to the switch. Suppose the switch first receives the packet $p1$ from worker 1. It stores the tensor values contained in $p1$ in the aggregator's registers $R1$, $R2$, $R3$. The switch then drops packet $p1$. When the switch then receives packet $p2$ from worker 2, it aggregates the tensor values contained in $p2$ with contents of $R1$, $R2$, $R3$. If there were additional workers, the switch would update the registers with the aggregation of both packets. In this example, because $p2$ is from

[1]The exact parameters of programmable switches and ATP are specific to "Tofino" programmable switches; if other programmable switches have similar limitations, ATP can be used similarly.


Parameter Servers (PS)
PS 1
PS 2
a_1 a_2 a_3 a_4 b_1 b_2 b_3 b_4
Workers
a_1 b_1 a_2 b_2 a_3 b_3 a_4 b_4
Gradients
Worker1
Worker2
Worker3
Worker4
Figure 1: Parameter servers (PS)


Workers
Worker 1
Worker 2
Packet 1 a_1 b_1 c_1
Packet 2 a_2 b_2 c_2
after switch received packet 1 from worker 1
after switch received packet 2 from worker 2
broadcast back worker 1 and 2
R1 R2 R3
Register(s)
a_1 b_1 c_1
$a_1 + a_2$ $b_1 + b_2$ $c_1 + c_2$
Switch
drop packet 1
write R1, R2, R3 back to packet 2
Figure 2: In-network aggregation example

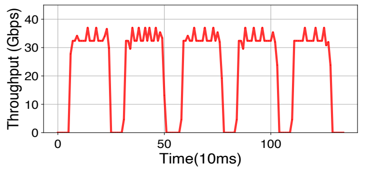


Figure 3: A DT job training VGG16 shows on-off communication pattern for a simple one worker-one PS setting.

the last worker, the switch overwrites the values in packet $p2$ with the aggregated result and multicasts $p2$ to both the workers. This architectural improvement not only reduces network traffic and eliminates the incast but also saves the CPU cycles used for aggregation operation at the end hosts. As this improvement only applies to communication, the *overall* training acceleration ratio depends specifically on the ratio of communication to computation in the DT job [28, 56].

A recent work, SwitchML [56], prototypes this idea for a single DT job in a rack-scale network. We use SwitchML as an example to illustrate the design space and underscore the key attributes of an ideal in-network aggregation service. SwitchML removes the PS by offloading gradient aggregation *entirely* to the top-of-rack switch. It allocates a static pool of aggregators in the rack switch to a DT job, and streams gradient fragment(s) from workers to the switch only after previously sent gradient fragment(s) are aggregated and have vacated aggregator(s) on the switch. We argue next that design choices in SwitchML need to be reconsidered in the multi-job and multi-rack settings, necessitating a systematic in-network service.

2.2 In-Network Aggregation as a Service

When applied to multiple DT jobs, SwitchML requires static partitioning of switch resources, where each job is statically assigned to a partition. In a multi-tenant scenario, this results in underutilization of switch resources. DT jobs go through on and off gradient aggregation phases as shown in Figure 3, and switch resources belonging to a DT job in the off phase can be shared with a DT job in the on phase in a *dynamic manner*, but static partitioning precludes this.

SwitchML offloads gradient aggregation for each DT job entirely to the rack switch. With heavy switch resource contention, DT jobs have to wait for switch resources leading to underutilization of the network link bandwidth from the workers to the PS(s). In a better design, a DT job could instead aggregate a fraction of gradients at the switch in a *best-effort manner* while aggregating the rest at the end-host.

Rack-scale solutions like SwitchML limit job scalibility and are not optimal in terms of traffic reduction for cross-rack jobs. Enabling aggregation service at every layer of the network topology complicates the service design and the network operation. ATP balances complexity and performance by enabling aggregation at the workers' and PS's ToR switches.

Thus, in the context of multi-job and multi-rack, an ideal in-network aggregation service should support *dynamic, best-effort, multi-rack* gradient aggregation for optimal efficiency and speedup. As we show in Section 3, realizing such an in-network aggregation service requires key innovations at end-hosts and in how switch resources are apportioned and dynamically (re)used. In addition, an in-network aggregation service brings to fore two other aspects of the network stack that need redesign, namely, reliability and congestion control.

Rethinking Reliability. In-network aggregation breaks end-to-end semantics as some packets are consumed inside the network during aggregation. Traditional end-host based reliability mechanisms can misinterpret in-network packet consumption as a packet loss, leading to unnecessary retransmissions and lead to incorrect gradient aggregation due to the inability of existing reliability mechanisms in dealing with these new class of packet events. Thus, we need a new *reliability algorithm* to deal with this new class of packet events.

Rethinking Congestion-Control. In the multi-tenant case, the network resources (switch aggregators and network bandwidth) available to a DT job fluctuates because (1) DT jobs exhibit on-off communication phases (Figure 3), (2) the total number of DT jobs varies, and (3) background traffic varies. Utilizing fluctuating network resources efficiently and sharing them fairly depends on congestion control. However, as end-to-end semantics are broken we cannot use traditional congestion control algorithms that rely on RTT or drops as the congestion signal. We need a new *congestion control algorithm* that identifies the right congestion signal so as to modulate the throughput of gradient fragments from workers' for each DT job to meet the requirements of efficient use and fair division of network resources across DT jobs.

3 Design

ATP is a network service that performs *dynamic, best-effort aggregation* across DT jobs. ATP's design aligns with guidelines for building robust and deployable in-network computation [53]: (1) offload reusable primitives: ATP is a network service for in-network aggregation and a common function to different DT frameworks; (2) preserve fate sharing: ATP is able to progress in the event of network device failure via fallback to aggregation at the end-host; (3) keep state out of the network: ATP's end-host reliability algorithms are able to recover lost data and deal with partial aggregation; (4) minimal interference: ATP chooses aggregation only at Top-of-Rack (ToR) switches to sidestep issues owing to probabilistic routing in the network.

3.1 ATP Overview

ATP lies in the transport layer which specifically targets in-network aggregation of gradient tensors in DT applications; it is not a general-purpose transport. Compared to general-purpose TCP: (a) ATP redesigns specific transport features, such as reliability, congestion control, and flow control for its target context. (b) ATP does not implement TCP's in-order byte-stream and multiplexing abstractions as they do not apply to the target context.

ATP performs aggregation at the granularity of fragments of a gradient that fit in a single packet, i.e., *gradient fragment packets*. ATP chunks the gradient tensor at each worker into a sequence of fixed-size fragments such that each fragment fits in a packet and assigns each a sequence number. Gradient aggregation for a DT job merges values at the same sequence number from each worker's tensor.

Upon booting, each ATP programmable switch allocates a portion of switch register memory to be shared by ATP jobs. This memory is organized as an array of fixed-size segments, which we refer to as *gradient fragment aggregators*, or just *aggregators*. Each aggregator is accessed by its index in the array, and aggregates gradient packets with a specific sequence number belonging to the same DT job.

ATP workers stream gradient fragment packets to the PS(s)[2]. ATP aggregates gradient fragment packets inside the network when in-network resources are available. If in-network resources are unavailable, gradient fragment packets are sent to the end-host PS for aggregation. ATP restricts in-network aggregation to ToR programmable switches. This means that gradients from each worker can at most be aggregated at two levels – (1) the rack switch at the worker and (2) the rack switch at the PS. This requires coordination of decisions to ensure that each gradient fragment packet is aggregated *exactly once*. We use a decentralized, dynamic, best-effort approach to determine where aggregation occurs.

Gradient fragment packets contains *direction* fields. These directions interact with the ATP switch logic at the programmable switches, to program *soft-state* in the aggregator to elicit a coordinated decision. The aggregator soft-state can be discarded at any time, leading to aggregation at the PS instead of the switch. The directions in a gradient fragment packet comprise fields that help switches decide whether to aggregate the packet, in which gradient aggregator to aggregate, and to identify completion or failure of aggregation at an aggregator. Switch logic uses these directions to program soft-state in the switch that identifies whether a gradient aggregator already exists for an incoming gradient fragment, and keeps track of intermediate aggregation results and completion of aggregation.

Soft-state in switches and directions in packets ensure that

[2]Note that two gradients from different workers that will be aggregated never meet at switches in ring all-reduce architecture [58]. To the best of our knowledge, any in-network aggregation, as well as ATP, can not apply to ring all-reduce architecture.

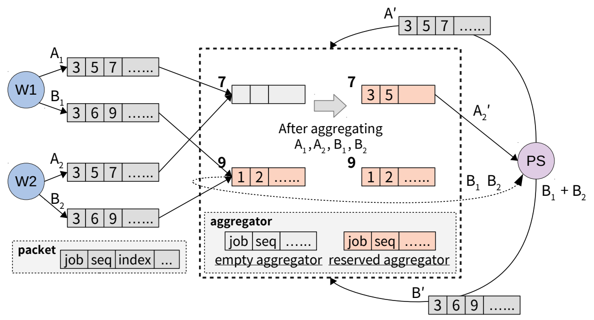


Figure 4: ATP dynamic, best-effort aggregation example. The *directions* fields are *job ID, sequence and aggregator index* in packet. *soft-state* is the values in the aggregators.

ATP does not require job-specific switch program changes (and avoids switch restarts) upon job arrival/departure.

Figure 4 exemplifies how ATP achieves dynamic, best-effort aggregation. A job with ID 3 has two workers, $w1$ and $w2$. The workers compute gradients which are subsequently broken by ATP at end hosts into two packets each - (A_1, B_1), and (A_2, B_2). ATP aggregates gradient packets A_1 with A_2, and B_1 with B_2, either at the switch or at the PS, as explained next. Packets A_1 and A_2 are routed and hashed to aggregator 7; since the aggregator is empty, it is "reserved" by packet A_1 by changing the aggregator's *soft-state* to its job ID and packet sequence. When A_2 arrives at the switch, it hashes to the same aggregator and triggers aggregation; then, the resulting packet containing the aggregation result, A_2', is sent to the PS. In contrast, packet B_1 can not reserve aggregator 9, because it is reserved by a packet with job ID 1 and sequence 2. Thus, packet B_1 is forwarded directly to the PS; the same occurs with B_2. Packets B_1 and B_2 are aggregated at the PS. For either pair of packets, the PS sends the parameter packets (A' and B') via multicast back to workers $w1$ and $w2$. When the switch receives A', aggregator 7 is deallocated and set as empty (i.e., A' is hashed to aggregator 7, and the aggregator's job ID and sequence match with those in A') to enable aggregator 7 to be used by future fragments from another job.

To detect and deal with packet losses ATP uses time-out based retransmission or out-of-sequence parameter ACK packets from the PS. When a packet is retransmitted, it sets the resend flag. This serves as a direction for the switch to deallocate and transmit any partially allocated result to the PS. Also, to deal with congestion, say if queue depth is above a certain threshold when packet A_2 is received, an ECN flag in A_2 is set and carried over to A_2'. This is copied to the parameter packet A' in PS and received by the workers who adjust their windows. The window adjustment is synchronized in both the workers as it is triggered by the same ECN bit in A'.

3.2 ATP Infrastructure Setup

ATP requires a one-time static setup involving programming and restarting switches to bring the service up. Any dynamic per-job setup is managed by inserting the appropriate job-specific directions in gradient fragment packets.

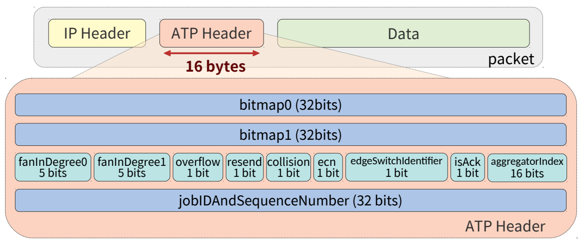


Figure 5: ATP packet format.

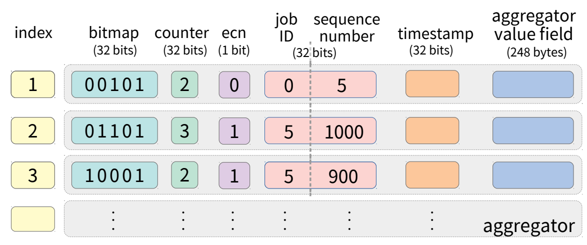


Figure 6: ATP Switch memory layout.

Static Infrastructure Setup. The infrastructure, comprising the switches and the end-host networking stack, is configured once to serve all ATP jobs. Each programmable switch installs a classifier to identify ATP traffic—gradient and parameter packets—and allocates a portion of switch resources—aggregators—to aggregate ATP traffic. The end host installs an ATP networking stack, which intercepts all the push or pull gradient calls from DT jobs. End-hosts have knowledge of the network topology—switch, end-host port connectivity, and total number of aggregators at a switch—so they can orchestrate aggregation across multiple switches.

Dynamic Per-Job Setup. Each new DT job is assigned a unique job ID. The job assigns each worker an ID from 1 to W, where W is the total number of workers. The job tracks the location of workers in the network topology to build an aggregation hierarchy. In case workers and PS are spread across racks, the job can use multiple switches for in-network aggregation. The ATP networking library computes the job's worker fan-in at each level of the aggregation hierarchy, which is used to determine when aggregation is complete (§3.5). ATP uses IGMP to build a multicast distribution tree for the PS to return parameters to workers.

3.3 Data Structures

Packet Format. Figure 5 shows the gradient fragment packet format. The ATP header fields comprise directions and contain metadata about the fragment. The `jobIDAndSequenceNumber` field is the identifier of a packet and is used to match gradient packets from different workers on the same job. The `Data` field contains tensor values (or aggregated tensor values).

One-hot encoding is used to identify the worker's position in the aggregation hierarchy (`bitmap0`), and the first-level switch's position at the second edge switch (`bitmap1`). The fan-in degree indicates the number of workers attached to the first edge switch (`fanInDegree0`) and workers or switches attached to the second edge switch (`fanInDegree1`). These four fields are used to determine when aggregation has completed (§3.5). The `edgeSwitchIdentifier` flag is set to 0 if the packet is en-route to the first edge switch in the aggregation hierarchy and 1 if the packet is en-route to the second edge switch.

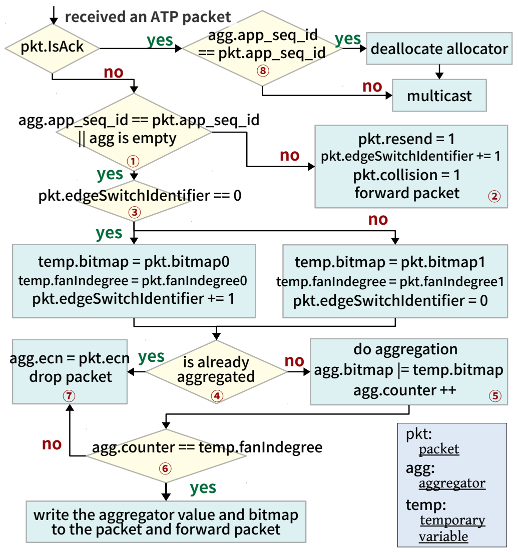
Workers detect dropped packets when they receive out-of-order parameter packets, which triggers them to resend gradient packets for aggregation (§3.7). The `resend` flag is set if it is a retransmitted gradient packet. The `ECN` flag is marked by a switch when the switch's output queue length exceeds some threshold, which is used for detecting network congestion. The `collision` flag is marked by a switch when it forwards a gradient packet onward due to the aggregator not being available because it is in use by a different job. This flag helps PS choose another aggregator to avoid collision in the next round.

Parameter packets use the same packet format, but indicate the different contents by setting the `isAck` flag. They are multicast from the switches to workers when an aggregation is complete, and serve as acknowledgments (ACKs) for the gradient packets sent by the workers.

Switch Memory. Figure 6 shows the switch memory layout. Switch memory is organized as an array of fixed-size aggregators, each accessed by its index in the array. The `value` field contains aggregated data from different workers. The size of the value field is the same as that of a gradient fragment value. The `bitmap` field records which workers have already been aggregated to the aggregator's value field. The `counter` field records the number of distinct workers included in the aggregated value. The `ECN` field records congestion status and is set if any aggregated packet had the `ECN` flag set. The `timestamp` field is updated when an aggregation is performed, and is used to detect when an aggregator has been abandoned (e.g., when all workers fail) and can be deallocated (§3.7). The identifier fields <`Job ID, Sequence Number`> uniquely identify the job and the fragment that this aggregator serves.

3.4 Inter-rack Aggregation

Scaling aggregation beyond a single rack provides more flexibility w.r.t. where DT executes in a cluster. Aggregating just at a worker's local ToR switch is simple, but leads to unnecessary network traffic to the PS when workers reside in different racks. Alternatively, aggregation can be done at higher layers of the network topology. However, this approach would greatly increase protocol complexity because the system has to handle route changes in the interior of the network. For example, ECMP-based routing can change the number of gradient streams incident at a particular switch in the interior of the network. This necessitates careful coordination between network routing and the aggregator allocation mechanism. Thus, ATP only deploys in-network aggregation in ToR switches, either at the worker's rack (first-level) or at the PS's rack (second-level). This complies with a recent study which shows that programmable switches today are



received an ATP packet
pkt.IsAck
yes
no
agg.app_seq_id == pkt.app_seq_id ⑧
yes
deallocate allocator
no
multicast
agg.app_seq_id == pkt.app_seq_id || agg is empty ①
no
pkt.resend = 1
pkt.edgeSwitchIdentifier += 1
pkt.collision = 1
forward packet ②
yes
pkt.edgeSwitchIdentifier == 0 ③
yes
no
temp.bitmap = pkt.bitmap0
temp.fanIndegree = pkt.fanIndegree0
pkt.edgeSwitchIdentifier += 1
temp.bitmap = pkt.bitmap1
temp.fanIndegree = pkt.fanIndegree1
pkt.edgeSwitchIdentifier = 0
agg.ecn = pkt.ecn
drop packet ⑦
yes
is already aggregated ④
no
do aggregation
agg.bitmap |= temp.bitmap
agg.counter ++ ⑤
no
agg.counter == temp.fanIndegree ⑥
yes
write the aggregator value and bitmap to the packet and forward packet
pkt: packet
agg: aggregator
temp: temporary variable

Figure 7: Pseudocode of the switch logic in the ideal case.

usually deployed at ToR switches for near-server computation offloading [27].

To coordinate where aggregation occurs, ATP uses two groups of `bitmap` and `fanInDegree` fields in gradient packets, i.e., `bitmap0/1` and `fanInDegree0/1` as shown in Figure 5. The `edgeSwitchIdentifier` field indicates which bitmap and degree a switch should use when processing the gradient packet, the first or second level of aggregation. When a first-level aggregation switch forwards a packet, it sets the `edgeSwitchIdentifier` bit in the packet header. The bitmap size limits the total number of workers ATP can support. As our testbed switch supports only 32-bit values, our implementation can support up to 1024 (=32×32) hosts; in general if the programmable switch can support n-bit values, ATP can support up to n^2 workers.

It is worth noting that ATP's multi-rack aggregation can be extended to more levels as long as the aggregation point is the fixed waypoint in the routing. As ATP's switches in two levels behave differently when handling packet retransmission (§3.7), switches whose level is higher than two should follow the logic in the second level. More details can be found in §A.2.

3.5 Switch Logic

Switch logic implements the algorithm that supports the *dynamic, best-effort* in-network aggregation service. It provides aggregator allocation, deallocation, and gradient aggregation. The allocation policy in ATP is First-Come-First-Serve without preemption: if a gradient fragment packet can reserve an aggregator, it keeps the aggregator until it is deallocated. In the ideal case, each edge switch completes aggregation of all incident worker gradient packets and sends the aggregated result downstream. We describe how failure cases (e.g., packet loss) are handled in §3.7. A detailed flowchart outlining switch logic can be found in Figure 7.

Aggregator Allocation. The arrival of a gradient fragment packet triggers aggregator allocation. When a gradient fragment packet arrives, the switch checks the availability of the aggregator at the packet's `aggregatorIndex` field. End-hosts compute `aggregatorIndex` as HASH(<`Job ID, Sequence Number`>)%`numAggregators`, which is consistent across all workers in a job. However, hash collisions could cause `aggregatorIndex` from different fragments and different jobs to map to the same index: e.g., in Figure 4, gradient fragment packets of Jobs 1 and 3 collide at index 9.

If the identifier field <`Job ID, Sequence Number`> in the aggregator is empty, we store the packet's identifier in the aggregator, and copy the gradient packet's `data` field into the aggregators `value` field. The switch copies the bitmap field in the aggregator from the appropriate bitmap field in the packet (`bitmap0` if `edgeSwitchIdentifier` is 0, else `bitmap1`), and initializes the counter to 1.

If the aggregator's identifier field is non-empty, the switch compares it to packet's identifiers (box ① in Figure 7). If they are different, there is a hash collision and ATP pushes the gradient fragment packet downstream to be processed at the PS. To avoid aggregation on this packet at the downstream switch and propagate the collision information to the PS, ATP sets `resend` and `collision` flags. To indicate this packet came from a switch and not a worker, it flips the edge switch identifier in the packet (② in Figure 7) as well. If the aggregator and packet identifiers are equal, gradient aggregation occurs.

Gradient Aggregation. If an aggregator is available for a gradient fragment packet, ATP uses the `edgeSwitchIdentifier` to fetch the fan-in degree and the bitmap for this switch from the packet (③ in Figure 7). Then, ATP checks whether this packet has been aggregated by testing the packet's bitmap against the aggregator's bitmap (④ in Figure 7). If not, ATP aggregates (adds) the packet's gradient data to the `value` field in the aggregators and also or's the bitmap field in the gradient packet to the bitmap field in the aggregator (⑤ in Figure 7). ATP increments the aggregator's `counter` field. If the packet already been aggregated (e.g., it was resent), ATP `OR`'s the `ECN` field from the packet into the aggregator's `ECN` field and drops the packet (⑦ in Figure 7).

If the `counter` in the aggregator is less than the corresponding fan-in degree (⑥ in Figure 7), ATP drops the gradient fragment packet and `OR`'s the `ECN` — this is the step that saves bandwidth to the PS. If, however, they are equal then aggregation at this switch is complete and ready to be pushed downstream. The switch replaces the packet's data field with the aggregator's value field, and the corresponding bitmap field with the aggregator's bitmap and sends the packet downstream towards the PS.

ATP chooses to forward the complete aggregation results to the PS instead of sending them back to workers. This design makes the aggregation results stored at the PS. When a parameter packet to workers is dropped, the PS can resend the aggregation result.

Aggregator Deallocation using Parameter Packets. The switch multicasts parameter packets back to the workers when it receives parameter packets from PS. A parameter packet works as the acknowledgement (ACK) of the gradient fragment packets, and must traverse the edge switches used for aggregation. When an ATP switch processes a parameter packet, the switch checks if the aggregator at the packet's index has a matching identifier, and if so deallocates the aggregator by changing all fields to null (⑧ in Figure 7).

3.6 End Host Logic

Workers in ATP push gradient fragment packets toward the PS and receive updated parameters back. The PS accepts gradient fragment packets, as well as partially or fully aggregated packets, to compute the full aggregation and sends the updated parameters back to workers. The PS also addresses collisions over aggregator indexes by *rehashing*.

Worker Pushing Gradients. The ATP end-host network stack obtains gradients by intercepting push or pull calls from DT jobs. It chunks these gradients into a sequence of 306B packets (58B header + 248B gradient values). ATP converts floating-point numbers in gradients to 32-bit integers [56] to work with switches that do not support floating-point operations. These gradient fragment packets are small and ATP introduces optimizations for high packet I/O throughput (§4).

PS Updating Parameters. ATP allocates an area of memory for each job at the PS for collecting aggregated gradients as an array of <`bitmap, value`> indexed by sequence number. The bitmap tracks which workers' gradient fragments have been aggregated in the `value` field. PS maintains a bitmap for each value to track which worker values have been aggregated. When a gradient fragment packet arrives, the PS compares its bitmap with that of the packet for an overlap. If they do not overlap, the PS aggregates the packet's data into its stored value, and updates the stored bitmap from the packet's bitmap. For example, if the incoming gradient packet is an individual gradient packet, PS checks if a packet from that worker was already aggregated (i.e., PS bitmap for the worker is set to 1) due to a resent packet, drops duplicates and otherwise updates the value and bitmap. On completion of aggregation of a parameter fragment, the PS sends the updated parameter fragment to the switch, which multicasts back to all workers in the job.

For a single gradient fragment, it is possible that the aggregator is busy when the first few packets arrive (hash collision), but available for later packets (released). In this case, the first packets are forwarded directly to the PS, while the remaining ones are aggregated at the switch. However, without intervention the switch will never send along the aggregated values because it is waiting for *packets that have already been sent*. Workers detect this stalled aggregation when they receive parameter packets for higher-sequenced fragments, and all workers will treat the stalled fragment as *a loss*. Each worker retransmits the stalled fragment with the `resend` bit set. This ensures completion of aggregation (by piggybacking on packet loss recovery; §3.7).

To reduce the frequency of aggregator collisions, we propose a *dynamic hashing scheme*. PS checks the `collision` bit of gradient packets. If the `collision` bit is set, the PS rehashes to get a new aggregator index (as HASH(<`aggregatorIndex`>)%`numAggregators`). It sends this new `aggregatorIndex` to workers in the unused `bitmap` field of the parameter packet. Workers remap the collision-prone index to the new index, and send any subsequent gradient that would have been sent with the old index with the new index instead. This simple approach helps *evolve* the hash function in a dynamic manner over time at each worker, making it collision-resistant.

Worker Receiving Parameters. The network stack maintains a sliding window over the sequence of gradient fragment packets. After sending an initial window of packets, the worker records the first un-ACKed sequence number as the *expected sequence number* and waits for parameter packets from the PS. The worker uses the parameter packets from PS to slide the window and send new gradient packets. When a worker receives a parameter packet, it checks if the packet has the expected packet sequence number. If the parameter packet was already received (e.g., because it is lost by some other worker and retransmitted at that worker's request), it is ignored. If it has the expected number, the worker increases the expected sequence number and invokes the congestion control algorithm (§3.7) to update the current window. If the number of in-flight packets is less than the congestion window, ATP sends the remaining window (congestion window size - in-flight packets) of gradients fragment packets. If the parameter has a sequence number higher than expected, ATP may consider the expected gradient fragment as lost, triggering loss recovery (§3.7).

3.7 Reliability and Congestion Control

Reliability. Due to loss of gradient fragment or parameter packets, the PS may not send parameter packets in sequence. As noted previously, when this occurs, a worker updates the received parameters but does *not* update expected sequence number. When a worker receives three consecutive parameter packets other than the expected sequence number, it detects loss of the gradient fragment with the expected sequence number. In this case, ATP worker retransmits the missing fragment packet with the `resend` bit set; this indicates to switches that there may be a partial aggregation state in the switch.

ATP takes a simple approach and does not try to do in-network aggregation of resent gradients. This design considers the case that packet drop is due to one or more worker failures, where worker recovery can take a significant amount of time (§A.3). In this case, the remaining active workers still keep retransmitting gradient packets, but the switch cannot complete the aggregation. The aggregators are occupied but do not perform effective aggregation, which wastes the aggregator.

ATP takes different steps at the first and second levels of aggregation. At the first level, when a resent packet arrives, the switch checks for a matching aggregator. If it exists, and the aggregator bitmap does not indicate that the resent packet's fragment has already been aggregated, then the switch aggregates the value from the packet into the aggregator, merges the bitmap from the packet into the aggregator's bitmap, forwards the results (which may be partial) downstream, and *deallocates* the aggregator. When subsequent resent packets arrive, the corresponding aggregator has already been deallocated, so the switch simply forwards the resent packet downstream.

At the second level, the switch is unable to merge partial results because it has one bit rather than a bitmap to indicate the aggregation status for each of its first-level aggregation. Instead, the second-level switch discards its aggregation state and forwards any resent packet (including partial aggregations from the first level) to the PS, where the aggregation ultimately completes. This ensures that upon packet loss, all gradient fragments are (re)sent to the PS and the aggregator for the fragment is deallocated.

The memory leaks can occur when a job stops abnormally before PS sends parameter packets to deallocate aggregators. Without any mechanism, the aggregator would remain occupied because the PS never ACKs with a corresponding parameter packet. To handle this, on every parameter packet, the switch checks the `timeout` value for the register specified by the parameter packet's index. Even if its job ID and sequence number do not match the parameter packet, the switch will deallocate the aggregator if the timestamp is older than a configured value.

Congestion Control. In a multi-tenant network, multiple ATP jobs and other applications share the network. They contend for various resources including network bandwidth, receiver CPU, and switch buffers. In ATP, multiple jobs also contend for aggregators at the switches.

High contention for aggregators in the switch can lead to a situation where aggregators cannot aggregate all traffic. This causes the traffic volume to increase, which will trigger queue length buildup in switches and packet loss due to switch buffer overflow. The observable symptoms in this case are similar to network congestion. Based on this, ATP uses *congestion control* to manage *all* contended resources.

In TCP, senders detect congestion using RTT, duplicated ACKs, or ECN marks, and respond by adjusting sending windows. For ATP, we pick ECN marks as the primary congestion signal. RTT measured from a worker sending a gradient packet to it receiving a parameter ACK packet will not work because it includes synchronization delay between workers. As all the workers receive the same parameter packet, using ECN ensures that all workers see similar congestion signals. We enable the ECN marking in switches, and use both ECN and (rare) packet loss as the congestion signal. To ensure that ECN marks are not lost during aggregation, ATP merges the `ECN` bit in fragment packets into the `ECN` bit in aggregator (⑦ in Figure 7), which is later forwarded to the PS when aggregation completes. This `ECN` bit is then copied to the parameter packet and eventually reaches all the workers.

Each ATP worker applies Additive Increase Multiplicative Decrease (AIMD) to adjust its window in response to congestion signals. The window size ATP starts at 200 packets, which at 300 bytes for each packet is within the bandwidth-delay product ($\sim$ 60KB) of a 100Gbps network. ATP increases window size by one MTU (1500 bytes or 5 packets) for each received parameter packet until it reaches a threshold, which is similar to slow start in TCP. Above the slow-start threshold, ATP increases window size by one MTU per window. When a worker detects congestion via ECN marking on a parameter ACK or three out-of-order ACKs, it halves the window, and updates the slow start threshold to the updated window.

3.8 Dealing with Floating Point

Gradient values are real numbers and DNN training frameworks offer several numerical types to represent them, each type offering varying trade-offs between range, precision and computational overhead. Gradient values are typically represented using 32-bit floating point type.

The current generation of programmable switches does not support 32-bit floating point arithmetic. Like prior work [56], ATP converts gradient values at each worker from 32-bit floating point representation to 32-bit integer representation by multiplying the floating point number by a scaling factor (10^8) and rounding to the nearest integer. The switch aggregates these 32-bit integers and PS converts the aggregated value back to 32-bit floating points by dividing by the scaling factor. ATP chooses a large scaling factor, i.e., 10^8, so as to minimize loss of precision as 32-bit floating point representation provides precision of 7 decimal digits [7]. A detailed justification of ATP's choice of scaling factor can be found in §B.1.

A large scaling factor can lead to the overflow of aggregated gradients. There are two alternatives to deal with overflows: proactive and reactive. The former (described in §B.2) is cautious and wastes opportunities for in-network aggregation by sending some gradient packets directly to the PS for aggregation. We explain the pitfalls of this mechanism in §B.2. Instead, ATP uses a reactive mechanism: all gradient packets are sent with the intention of aggregation at a network switch. If a gradient packet triggers overflow at an aggregator in the switch, we utilize a switch feature (*saturation*) to set the aggregator value to the maximum value or minimum value represented with 32-bit integers. If the aggregator value is saturated, any further gradient packets destined at this aggregator only update the directions (i.e., bitmap, fanIndegrees) and the value remains saturated. When the aggregation is done, i.e., fanInDegree value is equal to the number of workers, the saturated aggregator value is written to the gradient packet and sent to the PS. If the PS finds the aggregator value is saturated, it requests the original gradient values in floating-point format from all workers. This triggers a retransmission and all the workers send packets with floating-point gradient values directly to the PS, which finally performs aggregation.

In the worst case, such a reactive overflow correction incurs the cost of retransmission of all gradient packets in an iteration. Note that the overhead incurred during overflow correction is exactly the same as that during packet loss recovery. In our evaluations, we see no deterioration in training throughput if the packet loss rate is $< 0.1\%$ (§5.2.3). This translates to overflow correction as well: if the frequency of overflow correction is $< 0.1\%$ (over packets), we will see no deterioration in training throughput. We empirically show that there is hardly any overflow for all popular models' training in §5.3. The frequency of overflow correction can be further reduced using a dynamic scaling factor (§B.3).

Because overflow correction adds little overhead, ATP's in-network aggregation yields substantial speed-up in per-epoch times. Coupled with quantization not affecting the number of epochs, ATP overall yields significant gains in time-to-target-accuracy, as we also show in §5.3.

4 Implementation

ATP's implementation consists of (i) the protocol logic on P4 switches and end host, and (ii) hardware offloads to optimize small-packet performance.

Programmable Switch. The switch implementation has processing logic for gradient aggregation and control logic to allocate, deallocate, and manage aggregators. The main challenge is that the whole packet must be parsed in a limited time budget and processed in the limited switch pipeline stages.

Aggregation. Prior work [56] processed packets in a single pass, which limited packets to 184B. ATP increases this limit by taking two passes (called **two-pass**) at the switch for each packet, which is a mixed usage of the *resubmit* [3] and *recirculate* features. The details are in §C. This raises the maximum packet size to just 306B – larger, but still small packets. This leaves 4 switch stages for control operations.

Control logic. This is responsible for checking whether an aggregator is available, processing protocol flags, and updating the aggregation state. To work within the restriction of one-time access to a register, ATP applies various techniques to handle complex operations. Consider the bitmap check process, which involves a read of the bitmap in the aggregator and then an arithmetic operation on the gradient value and bitmap value; finally, a write to the bitmap in the aggregator. We instead note that a write to bitmap is equivalent to setting a bit always. This allows us to reorder the write operation to just before the read operation. This read-followed-by-write serves as one-time register access which is permissible. Another method in ATP to address one-time access restriction for one packet is to use two packets; e.g., ATP allocates the aggregator with gradient packets but deallocates the associated aggregators using parameter packets.

[3]We leverage ‘ force_shift ingress‘ feature to drop the data part that has been aggregated before the resubmit.

End-Host Networking Stack. We implement ATP as a BytePS [69] plugin, which integrates in PyTorch [15], TensorFlow [14] and MXNet [19]. BytePS allows ATP's use without application modifications. ATP intercepts the *Push* and *Pull* function calls at workers as they communicate with ATP PS.

Small Packet Optimizations. ATP's network stack leverages Mellanox's RAW ETHERNET userspace network programming model [1]. ATP uses **TSO** [50] and Multi-Packet QP (**MP-QP**) [42] hardware acceleration features to improve small-packet processing. TSO speeds packet sending by offloading packetization to the NIC and improves PCIe bandwidth via large DMA transfers. To improve packet receiving rate, MP-QP uses buffer descriptors that specify multiple contiguous packet buffers and reduce the NIC memory footprint by at least $512X$. These features together reduce CPU cost via fewer calls to send/receive packets and fewer DMA operations to fetch packet send and receive descriptors.

ATP uses multiple threads to speed up packet processing. When ATP receives tensors to transfer, it assigns the tensor to a thread to send, which may cause load imbalance across different threads. Each worker in ATP has a centralized scheduler to receive tensors from the application layer, and maintains the total workload for each thread. Whenever the scheduler receives tensors to be sent, it extracts the size and assigns the tensor to the least loaded thread to balance the load.

Baseline Implementation. We implement a prototype of SwitchML [56], which uses the switch as the PS and provides a timeout-based packet-loss recovery mechanism. We apply TSO and MP-QP features to the SwitchML implementation at end hosts to improve small-packet operations, but do not apply the two-pass optimization at the switch to align with the public version of SwitchML. As a result, the packet size for SwitchML implementation is 180 bytes. We also open-source our SwitchML implementation [3].

5 Evaluation

We evaluate ATP via testbed experiments and software emulation to answer the following questions:

1. How does ATP perform compared to state-of-the-art approaches for a single job (§5.2.1)?
2. How does ATP's inter-rack aggregation perform compared to an alternate rack-scale service (§5.2.2)?
3. What are the overheads of ATP's loss recovery mechanisms (§5.2.3)?
4. How does conversion to integers in ATP affect time to accuracy (§5.3)?
5. How does dynamic aggregator allocation compare to a centralized static scheme under multi-tenancy (§5.4)?
6. How effective is ATP's congestion control (§5.5)?

5.1 Experimental Setup

Cluster Setup. We evaluate ATP on a testbed with 9 machines. 8 of them have one NVIDIA GeForce RTX 2080Ti GPU with NVIDIA driver version 430.34 and CUDA 10.0.

All machines have 56 cores of Intel(R) Xeon(R) Gold 5120T CPU @ 2.20GHz, 192GB RAM with Ubuntu 18.04 and Linux kernel 4.15.0-20. Each host has a Mellanox ConnectX-5 dual-port 100G NIC with Mellanox driver OFED 4.7-1.0.0.1. All the hosts are connected via a 32x100Gbps programmable switch with a Barefoot Tofino chip. We evaluate inter-rack aggregation using Tofino's software switch model [5] to emulate ATP switches in software with the same code.

Baselines. We compare ATP against BytePS [69]; both use a worker-PS architecture. BytePS supports TCP (BytePS + TCP) and RDMA over Converged Ethernet (RoCE) [36] (BytePS + RDMA) as network protocols. We turn on PFC [25] for RDMA to provide a lossless network. While ATP uses the default of N workers to 1 PS (labeled Nto1), we optimize BytePS with as many PSs as the workers (labeled NtoN) to alleviate the network bottleneck. Also, we co-locate one each of the N PSs and N workers in the same machine. We also compare ATP with our implementation of SwitchML, a state-of-the-art baseline with in-network aggregation support. We also compare ATP against Horovod [58] with RoCE (Horovod+RDMA) and with TCP (Horovod+TCP) which uses a ring all-reduce architecture [61].

Workloads. We run Pytorch on top of the above schemes to evaluate many popular real-world models: AlexNet, VGG11, VGG16, VGG19, ResNet50, ResNet101, and ResNet152 [33, 44, 60]. Each model trains on the ImageNet dataset. The DT job has 8 workers unless specified. For most experiments we use VGG16 (model size 528MB) and ResNet50 (model size 98MB), as representatives for network-intensive and compute-intensive workloads, respectively. We also run an aggregation microbenchmark where each worker repeatedly transfers 4MB tensors (maximum size BytePS supports), which are aggregated in the network (ATP) or at the PS(s) (BytePS), and are then sent back to the workers. In contrast to real jobs, this microbenchmark has equal-sized tensors and always has data to send with no "off" phase.

Metrics. We use three metrics to measure performance: (1) *training throughput* for DT jobs, which is the number of images processed per second (*image/sec*) normalized by the number of workers; (2) *time to accuracy* to show a DT job's quality, which is the training time to reach a target or maximum accuracy; and (3) *aggregation throughput* for the microbenchmark, which is the total bytes of parameters received at each worker per second (*Gbps*).

5.2 Single Job Performance

5.2.1 ATP Training Performance

We compare ATP against all baselines on single-job training throughput for all the models in our workload as shown in Figure 8. ATP achieves the best performance for all jobs with a maximum speedup of 1.5X over BytePS NtoN RDMA, 1.24X over Horovod RDMA, 2.5X over BytePS NtoN TCP, 8.7X over BytePS Nto1 TCP, 4.2X over Horovod TCP and 1.5X over SwitchML. Performance gains are larger on network-intensive workloads (VGG) than compute-intensive workloads (ResNet). PS-based ATP is comparable to, and in many cases outperforms, the state-of-the-art ring all-reduce approach (Horovod+RDMA). ATP outperforms SwitchML due to support for larger packet size made possible via our optimized two-pass implementation of switch logic.

5.2.2 Inter-rack Aggregation

ATP provides aggregation at two levels in inter-rack configurations. We pick a typical network topology as shown in Figure 9, where the PS is connected to switch $SW2$ and workers $w0$-$w5$ are connected to different switches. We compare against a rack-scale solution (*RSS*) that aggregates locally at each rack and forwards partial aggregates to the PS. In our test topology, *RSS* performs aggregation for $w0$-$w1$ at $SW0$, $w2$-$w3$ at $SW1$, $w4$-$w5$ at $SW2$, and then, aggregation from $SW0$, $SW1$ and $SW2$ at the PS. This approach simplifies the algorithm at the switch at the cost of more network traffic to the PS. We measure the amount of traffic the PS receives with ATP and *RSS* using the software simulator (so we cannot measure real throughput). PS with *RSS* receives $3X$ more traffic than PS with ATP. This is because *RSS* sends the partial aggregates from $SW0 - SW2$ to PS while ATP aggregates individual packets from $w5$ and $w4$ and partial aggregates from $SW0$ and $SW1$ at switch $SW2$ before they are sent to PS; this eliminates $^2\!/_3$ of the traffic to the PS.

5.2.3 Packet Loss Recovery Overhead

ATP handles packet loss at the end host, but guarantees aggregation correctness and prevents memory leaks in the switch. To evaluate the overhead of packet loss recovery, we configure one worker to adversarially drop packets with a packet loss rate between 0.001% and 1% (as in prior work [56]). We compare against SwitchML, which uses a timeout mechanism to detect packet loss, and turn off ATP's congestion control to avoid window back-off due to this adversarial packet loss. We use the timeout value (1ms) from SwitchML.

Figure 10 shows the training and microbenchmark throughput normalized to no loss for varying loss rates. Overall, ATP degrades gracefully when the loss rate increases, and to a lesser degree than SwitchML. This is because ATP adopts out-of-sequence ACKs as a packet loss signal, which enables ATP to detect and respond to packet losses faster than SwitchML.

5.3 ATP Time-to-Accuracy (TTA)

5.3.1 Single Job TTA

ATP changes the nature of computation of gradient aggregation via conversion of gradient values to integers to enable aggregation on the switch and has reactive mechanisms to deal with overflows. To confirm this does not affect the training quality of our workload, we evaluate ATP's accuracy over time against the baseline (BytePS NtoN RDMA). We find that ATP spends the same number of epochs to achieve the same top-5 accuracy as BytePS NtoN RDMA for all the models. Figure 11 plots the top-5 accuracy with time for ResNet50,

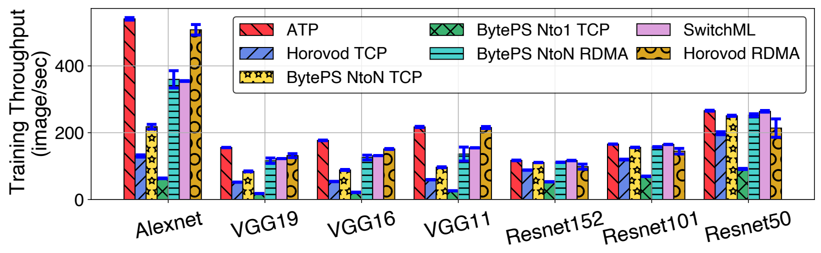


Figure 8: Single job throughput

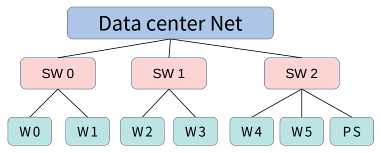
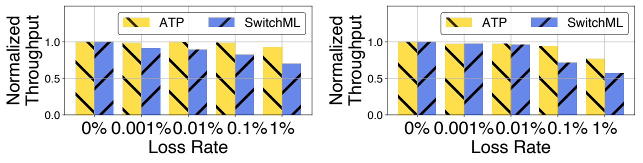


Figure 9: Multi-rack topology



(a) VGG16 (b) Microbenchmark

Figure 10: Throughput in different packet loss rate.

VGG16 and ResNet152. With VGG16, a network-intensive workload, ATP outperforms BytePS and reaches 75% top-5 accuracy 1.25X faster than BytePS. With ResNet50, ATP and BytePS reach 93% top-5 accuracy in comparable amount of time (ATP is 1.02X faster) as shown in Figure 11a. ATP does not speed up training for ResNet50 as it is a compute-intensive workload. ResNet152 (Figure 11c) exhibits the same trend as ResNet50. We also conduct TTA experiments on VGG19, ResNet101 and AlexNet models, and observe that ATP reaches target accuracy 1.2X, 1.01X and 2.39X faster, respectively. Figures for these results are in §D.3.

Overflow Correction. Retransmissions due to ATP's overflow correction mechanism can be a source of overheads. In our results, we see overflows only with ResNet152, with 445 aggregated gradient packets (i.e., 0.00002% of all aggregated gradient packets generated in an epoch) requiring overflow correction. These happen only in the first few iterations of the first epoch when the model is initialized with random weights and the magnitude of gradient updates is large.

Worst-case Precision Loss. ATP uses a scaling factor of 10^8. Gradient values with a magnitude less than 10^{-8} are approximated as zero and experience complete loss of precision. We observe that an average of only 0.00002% of the gradients values per epoch in ResNet50 are less than 10^{-8}. As a result, ATP causes no loss of accuracy in the final trained model.

5.3.2 Multi-job Time-to-Accuracy (TTA)

We also evaluate TTA with 2 VGG16 jobs for ATP, BytePS NtoN RDMA, and Horovod with RDMA. Each job has three workers. We use one switch to emulate a dumbbell topology and separate one worker from the other two of a job at the two ends of the dumbbell. The link line rate is 100$Gbps$. A worker-to-worker (or PS) path from each job share the dumbbell link. Figure 12 shows the 75% top-5 accuracy with time for each VGG16 job. Similar as single job TTA performance, ATP outperforms BytePS and Horovod and reaches 75% top-5 accuracy 1.20X faster than the fastest BytePS job, and 1.25X faster than Horovod. Two jobs in ATP and Horovod achieve the same accuracy with the same training time, while this is not the case for BytePS. We observed PFC storms from NICs in BytePS, which we suspect is due to the heavy PCIe contention between NIC and GPU.

In summary, these results demonstrate that ATP does not compromise training quality and is able to achieve baseline training accuracy in less time than other methods owing to the acceleration provided by in-network aggregation.

5.4 Multiple Jobs

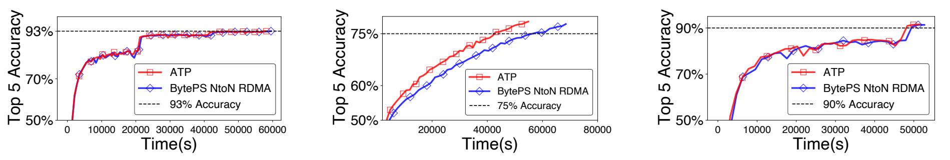
ATP does dynamic best-effort sharing of switch resources across multiple jobs. Our multi-tenant extension of SwitchML statically partitions switch resources equally across jobs.

5.4.1 Dynamic vs. Static Sharing

We compare ATP's dynamic best-effort approach against SwitchML's static approach by launching 3 identical VGG16 jobs (with identical placement for workers and PS) connected to one programmable switch. We vary the number of aggregators for the 3 jobs on the switch. The static approach allocates a fixed 1/3 fraction of the aggregators to each job, while ATP's dynamic approach shares these aggregators dynamically, so that when any job is in an off phase its aggregators are available to other jobs.

We first tune the number of aggregators reserved for the 3 jobs to find the minimum number needed to get maximal training throughput for each job with the static approach. We find that 1980 aggregators, referred to as *peak throughput aggregators* (PTA), equally divided across the three jobs maximize throughput because jobs saturate the link from the switch to the PS. To measure the impact of sharing strategy under contention, we reduce the number of available aggregators from 100% of PTA to 33% of PTA, and measure training throughput with both static and dynamic approaches.

Figure 13a shows the average training throughput (measured after warm-up, from the second epoch of training onwards) as we vary the number of aggregators available to the 3 jobs from 100% of PTA (1980 aggregators) to 33% of PTA (660 aggregators). With 100%, the dynamic approach performs similarly to static approach. This is because in both cases all aggregation happens in-network, and the switch to PS link is fully utilized. As we reduce the number of aggregators available, throughput for the dynamic approach degrades



(a) ResNet50 (b) VGG16 (c) ResNet152

Figure 11: Time to accuracy

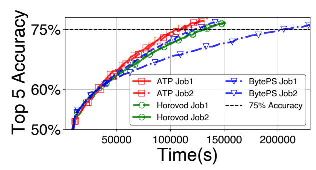
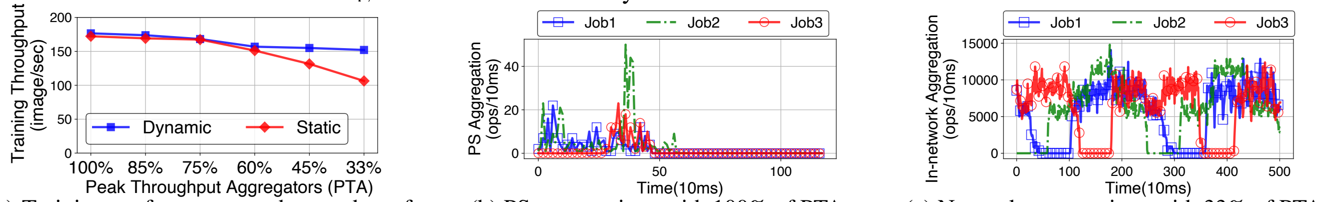


Figure 12: TTA with two VGG16 jobs



(a) Training performance as the number of PTA changes. (b) PS aggregations with 100% of PTA (c) Network aggregations with 33% of PTA

Figure 13: Dynamic v.s. static allocation with different peak throughput aggregators (PTA) ratio.

less than that with the static approach. This is because the dynamic approach allows sharing of unused aggregators across jobs and also uses available switch-to-PS link capacity for PS-based aggregation. We show this by delving into ATP's dynamic approach when using 100% and 33% of PTA.

Figure 13b shows the total number of aggregations at the PS every 10ms for each job with ATP's dynamic approach using 100% of PTA, starting at job warm-up. Initially, there are hash collisions because of jobs allocating the same aggregator index, which leads to aggregation at the PS. After 500ms, ATP's hash-collision avoidance kicks in and the dynamic approach converges to complete in-network aggregation.

Figure 13c shows how ATP's dynamic approach responds when jobs enter an off phase. The figure shows the number of aggregations performed at the switch every 10ms when only 33% of PTA are available. The trace shows that when one job enters an off phase (zero in-network aggregations), the number of in-network aggregations for the remaining jobs increases. For example, at sample 120, job 3 (red) enters an off phase, and the in-network aggregations for jobs 1 and 2 increase until job 3 resumes at sample 180.

5.4.2 Effectiveness of ATP's Hashing Scheme

In §5.4.1, we show ATP's hashing scheme works with 100% of PTA for 3 jobs. To evaluate at larger scale, we launch 8 microbenchmark jobs with 100% of PTA with ATP's hashing enabled (**Hash-based dynamic**) and without hashing enabled (**Linear-based dynamic** which allocates aggregator indexes in sequence, i.e., index = seq_num % N), and compare against the baseline scheme (static allocation with 100% of PTA).

Figure 14 plots the aggregation throughput for the three schemes as the number of jobs increases. ATP's hash-based scheme matches the baseline static scheme and greatly outperforms the linear-based scheme, as without hashing, a continuous sequence of gradients from multiple jobs collide, introducing a significant amount of retransmission. Overall, these experiments indicate that the dynamic hash function can effectively distribute aggregators to each job. It can achieve

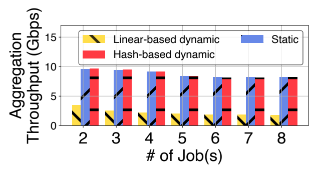


Figure 14: Large scales

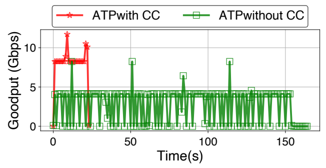
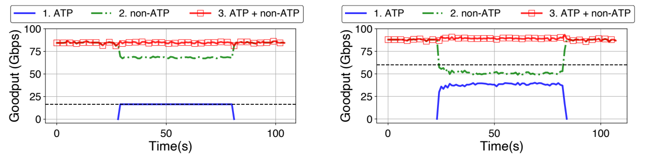


Figure 15: ATP job with and without congestion control



(a) ATP: VGG16 (b) ATP microbenchmark

Figure 16: Aggregation goodput with non-ATP traffic.

comparable performance to the static approach when there is no contention for switch aggregators, and outperform the static approach under contention.

5.5 Effectiveness of Congestion Control

ATP's congestion control mechanism aims to minimize packet loss while maximizing the link utilization. Network congestion happens (1) when ATP traffic is co-located with bandwidth-hungry normal traffic, such as TCP or RDMA transfers; (2) when ATP does not perform in-network aggregation due to a shortage of switch resources from other contending ATP jobs, causing an incast to the PS. We evaluate ATP's congestion control mechanism in both cases.

With non-ATP traffic. It is common to co-locate multiple systems and applications in a multi-tenant multi-rack cluster, such as storage and data pre-processing systems. We validate ATP's congestion control effectiveness when co-locating with such traffic. We launch a training job with 6 workers and 1 PS for ATP. We add background flows competing for bandwidth on a link to a worker, which individually can achieve line rate. The experiment starts with background traffic, and then starts a training job at $t = 25$s. We stop the training job after 50s.

We perform this experiment both with VGG16 and with our microbenchmark to emulate large models on ATP.

Figure 16a reports the aggregation goodput (total size of parameters aggregated in a second) from the worker that experiences network congestion for VGG16 (ATP), the goodput from the non-ATP traffic (non-ATP), and the aggregate goodput (ATP + non-ATP) over time. The dashed black line shows the peak goodput VGG16 job can achieve without background traffic; this link bandwidth demand is less than fair share. We see that the VGG16 job with ATP is able to achieve peak goodput (as demand is less than fair share) and that the sum of goodputs is close to line rate on the uplink from this worker. This indicates that ATP's CC is able to co-exist with non-ATP background traffic with max-min fair allocation in this setting.

Figure 16b plots the same scenario instead with the microbenchmark job. The dashed black line shows the peak goodput this job can achieve without background traffic, which is indicative of a link bandwidth demand greater than fair share of the network uplink rate at the worker. We see that ATP is able to achieve near the fair share of the bottleneck link (the uplink from this worker) and that the sum of goodput from both traffic types is close to link rate. This showcases near equitable sharing of link bandwidth. Figure 16a and Figure 16b show that the congestion control of ATP is able to respond quickly to changes in congestion, and converge to a new bandwidth which is very stable.

With other ATP traffic. We launch a single VGG16 job (8 workers) using ATP for 10 iterations. We reserve only 50% of the switch aggregators needed to achieve peak training throughput with complete in-network aggregation (to emulate contention from background ATP jobs). Figure 15 shows the aggregation goodput averaged per second from a single worker against time with and without congestion control (CC). We see that the congestion control of ATP kicks in, and the aggregation goodput stabilizes around 7.5Gbps for the worker. Note that this goodput is lower than the peak goodput ($\sim$20Gbps) from the previous experiment because here we have an 8-to-1 (8 workers to 1 PS) incast, only 50% of traffic is reduced in the network, and the PS in our implementation saturates at 60Gbps (§D.1).

Without CC, the goodput fluctuates dramatically between 4Gbps and 0; this is because incast causes frequent packet loss without congestion control, and introduces high packet loss recovery overhead. The training goodput of ATP with congestion control is 66.8 img/s while that without congestion control is 23.2 img/s, a decrease of 65%. These results show that the congestion control for ATP can effectively maintain high goodput and is effective in avoiding packet loss.

In summary, ATP's congestion control helps it co-exist with other tenants (both ATP and non-ATP).

6 Other Related Work

We discuss prior works that propose advances in hardware, software, offloads, and algorithms to accelerate DNN training.

Speedup Network Transmission. Prior efforts propose to improve gradient aggregation time by (1) smarter network scheduling – increasing the overlap between GPU/CPU computation and network transmission via fine-grained tensors transmission scheduling (per layer instead of the whole gradient or parameter) [32, 37, 52, 70]; combining model- and data-parallelism via pipelining [31, 35]; using asynchronous IO [20, 24]. (2) reducing network traffic – using large batch size to reduce the communication frequency [11, 29, 38]; using quantization [57] or reducing redundancy in SGD [45] to reduce bytes sent over the network; optimizing mixture of local-global aggregation to adapt to network change at runtime [21, 64]. ATP can incorporate these optimizations to further improve its performance.

In-network Aggregation. The idea of in-network aggregation has been explored in wireless networks [18, 59]; in big-data systems and distributed training systems using end-hosts [23], a specialized host [48], high performance middleboxes [49] or overlay networks [17,63]. DAIET [55] proposed a simple proof-of-concept design of in-network aggregation without a testbed prototype. ShArP [30], supported by special Mellanox Infiniband switches, builds an overlay reduction tree to aggregate data going through it, but it does not apply the aggregation until the switch receives all the data. ATP is the first to provide a *dynamic, best-effort* in-network aggregation service for multi-tenant multi-switch clusters.

7 Conclusion

We build an in-network aggregation service, ATP, to accelerate DT jobs in multi-tenant multi-rack networks. ATP provides a *best effort in-network aggregation* primitive via careful co-design of switch logic (for aggregation) and the end-host networking stack (for reliability and congestion control). Testbed experiments show that ATP is able to outperform existing systems by up to $8.7X$ for a single job, and it is even slightly better than the current state-of-the-art ring all-reduce with RDMA. In a multi-tenant scenario, best-effort in-network aggregation with ATP enables efficient switch resource usage, and outperforms current state-of-the-art static allocation techniques by up to 38% in terms of training time when there is heavy contention for on-switch resources.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd Kai Chen for their thoughtful feedback. Yanfang Le, Kshiteej Mahajan, Aditya Akella and Michael Swift are supported in part by NSF grants CNS-1565277, CNS1719336, CNS-1763810, CNS-1838733, gifts from Google and VMware and by NeTS-1717039. ChonLam Lao, Yixi Chen and Wenfei Wu are supported by National Natural Science Foundation of China Grant No.61802225 and the Zhongguancun Haihua Institute for Frontier Information Technology.

References

[1] RAW Ethernet Programming. https://docs.mellanox.com/display/MLNXOFEDv461000/Programming.

[2] ATP Source Code. https://github.com/in-ATP/ATP.

[3] ATP-SwitchML Source Code. https://github.com/in-ATP/switchML.

[4] Barefoot Tofino. https://www.barefootnetworks.com/technology/#tofino.

[5] Barefoot Tofino Software Behavior Model. https://www.barefootnetworks.com/products/brief-p4-studio/.

[6] Google TPU. https://cloud.google.com/tpu/.

[7] IEEE 754-1985. https://en.wikipedia.org/wiki/IEEE_754-1985.

[8] Intel FlexPipe. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/ethernet-switch-fm6000-series-brief.pdf.

[9] LiquidIO Server Adapters. http://www.cavium.com/LiquidIO_Server_Adapters.html.

[10] Netronome NFP-6000 Intelligent Ethernet Controller Family. https://www.netronome.com/media/redactor_files/PB_NFP-6000.pdf.

[11] Now anyone can train imagenet in 18 minutes. https://www.fast.ai/2018/08/10/fastai-diu-imagenet/.

[12] Nvidia clocks world's fastest bert training time and largest transformer based model, paving path for advanced conversational ai. https://devblogs.nvidia.com/training-bert-with-gpus/.

[13] XPliant Ethernet Switch Product Family. http://www.cavium.com/XPliant-Ethernet-Switch-Product-Family.html.

[14] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, November 2016. USENIX Association.

[15] Paszke Adam, Gross Sam, Chintala Soumith, Chanan Gregory, Yang Edward, DeVito Zachary, Lin Zeming, Desmaison Alban, Antiga Luca, and Lerer Adam. Automatic differentiation in pytorch. In *In NIPS 2017 Autodiff Workshop: The Future of Gradient-based Machine Learning Software and Techniques*.

[16] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. Conga: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 503–514, New York, NY, USA, 2014. Association for Computing Machinery.

[17] D. C. Arnold and B. P. Miller. Scalable failure recovery for high-performance data aggregation. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–11, 2010.

[18] Raghav Bhaskar, Ragesh Jaiswal, and Sidharth Telang. Congestion lower bounds for secure in-network aggregation. In *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WISEC '12, page 197–204, New York, NY, USA, 2012. Association for Computing Machinery.

[19] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[20] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 571–582, Broomfield, CO, October 2014. USENIX Association.

[21] M. Cho, U. Finkler, M. Serrano, D. Kung, and H. Hunter. Blueconnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy. *IBM Journal of Research and Development*, 63(6):1:1–1:11, 2019.

[22] Eyal Cidon, Sean Choi, Sachin Katti, and Nick McKeown. Appswitch: Application-layer load balancing within a software switch. In *Proceedings of the First Asia-Pacific Workshop on Networking*, APNet'17, page 64–70, New York, NY, USA, 2017. Association for Computing Machinery.

[23] Paolo Costa, Austin Donnelly, Antony Rowstron, and Greg O'Shea. Camdoop: Exploiting in-network aggregation for big data applications. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 29–42, San Jose, CA, April 2012. USENIX Association.

[24] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, page 1223–1231, Red Hook, NY, USA, 2012. Curran Associates Inc.

[25] Claudio DeSanti. IEEE 802.1: 802.1Qbb - Priority-based Flow Control. http://www.ieee802.org/1/pages/802.1bb.html, 2009.

[26] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[27] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 435–450, New York, NY, USA, 2020. Association for Computing Machinery.

[28] Nadeen Gebara, Tenzin Ukyab, Paolo Costa, and Manya Ghobadi. Panama: Network architecture for machine learning workloads in the cloud. https://people.csail.mit.edu/ghobadi/papers/panama.pdf.

[29] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[30] R. L. Graham, D. Bureddy, P. Lui, H. Rosenstock, G. Shainer, G. Bloch, D. Goldenerg, M. Dubman, S. Kotchubievsky, V. Koushnir, L. Levi, A. Margolin, T. Ronen, A. Shpiner, O. Wertheim, and E. Zahavi. Scalable hierarchical aggregation protocol (sharp): A hardware architecture for efficient data reduction. In *2016 First International Workshop on Communication Optimizations in HPC (COMHPC)*, pages 1–10, 2016.

[31] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv preprint arXiv:1806.03377*, 2018.

[32] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy H Campbell. Tictac: Accelerating distributed deep learning with communication scheduling. *arXiv preprint arXiv:1803.03288*, 2018.

[33] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[34] Keqiang He, Eric Rozner, Kanak Agarwal, Wes Felter, John Carter, and Aditya Akella. Presto: Edge-based load balancing for fast datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 465–478, New York, NY, USA, 2015. Association for Computing Machinery.

[35] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems*, pages 103–112, 2019.

[36] InfiniBand Trade Association. Supplement to InfiniBand Architecture Specification Volume 1 Release 1.2.1 Annex A17: RoCEv2. https://cw.infinibandta.org/document/dl/7781, 2014.

[37] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. Priority-based parameter propagation for distributed dnn training. *arXiv preprint arXiv:1905.03960*, 2019.

[38] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, et al. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes. *arXiv preprint arXiv:1807.11205*, 2018.

[39] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed DNN training in heterogeneous gpu/cpu clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 2020.

[40] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In

15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18), pages 35–49, Renton, WA, April 2018. USENIX Association.

[41] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 121–136, New York, NY, USA, 2017. Association for Computing Machinery.

[42] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter rpcs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 1–16, Boston, MA, February 2019. USENIX Association.

[43] Antoine Kaufmann, SImon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. High performance packet processing with flexnic. In *Proceedings of the 21th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 67–81, New York, NY, USA, 2016. Association for Computing Machinery.

[44] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017.

[45] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887*, 2017.

[46] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. Incbricks: Toward in-network computation with an in-network cache. In *Proceedings of the 22th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 795–809, New York, NY, USA, 2017. Association for Computing Machinery.

[47] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. Distcache: Provable load balancing for large-scale storage systems with distributed caching. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 143–157, Boston, MA, February 2019. USENIX Association.

[48] Liang Luo, Jacob Nelson, Luis Ceze, Amar Phanishayee, and Arvind Krishnamurthy. Parameter hub: A rack-scale parameter server for distributed deep neural network training. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 41–54, New York, NY, USA, 2018. Association for Computing Machinery.

[49] Luo Mai, Lukas Rupprecht, Abdul Alim, Paolo Costa, Matteo Migliavacca, Peter Pietzuch, and Alexander L. Wolf. Netagg: Using middleboxes for application-specific on-path aggregation in data centres. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, page 249–262, New York, NY, USA, 2014. Association for Computing Machinery.

[50] Mellanox Technologies. Mellanox OFED for Linux User Manual. Rev 3.40. http://www.mellanox.com/related-docs/prod_software/Mellanox_OFED_Linux_User_Manual_v3.40.pdf.

[51] Heng Pan, Zhenyu Li, JianBo Dong, Zheng Cao, Tao Lan, Di Zhang, Gareth Tyson, and Gaogang Xie. Dissecting the communication latency in distributed deep sparse learning. In *Proceedings of the ACM Internet Measurement Conference*, IMC '20. Association for Computing Machinery, 2020.

[52] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 16–29, New York, NY, USA, 2019. Association for Computing Machinery.

[53] Dan R. K. Ports and Jacob Nelson. When should the network be the computer? In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, page 209–215, New York, NY, USA, 2019. Association for Computing Machinery.

[54] Davide Sanvito, Giuseppe Siracusano, and Roberto Bifulco. Can the network be the ai accelerator? In *Proceedings of the 2018 Morning Workshop on In-Network Computing*, NetCompute '18, page 20–25, New York, NY, USA, 2018. Association for Computing Machinery.

[55] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. In-network computation is a dumb idea whose time has come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets-XVI, page 150–156, New York, NY, USA, 2017. Association for Computing Machinery.

[56] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan RK Ports, and Peter Richtárik. Scaling distributed machine learning with in-network aggregation. *arXiv preprint arXiv:1903.06701*, 2019.

[57] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 1-bit stochastic gradient descent and application to

data-parallel distributed training of speech DNNs. In *Interspeech 2014*, September 2014.

[58] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[59] Mohamed A. Sharaf, Jonathan Beaver, Alexandros Labrinidis, and Panos K. Chrysanthis. Tina: A scheme for temporal coherency-aware in-network aggregation. In *Proceedings of the 3rd ACM International Workshop on Data Engineering for Wireless and Mobile Access*, MobiDe '03, page 69–76, New York, NY, USA, 2003. Association for Computing Machinery.

[60] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[61] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in mpich. *International Journal of High Performance Computing Applications*, 19(1):49–66, February 2005.

[62] Indu Thangakrishnan, Derya Cavdar, Can Karakus, Piyush Ghai, Yauheni Selivonchyk, and Cory Pruce. Herring: Rethinking the parameter server at scale for the cloud. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020.

[63] Raajay Viswanathan and Aditya Akella. Network-accelerated distributed machine learning using mlfabric. *arXiv preprint arXiv:1907.00434*, 2019.

[64] Jianyu Wang and Gauri Joshi. Adaptive communication strategies to achieve the best error-runtime trade-off in local-update sgd. *arXiv preprint arXiv:1810.08313*, 2018.

[65] Wei Wang, Meihui Zhang, Gang Chen, H. V. Jagadish, Beng Chin Ooi, and Kian-Lee Tan. Database meets deep learning: Challenges and opportunities. *SIGMOD Rec.*, 45(2):17–22, September 2016.

[66] Zheng Wang and Michael O'Boyle. Machine learning in compiler optimization. *Proceedings of the IEEE*, 106(11):1879–1901, 2018.

[67] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[68] Hyunho Yeo, Youngmok Jung, Jaehong Kim, Jinwoo Shin, and Dongsu Han. Neural adaptive content-aware internet video delivery. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 645–661, Carlsbad, CA, October 2018. USENIX Association.

[69] Jiang Yimin. BytePS. https://github.com/bytedance/byteps.

[70] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P. Xing. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 181–193, Santa Clara, CA, July 2017. USENIX Association.

[71] Hang Zhu, Zhihao Bai, Jialin Li, Ellis Michael, Dan R. K. Ports, Ion Stoica, and Xin Jin. Harmonia: Near-linear scalability for replicated storage with in-network conflict detection. *Proc. VLDB Endow.*, 13(3):376–389, November 2019.

A Additional Design Details

This section is a supplement to ATP's design description from §3. We first provide additional design details for clearer exposition and then outline design extensions to make ATP more general and robust.

A.1 Switch Algorithm for Reliability

Figure 17 explains the details on how the switch handles the three issues discussed in Section 3.7. If the retransmitted gradient fragment packets have an aggregator at switch (① in Figure 17), ATP uses the bitmap in the gradient fragment packet header to check if bitmap field in the aggregator is set (⑤). If unset, the data field element is aggregated to the value field of the aggregator and the bitmap field in the aggregator is set (⑥). After aggregation or if the retransmitted gradient has already been seen, ATP copies the value field and the bitmap field from the aggregator to the data field and the bitmap field of the packet, deallocates the aggregator, and sends the packet downstream (⑦). Given that only the first retransmitted gradient packet triggers aggregator deallocation, the retransmitted packets with the same sequence number from other workers do not hit the aggregators or reserve a new aggregator and thus, they will be directly forwarded to upstream devices. Note that at the second level of aggregation, ATP directly forwards a retransmitted gradient packet for simplicity (③). In all the cases, ATP deallocates the aggregator and sets all fields to null (③ and ⑦).

The parameter packet, whether it is retransmitted or not, always triggers the deallocation of the aggregator with the same <Job ID, Sequence Number>.

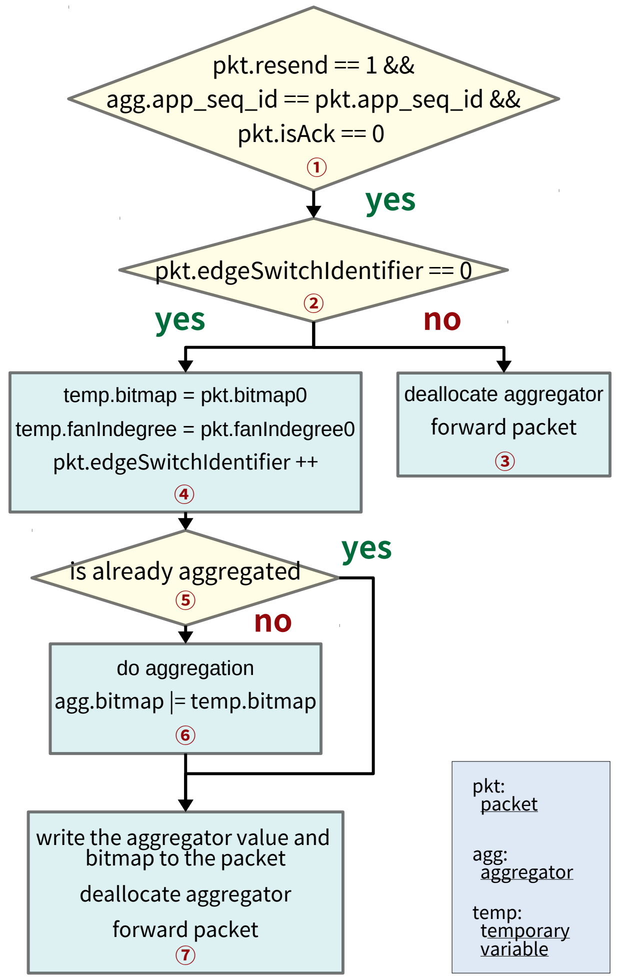

pkt.resend == 1 &&
agg.app_seq_id == pkt.app_seq_id &&
pkt.isAck == 0
①
yes
pkt.edgeSwitchIdentifier == 0
②
yes
no
temp.bitmap = pkt.bitmap0
temp.fanIndegree = pkt.fanIndegree0
pkt.edgeSwitchIdentifier ++
④
deallocate aggregator
forward packet
③
is already aggregated
⑤
yes
no
do aggregation
agg.bitmap |= temp.bitmap
⑥
write the aggregator value and bitmap to the packet
deallocate aggregator
forward packet
⑦
pkt: packet
agg: aggregator
temp: temporary variable

Figure 17: Pseudocode of the switch logic dealing with packet loss.

A.2 Aggregation at Multiple Hierarchy Levels

ATP can be extended to support aggregation at multiple hierarchy levels, i.e., beyond ToR switches to also enable aggregation at aggregation layer switches and core layer switches. The only pre-requisite is that packet routes for all gradient packets have to be deterministic (e.g., as in ECMP) and known ahead of time so that ATP knows the exact switch in the network at which gradient packets (or partially aggregated gradient packets) for a particular sequence number converge. This helps to precisely determine the fan-in degree at each intermediate switch from workers to the PS. This helps higher-level switches to aggregate partially aggregated results from lower-level switches and determine when aggregation at higher-level switches is complete. To enable multi-level aggregation, we need to add more `bitmap` and `fanInDegree` fields to ATP packet header (one for each additional in-network gradient aggregating switch) to track the progress of aggregation at all the switches involved in gradient aggregation enroute from the workers to the PS.

Notably, for non-deterministic route load balancing schemes, such as Presto [34] and CONGA [16], ATP packets only deterministically route through TOR switches with non-deterministic routing at higher levels of the datacenter network hierarchy. Thus, to support such cases, ATP does aggregation only at the TOR-layer.

A.3 Recovering from Worker Failures

PS-based architectures deal with worker failures by re-spawning a worker process, perhaps on a different machine connected to a different ToR switch. This requires invalidating stale aggregators having incomplete aggregation results that are waiting on gradient packets from the failed worker. Such stale entries are invalidated as ATP has checks and balances to overcome switch memory leaks (described in §3.7). Also, the re-spawned worker might be on a different machine and connected to a different TOR switch. This might require changes to the `fanInDegree` field in ATP packet headers. ATP re-triggers dynamic job-setup phase (described in §3.2) after any such failure event and the PS re-initiates gradient aggregation by sending an ACK on the new multicast tree that includes the re-spawned worker for the earliest sequence number that is yet to be aggregated.

A.4 Dealing with Stragglers

A slow worker or a slow link can slow down the whole training process. However, this is an artifact of synchronous training and not an issue with ATP. Note that because ATP alleviates network bottlenecks as it aggregates gradients in the network, it reduces the likelihood of network-induced stragglers.

A.5 Comparison to Ring All-reduce Approach

Ring all-reduce uses one-to-one communication in each training iteration/round. With this communication pattern, ATP does not have any opportunity to assist. A drawback of ring all-reduce is that the amount of data that each worker sends and receives is higher and is $4(n-1)\frac{|U|}{n}$, where n is the number of workers, $|U|$ is the total number of bytes to be aggregated. With ATP the amount of data that each worker sends and receives is $2|U|$; the amount of data that each PS sends and receives is $2|U|/m$ where m is the number of PSs. Quantitatively, this indicates that ring all-reduce generates more network traffic than ATP, which may congest the network for other running applications in the cluster. We compare ATP against the ring all-reduce approach (**Horovod RDMA** and **Horovod TCP**) in Figure 8, demonstrating that ATP is better than ring all-reduce for training popular models.

B Addtional details on dealing with floating point

ATP converts floating point values to integers by multiplying with a scaling factor to enable gradient aggregation on programmable switches.

B.1 ATP's Choice of Scaling Factor

The choice of the scaling factor is crucial. A smaller scaling factor can lead to rounding-off a lot of digits after the decimal point and a greater loss in precision. Theoretically, the precision loss due to this conversion is bounded by $\frac{n}{s}$ (Theorem 1 in [56]), where n is the number of workers and s is the scaling factor. A large scaling factor can lead to aggregation of large integers and may cause overflow at the switches. Theoretically, there is no overflow if gradient values are less than an upper bound $B = \frac{2^{31}-n}{ns}$ (Theorem 2 in [56]).

Prior work [56] relies on empirical measurements to find this upper bound B for gradient values of a popular suite of ML training jobs and chooses a scaling factor $s = \frac{2^{31}-n}{nB}$. The precision loss with this choice of scaling factor is bounded by $\frac{n^2B}{2^{31}-n}$.

There are two drawbacks with this approach. First, the guarantee of no overflows with this approach strictly relies on obtaining an accurate estimate of the upper bound B on gradient values. This makes the approach expensive to accommodate jobs that train new and unseen models as empirically determining the value of B for a new model requires an end-to-end training run. Second, the scaling factor decreases as the value of B and the number of workers n increase, which leads to an increasing loss of precision. With 100 workers and $B = 200$, the value of the scaling factor is $\sim 10^5$ and the maximum loss of precision is $\sim 10^{-5}$ per gradient value. We sample gradient values below 10^{-5} in some epochs across models and find that there are 18% such gradient values for ResNet50, 42.8% for VGG16 and 39.8% for AlexNet. These values will experience completed loss of precision, hurting model accuracy upon training.

With ATP, we choose a high scaling factor so as to minimize loss of precision. We note that 32-bit floating point representation provides the precision of 7 decimal digits [7]. Thus, to provide an equivalent precision, ATP chooses the scaling factor of 10^8. The decoupling of the scaling factor from B completely eliminates the first drawback and partially eliminates the second drawback described above. However, a large scaling factor can lead to an overflow of aggregated gradients.

There are broadly two mechanisms to overcome overflows: proactive and reactive. ATP chooses a reactive mechanism (§3.8) because there are drawbacks to using a proactive mechanism that we discuss next.

B.2 Pitfalls of Proactive Overflow Mechanism

The proactive mechanism prevents overflows from ever occurring at the switch. It determines a maximum gradient value threshold B such that, as long as gradient values aggregated in the switch are less than B, any overflow is avoided. The value of B ($= \frac{2^{31}-n}{ns}$) is computed at each worker during job initialization. Packets with gradient values $\leq B$ are aggregated in the switch, while workers flag packets with gradient values $> B$. Packets with this flag are not aggregated in the switch and are eventually aggregated at the PS. It is worth noting that this mechanism naturally fits with the best-effort aggregation service provided by ATP and cannot be implemented in the prior SwitchML work.

Job	False Positive Packets
ResNet50 - 8 Workers	37,002
ResNet50 - 16 Workers	231,528
ResNet50 - 32 Workers	1,156,126
ResNet50 - 64 Workers	14,602,998

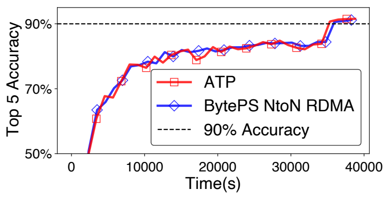
Table 1: Average false positive packets per epoch increase as workers increase.

Unfortunately, a proactive mechanism severely limits the opportunities for in-network aggregation. With 100 workers and a scaling factor of 10^8, the value of B is 0.2. To highlight this, in Table 1, we measure the number of false positives in a single epoch when training a ResNet50 model. We classify a gradient packet with a particular sequence number as a false positive if the gradient value in that packet is $> B$ but the aggregated gradient value for that sequence number from all the workers does not overflow the integer bound (2^{31}). Each false positive packet could have been aggregated in the switch and consumed in the network, but instead with the proactive approach ends up traversing the network from the worker all the way to the PS. As seen in Table 1, the number of false positive packets gets worse as we increase the number of workers in a ResNet50 job. This trend applies to all models, although the magnitude of false positive packets might change. This is not desirable because a higher number of false positive packets means that we lose out on in-network aggregation opportunities and that the number of packets traversing the link to the PS increases. Thus, increasing false positive packets leads to an increased likelihood of incast which only snowballs as the number of workers in the job increases.

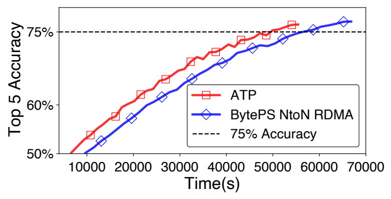
For these reasons, in ATP, we avoid using a proactive mechanism and use a reactive mechanism for overflow correction (§3.8).

B.3 Dynamic Scaling Factor

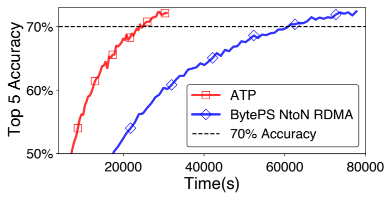
A static scaling factor can, in the worst case, lead to a very high overflow correction frequency when the magnitude of gradient values from all workers is large. In such a case, reducing the value of the scaling factor will reduce the overflow correction frequency and will also not lead to a loss of precision (as the gradient values are large). Thus, a possible optimization would be to make dynamic adjustments to the scaling factor in reaction to the current range of gradient values in recent iterations, especially for the case when the number of workers n is large. In our evaluations with popular models and the scale at which these distributed models are trained today, we do not see the need for such dynamic



(a) ResNet101



(b) VGG19



(c) AlexNet

Figure 18: Time to accuracy

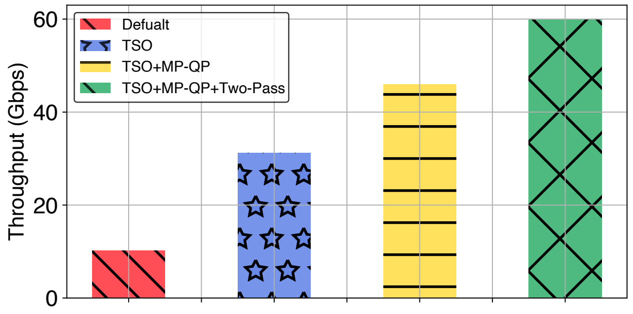


Figure 19: Throughput with different hardware acceleration.

scaling.

However, dynamic scaling adjustments may be beneficial in the future as new large models emerge and training jobs scale-out even further. Inspired by SwitchML [56], workers compute the scaling factor for each gradient in the next window with $s = \frac{2^{31}-n}{nb}$ (Theorem 2 in [56]), where b is the gradient value. Workers pick the minimal scaling factor in a packet, which guarantees there is not any overflow within a packet. When a tensor aggregation starts, a gradient packet piggybacks this value of the subsequent packet, which is sent after receiving the current packet's corresponding parameter packet. The switch computes the minimal scaling factor among workers, which will be carried to the PS. PS can instruct workers to use this scaling factor only when the overflow happens frequently. Note that the first gradient packet aggregation for each tensor can overflow with this approach. SwitchML sends an empty packet only containing the scaling factor at the beginning of the aggregation for each tensor, which causes one RTT waste and the finishing time of a tensor can be doubled if the tensor is small. ATP chooses the reactive approach, where the cost is one RTT in the worst case that all the gradient aggregations in the first window overflow. We leave this as future work.

C Additional Implementation Details

This section supplements the description of ATP's implementation in §4.

To increase the packet size processed by the switch, ATP programs the switch to process each ATP packet twice, which we refer to as **two-pass**. The first half of the gradients in the packet are aggregated in the first pass and the second half are aggregated in the second pass. Instead of using two end-to-end (all the way from ingress on a port to egress to a port) pipelines, which requires two ports to recirculate the packets, ATP uses the *resubmit* and *recirculate* features together, which allows to re-process the same packet (except the last packet that completes gradient aggregation) only in the ingress pipeline of the switch. This avoids using an additional port at the egress pipeline.

Recall that ATP drops the first $n-1$ packets (where, n is number of workers) after aggregation at the switch, and writes the aggregated results from the aggregator to the n-th worker's gradient packet, where n is the number of workers. ATP uses *resubmit* for the first $n-1$ gradient packets after the first pass. As the name suggests, this takes the packet from the end of the ingress pipeline after the first processing pass, left shifts the packet using the 'force_shift ingress' feature on the parser to drop the data part that has been aggregated in the first pass, and immediately resubmits this packet to the ingress pipeline for a second pass to process the second half of the packet. However, we can not apply the resubmit feature to the last gradient packet because ATP writes the first half of the aggregated results to the last packet during the first pass and a resubmit with left shift will lead to a loss of this aggregated result. To deal with this, ATP avoids using the resubmit feature for the last packet. Instead, we use 'recirculate' to enable a second end-to-end pass for the second half of the last packet. This approach requires only one additional port to recirculate the last packet. If the next generation of programmable switches is able to process larger packet sizes or the NIC supports higher packet processing rate, ATP will not require a two-pass implementation at the switch.

D Additional Evaluation Details

This section provides additional evaluation results (§D.1, §D.3) and supplements some existing results (§D.2).

D.1 Small Packets Optimizations

ATP applies multiple hardware acceleration techniques to boost throughput with small-packets as mentioned in Section 4. We demonstrate the effectiveness of each optimization by measuring the throughput of the microbenchmark using configurations that incrementally add more optimizations. We launch two hosts attached to a single switch, one as a worker, and the other as PS. The switch logic of ATP is forwarding packets from the worker to the PS. The PS copies the received data from the receiver memory region to the sender memory

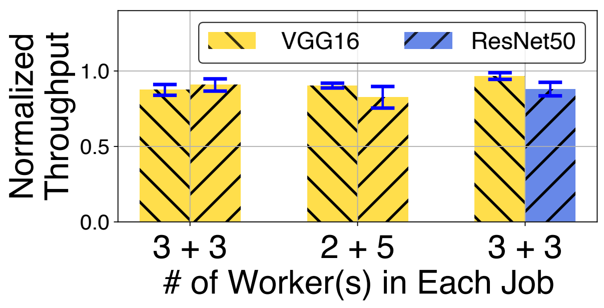


Figure 20: Multiple jobs.

region and then sends the data back to the worker.

Figure 19 shows the network throughput gains as we progressively add optimizations. TSO provides the biggest benefit, and increases performance by $3X$. Incorporating MP-QP (§4) at the end hosts further increases performance $1.47X$. Finally, adding aggregation based on two-pass implementation at the switches increases the throughput by a final $1.3X$. Overall, ATP's **TSO+MP-QP+Two-Pass** optimizations provide a $6X$ throughput improvement over no hardware acceleration (**Default**). We also notice that throughput does not double between **TSO+MP-QP** and **TSO+MP-QP+Two-Pass** when packet size doubles. This occurs because throughput is bottlenecked by the memory copy at the PS and workers [4].

In summary, ATP is able to achieve high throughput with small packets by utilizing recent advances in hardware acceleration.

D.2 Switch Resource Sharing

In this section, we study ATP's dynamic aggregator allocation approach and observe the impact of the number of workers and the model size on the performance of contending jobs. We launch two DT jobs on ATP under various settings.

First, we launch two VGG16 jobs each with 3 workers. Figure 20 shows the training throughput normalized against the job in isolation. We see equal throughput, across identical jobs (same model and number of workers), indicating equal sharing of switch resources. Also, the throughput reduction for each of the two jobs is only 10%.

Second, we launch two VGG16 jobs with 5 and 2 workers each. The job with more workers performs slightly worse than that with fewer workers (145.6 image/s for 5 v.s. 159.3 image/s for 2 workers).

Third, we launch a VGG16 and a ResNet50 job and observe only 5% and 10% throughput reduction respectively; reduction for the large model (5% in VGG16) is less than that for the small model (10% in ResNet50).

In summary, ATP's switch resource sharing is equitable for similar workloads. In the case of different workloads it tends to favor jobs using fewer workers and larger models.

[4]The memory copy overhead can be eliminated by proper memory alignment. We leave this improvement for future work.

D.3 Time To Accuracy

Figure 18 illustrates the TTA training curve for ResNet101, VGG19 and AlexNet with the same setting from §5.3. ResNet101 (Figure 18a) does not see a noticeable speed up ($1.01X$ in ATP) as it is a compute-intensive model similar to our measurements in §5.3 (ResNet50 and ResNet152). The results for VGG19 (Figure 18b) and AlexNet (Figure 18c) reflect $1.2X$ and $2.39X$ speed up, which is consistent with the training throughput presented in §5.2.1 for communication-intensive models.

On the Use of ML for Blackbox System Performance Prediction

Silvery Fu
UC Berkeley

Saurabh Gupta
UIUC

Radhika Mittal
UIUC

Sylvia Ratnasamy
UC Berkeley

Abstract

There is a growing body of work that reports positive results from applying ML-based performance prediction to a particular application or use-case (*e.g.,* server configuration, capacity planning). Yet, a critical question remains unanswered: does ML make prediction simpler (*i.e.,* allowing us to treat systems as blackboxes) and general (*i.e.,* across a range of applications and use-cases)? After all, the potential for simplicity and generality is a key part of what makes ML-based prediction so attractive compared to the traditional approach of relying on handcrafted and specialized performance models. In this paper, we attempt to answer this broader question. We develop a methodology for systematically diagnosing whether, when, and why ML does (not) work for performance prediction, and identify steps to improve predictability.

We apply our methodology to test 6 ML models in predicting the performance of 13 real-world applications. We find that 12 out of our 13 applications exhibit inherent variability in performance that fundamentally limits prediction accuracy. Our findings motivate the need for system-level modifications and/or ML-level extensions that can improve predictability, showing how ML fails to be an easy-to-use predictor. On implementing and evaluating these changes, we find that while they do improve the overall prediction accuracy, prediction error remains high for multiple realistic scenarios, showing how ML fails as a general predictor. Hence our answer is clear: ML is not a general and easy-to-use hammer for system performance prediction.

1 Introduction

Performance prediction has long been a difficult problem, traditionally tackled using handcrafted performance models tailored to a specific application [26,27,43,44,53,56,61,62] or use-case[1] [23,30,55,58]. However, this approach is tedious, doesn't generalize, and is increasingly difficult given the growing complexity of modern systems. Ideally, one would want a predictor that is **accurate, general, and easy to use**. By *general*, we mean an approach that applies to a broad range of applications and a broad range of use-cases; by *easy to use*, we mean an approach that can be applied without requiring detailed knowledge of the application internals or use-case.

[1]Throughout this paper, we use the term *use-case* to refer to an application of prediction such as scheduling (*e.g.,* where/when to run jobs), configuration (*e.g.,*, how many workers or how much memory to use), or capacity planning (*e.g.,* determining what server configurations to purchase).

Given the success of machine learning (ML) in many domains, it is natural to think that ML might offer a solution to this challenge: *i.e.,* that an ML model can learn the relationship between a system's externally observable features and its resultant performance *while treating the application as a black-box*. The ability to treat the application as a black-box and remain agnostic to the prediction's use-case would enable a predictor that meets our goals of generality and ease-of-use.

But does ML deliver on this promise? Several recent efforts have applied ML to predict or optimize performance [22, 37, 47, 57, 60]; these report positive results and hence one might assume that the answer to our question is "yes." However, as we discuss in §9, these effort focus on specific applications, models, or use-cases and hence do not shed light on our question of broad generality and ease-of-use. In this paper, we take a first step towards filling this gap, empirically evaluating whether ML-based prediction can simultaneously offer high accuracy, generality, and ease-of-use.

The first step in such an undertaking is to define a methodology for evaluation. As we shall show, evaluating the accuracy of a model's prediction is subtle particularly when prediction fails because in such cases we need to understand the cause of failure: was it a poor choice of model? was the model poorly tuned? or was the application's performance somehow fundamentally not predictable? In other words, we need a methodology that allows us to both, evaluate the accuracy of a predictor and *attribute* errors in prediction.

This observation led us to define a methodology for systematic evaluation and analysis of ML-based predictors that, as we detail in §2, provides us with two bounds: a lower bound on the prediction error that *any* model can hope to achieve, and a more realistic bound that is based on the best prediction made by the ML models that we consider. We apply our methodology to evaluate 6 different ML models (*e.g.,* k-nearest neighbors, random forest, neural networks) in predicting performance for 13 real-world applications (*e.g.,* Tensorflow, Spark, nginx) under a range of test scenarios (*e.g.,* predicting the impact of dataset size).

Our first key finding is that *irreducible* prediction errors are common (§4). In particular, we find that the majority of our applications exhibit a high degree of performance variability that cannot be captured by any black-box parameters and that manifests even in best-case scenarios (*e.g.,* running an identical configuration of the application, on identical hardware, with no contention for resources). E.g., we show that, even across identical runs, the performance of the JVM Garbage Collector (GC) varies between two modes depending on the precise timing of GC events (§5). Because of this variability, our lower bound on prediction error is non-trivial: we show that no application consistently achieves a lower bound on prediction error that is <10% and many applications fare far worse; *e.g.,* the lower bound on prediction error in memcached is >40% in ∼20% of our prediction scenarios. Borrowing the terminology of the ML community [34], we say that the prediction error that results from this variability is "irreducible" as it stems from behavior that cannot be modeled or controlled, and hence cannot be learned. Irreducible error fundamentally limits the accuracy that *any* ML model can achieve.

We further find that, while the root cause of irreducible error varies across applications, a common theme is that they stem from design decisions that were made to optimize performance, efficiency or resilience but in the process led to a fundamental trade-off between predictability and these other design goals (§5).

These findings suggest a clear negative result for our goal of an ML-based predictor that is both accurate and general. So, where do we go from here? A natural follow-on is to ask whether we can usefully relax our goals of generality and ease-of-use - *i.e.,* make some assumptions about application design or prediction use-cases that would improve predictability.

The second part of our paper explores this question. We do so from two different vantage points: that of the application developer and that of the operator who is using predictions for some operational task.

Our *developer-centric* exploration asks: if developers expose knobs that give operators the *option* to disable design features that lead to irreducible errors, how much would this improve the accuracy of ML predictors?

Our *operator-centric* exploration asks: if we assume operators can accommodate some notion of uncertainty in how they use the predictions, then could ML meet our goals for a useful predictor?

We note that both of the above represent a non-trivial compromise on our original goal and, perhaps more importantly, in neither case do we address the question of *how* developers/-operators would make such changes nor the impact that such changes might have on other design goals such as efficiency, resilience, etc. Instead we are merely asking whether making these changes would improve predictability.

Our findings on this front are mixed. We find that, in both cases, our relaxed assumptions do significantly improve predictability in our best-case scenarios, but we continue to see prediction fail in a non-trivial fraction of our more realistic tests. E.g., in our best-case setup, the lower bound on prediction error is now <6% in >90% of our test cases but, in our more realistic tests, 3 (out of 13) applications see error rates >30% in ∼10% of test cases.

While not the clean result that one might have hoped for, they reinforce that ML-based performance prediction is best applied with a scalpel rather than a hatchet. In this sense, our study mirrors the extensive literature in applied machine learning that explores the trade offs between black- *vs.* gray-box learning. While this trade off has been studied (and debated!) in many other domains [35, 38, 54], to our knowledge we are the first to do so for performance prediction in systems.

Thus unlike many NSDI papers, our contribution lies not in the design and implementation of a particular system but instead in triaging and critically examining the role of ML for managing system performance. Specifically:

• We provide the first broad evaluation of the generality of ML-based prediction, showing that blackbox prediction is often fundamentally limited and expose why this is the case.
• In light of these limitations, we propose and empirically evaluate two complementary approaches aimed at broadening the applicability of ML-based prediction and show that these approaches alleviate but don't eliminate the above limitations.

We view our study as a first step and recognize that it must be extended to more applications and models before we can draw final conclusions on the generality of ML-based performance predictors. We hope that our methodology and results provide the foundation for such future work.

2 Methodology

Our methodology is based on two tests and two predictors: the Best-Case (BC) and Beyond Best-Case (BBC) tests, plus the Oracle and Best-of-Models predictor. In what follows, we first define our metrics and parameters followed by these tests and predictors.

2.1 Metrics and Parameters

We test prediction accuracy by comparing an ML model's prediction to the true performance measured in our experiments. We measure quality of predictions using the root mean square relative error (rMSRE) which is computed as:

$$rMSRE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\frac{Y_i - f(X_i)}{Y_i}\right)^2}, \quad (1)$$

where n is the number of points in the test set for a given prediction scenario and (X_i, Y_i) is a test sample where Y_i is the true measured performance and f is the function learned by the ML model that, given a test feature of value X_i, predicts the performance as $f(X_i)$. rMSRE is a common metric used in regression analysis [34] and in prior work on performance prediction [60]. Note that rMSRE measures the true error in predicted performance - *i.e.,* comparing predicted to actual performance. It is possible that a predictor with high rMSRE is still "good enough" for a particular use-case[2] and, indeed, it is common in papers that focus on specific use-cases to evaluate prediction in terms of the benefit to their use-case. However, given our goal of generality and remaining agnostic

[2]For example: say that a predictor must pick between two configurations c_1 and c_2 where c_1 leads to a JCT of 10s and c_2 to 100s, then even a predictor with an rMSRE of 50% would successfully pick the right configuration.

to the specifics of a use-case, we believe rMSRE is the correct metric for a *predictor* and one can separately study what prediction accuracy is required for a target *use-case*.

We consider the following three classes of parameters that impact an application's performance[3]:

(i) **Application-level input** parameters capture inputs that the application acts on; *e.g.,* the records being sorted, or the images being classified. We consider both the size of these inputs and (when noted) the actual values of these inputs.

(ii) **Application-level configuration** parameters capture the knobs that the application exposes to tune its behavior – *e.g.,* the degree of parallelism, buffer sizes, etc.

(iii) **Infrastructure** parameters capture the computational resources on which the application runs – *e.g.,* CPU speed, memory size, whether resources are shared vs. dedicated.

These parameters correspond to what-if questions that are likely to be of practical relevance: *e.g.,* predicting how performance scales with input data size, under increasing parallelism (executors), or with more infrastructure resources. We note that the above parameters, when used as features for our ML models, capture the application as a black-box, together with the infrastructure it runs on. We also note that these are static parameters known prior to running the application, in contrast to runtime metrics and counters such as a task's CPU or cache utilization. While runtime counters are widely used for monitoring performance, relying on them for prediction limits our potential use-cases to scenarios where prediction is invoked after the application is *already running* rather than at the time of planning, placement, or scheduling. Moreover, it is unclear how one might *use* a predictor based on runtime counters to manage performance: *e.g.,* even if an operator can predict that a CPU utilization value of C leads to a desired performance target, she must still know how to set system parameters in order to achieve the desired CPU utilization. In other words, runtime counters are *measures* not parameters.[4] Hence, in this paper, we assume the ML models cannot use runtime counters as features for prediction.

2.2 The Best-Case (BC) Test

Our BC test is designed to give a predictor the "best chance" at making accurate predictions. It does so by making very strong assumptions on both the systems and ML front, as we describe below.

(1) ML front: simplifying the predictive task. The BC test makes two assumptions that greatly simplify the prediction task. First, in all the data given to the model (training and test), only a single parameter is being varied and that parameter is the *only* feature on which the model is trained. In other words: say we ask an ML model M to predict an application's performance for a particular configuration c_i that is defined by k parameters: i.e., $c_i =< p_1 = X_1, p_2 = X_2, p_3 = X_3, \ldots, p_k = X_k>$, where the p_i are parameters and X_i are parameter values. In the BC test, we give M a training data set in which only one parameter - say p_2 – is varied and all other parameters are set to the test value; *i.e.,* all training data will come from runs where p_2 is varied while $p_1 = X_1, p_3 = X_3, \ldots, p_k = X_k$. We call this our **one-feature-at-a-time** assumption.

Second, the model's training data always includes datapoints from the scenario it is being asked to predict. I.e., continuing with the above example, when predicting the application's performance for an input configuration $<p_1 = X_1, p_2 = X_2, p_3 = X_3, \ldots, p_k = X_k>$, not only do we enforce the one-feature-at-a-time assumption, we also require that the training set for the task include datapoints with $p_2 = X2$. Hence, the training set contains data points from an identical configuration to the one the model is being asked to predict! For example, if we ask the model to predict the time it takes to sort a dataset of size 5GB, the training set already includes times for sorting the same 5GB dataset. We call this the **seen-configuration** assumption since it ensures that, during training, the ML model has already "seen" the configuration it is being asked to predict.

We emphasize the extreme simplification due to the above assumptions: the complexity of prediction has been reduced from understanding the impact of k features to just one feature (*e.g.,* p_2), the training data is deliberately selected to cleanly highlight the impact of this feature, *and* the training set includes data from test configurations that are *identical* to what the model is being asked to predict!

(2) Systems front: best-case assumptions. Given our assumptions so far, the only reason prediction might be non-trivial is if we see variable performance across repeated runs of the application even with a fixed configuration of parameters. That software systems may exhibit variable performance is well recognized in the systems community with two commonly cited reasons for this: (i) contention for resources that an application might experience when it shares physical infrastructure with other applications/tenants [33,40,49,52] and (ii) variability that arises when the processing time depends on the values of data inputs [29] Our assumptions on the systems side aim to remove the likelihood of such variability. To avoid variability due to contention, we run our workloads on *dedicated* EC2 instances [1], and only run a single experiment at a time on a given server. We call this our **no-contention** requirement. We are still left with the possibility of contention within the datacenter network.Our workloads are not network-heavy and, with one exception (nginx, discussed later) none of our workloads appear to be impacted by contention with other apps over network bandwidth and hence we optimistically conclude that network contention is unlikely to have affected predictability for our workloads. We still cannot *entirely* avoid contention, however – *e.g.,* some of our apps use shared cloud

[3]We define these precisely in the context of each application in §3.

[4]I.e., one would need to predict how a parameter impacts the runtime metric in addition to how the runtime metric impacts performance. This might be appropriate for certain use-cases that include long-running jobs, *e.g.,* [61], and where we cannot directly predict how parameters impact performance but, for now, we focus on understanding whether the more general and direct/simple approach works.

services such as the S3 and DNS. Nonetheless, we believe the scenario we construct is far more conservative than what is commonly used in production and hence we view it as a pragmatic approximation of the best case while still running on real-world deployment environments like EC2.

Our second assumption is that, for a given input dataset size, the application's input data is identical across all experiments. I.e., repeated runs of a test configuration act on the same input data. E.g., all training/test data for an application that sorts N records, will involve exactly the same N records. We call this our **identical-inputs** assumption.

The sheer simplicity of our prediction tasks should be immediately apparent:we're essentially asking a model to predict performance for a test configuration that is identical to that seen in training. Our expectation was that, under these conditions, a model should be able to predict performance with a very high level of accuracy - *e.g.,* with error rates well under 5-10% – and if a model cannot accurately predict performance under the above conditions then it is unlikely to be a useful predictor under more realistic conditions.

2.3 The Beyond Best-Case (BBC) Test

In the BBC test, we systematically relax each of the constraints/assumptions imposed in the BC test to study prediction accuracy under more realistic scenarios:

(i) Relaxing the seen-configuration assumption. For this, we perform a leave-one-out analysis in which all data samples corresponding to the configuration on which a model is tested are withheld from the training set. More precisely, for a performance dataset with N configurations $\{c_1, c_2, \dots, c_N\}$, when testing a datapoint with configuration c_i, we use a model trained on a dataset that consists of the $N-1$ configurations *other* than c_i; *i.e.,* our training set excludes all training datapoints for configuration c_i. We perform this N-fold leave-one-out analysis for each of our prediction scenarios.

Note that predictions are now harder as models must learn the *trend* in performance as a function of the parameter being varied. Such predictions are useful in answering *what-if* questions of the form: *"what performance can we expect if we increase the number of workers to 10?"*

(ii) Relaxing the one-feature-at-a-time assumption. We relax our constraint of varying only one parameter at a time and instead enumerate the configuration space generated by simultaneously varying all the features in question and then sample from this space to collect training and test data on which we rerun our predictions. We describe the details of our approach inline when presenting our results.

(iii) Relaxing the no-contention assumption. For this, we repeat the experiments used to collect training and test data but this time do not run our experiments on dedicated EC2 instances. We present additional detail on our experimental setup in the following section.

(iv) Relaxing the identical-inputs assumption. For this, we generate a different input dataset for each datapoint in the training and test set. Section 3 provides additional detail on our input datasets in the context of each test application.

We studied the impact of relaxing each of the above assumptions individually and then all together. In this paper, we present a subset of our results as relevant.

2.4 Predictors

As mentioned, our evaluation considers two predictors.

The Best-of-Models (BoM) predictor. Recall that in order to obtain a broad view of ML-based performance prediction, we consider a range of ML models (§3). For any given prediction test, we compute the rMSRE for each ML model, and define the *best-of-models error (BoM-err)* as the minimum rMSRE across all the models we consider. Thus BoM-err tells us how well *some* ML model can predict system performance. However, if BoM-err is high, we still cannot tell whether this is because of a poor choice or tuning of ML models, or whether performance prediction was inherently hard. For this, what we would like to have is a lower bound on the error rate we can expect from *any* ML model. We achieve this through our *Oracle Predictor*.

The Oracle predictor looks at all the data points in the *test* set that share *the same feature values* as the prediction task, and returns a prediction that will minimize the metric in Eqn. 1 for all these data points. We obtain this by differentiating the expression for the metric in Eqn. 1 with respect to the prediction and finding the global optima for each *unique feature value*, , as below.

$$f_{\text{oracle}}(X) = \left(\sum_{i=1}^{n} \frac{\delta(X_i,X)}{Y_i}\right) \Big/ \left(\sum_{i=1}^{n} \frac{\delta(X_i,X)}{Y_i^2}\right), \tag{2}$$

$$\delta(a,b) = 1 \text{ if } a \text{ is equal to } b\text{, and 0 otherwise.} \tag{3}$$

Note that our features are discrete entities (number of workers, size of dataset, type of instance) and thus the oracle is well-defined. We use O-err to denote the error rate obtained by this Oracle. If there is no variance at all in these data points, the Oracle will achieve zero error. Simply put, O-err quantifies the impact of variance in performance – across multiple runs of the same application and under *identical* configurations – on prediction accuracy.

Clearly, our Oracle predictor is not usable in practice since it is allowed to "peek" at both the test data and the error function; nonetheless O-err is helpful in attribution. Specifically, it gives us a lower bound on the prediction error that *any* ML model could achieve and, in this sense, sheds light on whether performance prediction is at all feasible, *i.e.,* a high O-err perhaps suggests that predicting system performance is "impossible." In addition, a small gap between O-err and BoM-err confirms that our model is well tuned (§3).

3 Test Setup

Our experimental setup comprises two main stages: application profiling and model training. In the profiling stage, an application is run under different configurations to generate a raw dataset. In the training stage, the dataset first undergoes pre-processing (featurization, normalization, and outlier removal), and is then randomly split into two disjoint datasets: the *training* set is used to train models and the *test* set is reserved for model evaluation. The training set is also used for hyper-parameter tuning of ML models. In what follows, we discuss key aspects of each stage. This is not an exhaustive description of our experimental setup – all datasets [6] and tooling [20] from our experiments are publicly available.

3.1 Application Profiling

Applications. Table 1 enumerates the applications used in our study. Our selection reflects multiple considerations including the application's relevance in production environments, diversity (spanning web services, timeseries database, microservices, data analytics, and model serving), and ease of instrumentation. See Appendix A for additional details.

Parameter Values and Performance Metrics. We select values for our three broad classes of parameters as follows:

(i) *application-level input parameters.* We experiment with varying the size of these inputs on a scale of 1 to 10, with scale 1 being the default input size in the workload generator;

(ii) *app-level configuration parameters.* We experiment with varying the number of worker nodes between 1 and 10.

(iii) *infrastructure parameters.* We experiment with 13 different EC2 instance types: from the 150+ instance types offered on EC2 we select the latest generation of the three common instance families (c5, m5 and r5) with four different scales (large, xlarge, 2xlarge, 4xlarge) plus a c4.4xlarge instance for a total of 13 instance types (see [2] for details). This selection matches the instances used in prior work [57, 60].

As listed in Table 1, we use different performance metrics (*e.g.,* job completion time, request throughput) depending on the application. We run each parameter setting 10 times recording the resulting performance. This constitutes our raw dataset. Appendix A and our code repository [20] include additional details on the setup.

3.2 Model Training

We select six ML algorithms: k-nearest neighbors, random forest regression, linear regression, linear support vector machine (SVM) regression, kernelized SVM regression, and neural networks. We select these as they are commonly used in practice and, taken together, represent well the various families of ML techniques: parametric and non-parametric models, linear and non-linear models, discrete and continuous [34]. This diversity is in contrast to prior work that focuses on a single ML technique [22, 37, 47, 57, 60].We follow the standard machine learning practice of k-fold cross-validation (k=3) for setting hyper-parameters. Folds were carefully picked such that they represented the data well and included points for each feature value present in the training set. We searched for regularization parameters (all models); kernels (RBF, polynomial, and sigmoid) and their parameters for SVMs; number and depth of trees for random forests; number of neighbors for nearest neighbors; and number of hidden layers (varying from zero to four), size of layers, activation functions (ReLU, tanh), and learning rates for neural networks. The neural networks we used, MLPs with different non-linearity, are the standard models for treating data of the form in this paper (as opposed to RNN/LSTM/CNNs).

As is standard in machine learning, we pre-process our dataset before training our models. We convert numeric features to have zero mean and unit standard deviation (by subtracting the mean and dividing by the standard deviation, computed per feature channel across the entire training set). We map all categorical features to their numerical index. We also throw out outlier data-points (that have any feature value or performance metric beyond the 99th percentile) from the training set. We use `scikit-learn`, a widely used machine learning tool-set for data preparation and model training [46].

We then train each model to predict the impact of a particular parameter p on performance for a given app A. For this, we select data points corresponding to runs of A in which *all* parameters other than p are fixed. We do a 50:50 random split of the resultant dataset into a training and test set.

4 Results: Existing Applications and Models

We start with the results from our BC test. Recall that each prediction task involves predicting the performance of a given application for a given configuration of: (i) application input size, (ii) number of workers, and (iii) instance types, while subject to the constraints and assumptions presented in §2.2.

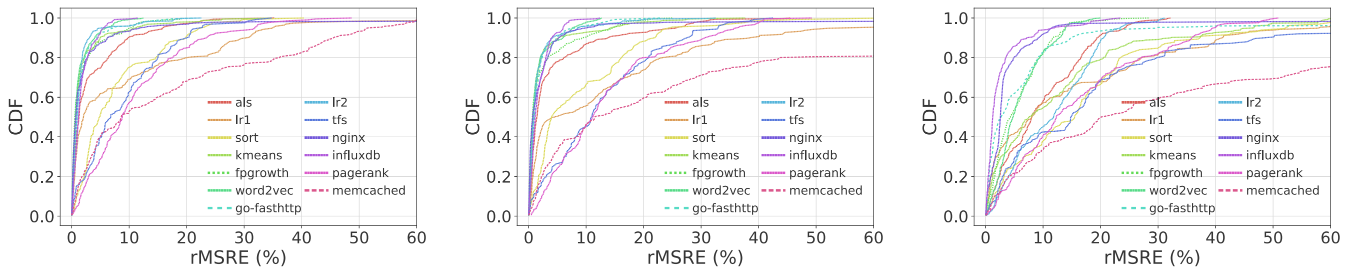
We start by looking at the results for our BC test. Fig.1a plots the cumulative distribution function (CDF) of the O-err for each application across all predictions tasks while Fig.1b shows the same for the BoM-err.[5] Recall that O-err captures the irreducible error inherent to an application (i.e., no ML model could achieve an error rate lower than the O-err) while the BoM-err captures the lowest error rate achieved by some ML model. Given the extreme simplicity of our best-case prediction task, we expected a very high degree of accuracy with an O-err of (say) well under 5% error. To our surprise, our results did not match this expectation. For example: in 5 of our 13 applications, O-err is $>$15% for at least 20% of prediction tasks; for LR1 O-err is $>$30% for 10% of the prediction tasks; for memcached O-err exceeds 40% for approximately 20% of prediction tasks; in fact, no application enjoys an O-err of $<$10% in all prediction scenarios.

The high O-err that we see in our (extremely generous) best-case scenario tells us that many applications suffer from non-trivial irreducible error which fundamentally limits our ability to achieve high prediction accuracy under the general black-box conditions that we desired. Given that we fail even at the BC test, we focus next on understanding the sources of this irreducible error. But first, a few additional observations:

[5]We show a detailed breakdown of error based on the feature being predicted in Appendix C.

Framework	Application/Description	Input Workload	Input Parameter	App. Config. Parameter	Infra. Parameter	Metric
Memcached [12]	Disributed in-memory k-v store	Mutilate [13]	value size	# servers	inst. type	mean query lat.
Nginx [9]	Web server, LB, Reverse Proxy	Wrk2 [5]	req. rate	# servers	inst. type	median req. lat.
Influxdb [15]	Open source time series database	Inch [10]	# points per timeseries	# servers	inst. type	mean query lat.
Go-fasthttp [7]	Fast HTTP package for Go	wrk2 [5]	# conn.	# servers	inst. type	median req. lat.
Spark [3]	*TeraSort:* sorting records	TeraGen	# records	# executors	inst. type	JCT
	PageRank: graph computation	GraphX SynthBenchmark [8]	# vertices			
	LR1: logistic regression	MLLib examples	# examples			
	LR2: logistic regression	Databricks Perf Test [16]				
	KMeans: clustering					
	Word2vec: feature extraction					
	FPGrowth: data mining					
	ALS: recommendation					
TensorFlow [17], Kubernetes [11]	*TFS:* Tensorflow model serving	Resnet examples [18]	# conn.	# servers	inst. type	requests/sec

Table 1: Applications, input workload, parameters, and metrics used in our study.



(a) CDF of O-err in the BC test

(b) CDF of BoM-err in the BC test

(c) CDF of BoM-err in the BBC test

Figure 1: O-err and BoM-err for existing applications and ML models.

(i) As one might expect, prediction accuracy deteriorates as we move from the BC to the BBC test. As a sample result, Figure 1c shows the CDF of the BoM-err for our BBC test when relaxing our seen-configuration assumption: for 6 of our 13 applications (*vs.* 3 in the BC test), the 80%ile BoM-err exceeds 20% (for 1 app, it exceeds 60% error!)
(ii) As the CDFs suggest (and as we validated on individual data points), BoM-err on our BC test tracks O-err very closely, confirming that the higher-than-expected errors arise from irreducible causes, as opposed to poor ML engineering.
(iii) The error rates vary across applications even when they use the same framework (*e.g.,* Spark-based sort vs. LR2). This reinforces the importance of considering a range of applications when evaluating ML for performance prediction.

5 Tackling Irreducible Error

The high irreducible errors that we saw in the previous section tells us that, even for a fixed configuration of parameters, there are times when an application's performance was so variable that *no* predictor could predict performance with high accuracy. While it is well recognized that software systems can experience variable performance due to various runtime factors, we were surprised to see the extent of this variability even in our best-case scenario. Recall that, in our best-case scenario, we are essentially just repeatedly running an identical software stack, on identical hardware, with identical data inputs, and without contention on our servers. What can cause performance to vary significantly across runs? In particular, we were interested in understanding whether the sources of variability could be captured by features that are known prior to runtime. If so, we could hope to achieve higher prediction accuracy by simply adding more features to our ML models.

Unfortunately, we find that this variability stems from design choices – often optimizations – the effect of which could *not* have been captured by system parameters known prior to running the application. In this section, we briefly summarize our findings (§5.1) and then discuss their implications (§5.2).

5.1 Root-Cause Analysis

Table 2 summarizes the findings from our root cause analysis. Each entry summarizes the root cause we discovered, the applications it impacted, the trade-off that eliminating this root cause would impose, the manner in which we modified applications to eliminate/mask their impact, and the burden associated with undertaking such analysis. We stress that the modifications we make are ad-hoc hacks intended only to verify that we correctly identified our root-causes; as we discuss later, we do *not* view them as the desirable long-term fix.

Finally, we report the person hours it took to identify and validate these root causes. We recognize that any such estimate depends on many nebulous factors such as our familiarity with the codebase and experience in performance analysis. We present it merely as anecdotal evidence that analyzing performance in modern systems is non-trivial and would benefit from better developer support as we discuss in §8.

To give the reader a tangible sense of these root causes, we first present a longer description of the first two root causes, followed by a very brief summary of the remaining ones. A detailed description – with experimental validation – is in Appendix B, while the following section (§6) evaluates the impact of eliminating these root causes on O-err and BoM-err.

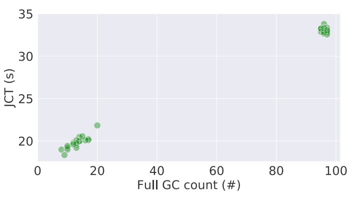
5.1.1 Spark's "Worker Readiness" Optimization

We found that the relatively high O-err in Spark's Terasort application stems from a dynamic sizing optimization in Spark. By default, Spark launches an application once at least 80% of its target worker nodes are ready, and the application partitions the input dataset based on the number of workers ready at this time. This optimization ensures resilience to failure and stragglers, but leads to variable parallelism and hence JCTs. This variability leads to irreducible errors and cannot be captured by any input parameters/features as the exact degree of parallelism is affected by small differences in worker launch times which cannot be predicted prior to runtime.

Disabling this optimization lowered Terasort's average O-err from 12.6% to 2.6% in our predictions that involve varying the instance type. While this optimization also affected other Spark apps (*e.g.,* PageRank), its impact there was small.

5.1.2 Adaptive Garbage Collection in the JVM

Our analysis showed that LR1's error stems from an optimization in the Java Virtual Machine's (JVM) garbage collector (GC). Specifically, we found a positive correlation between the number of "full" GC events (explained below) and JCT, leading to the bimodal behavior in the figure above.



As a performance optimization, the JVM GC divides the memory heap into two regions – young and old – and typically tries to constrain garbage collection to just the young region. However, in situations where the memory heap is consistently low on free space, the JVM GC runs a "full" collection that operates over the entire heap space (young and old regions). To determine whether a full GC is needed, the JVM maintains a *promotion estimate* metric. Our analysis revealed that due to minor differences in the timing of memory allocation and GC events, the promotion estimate ends up just above the threshold (triggering a full GC) in some runs and just below it in others, leading to the bimodal behavior (see Appendix B.2). Once again, since the exact mode being triggered depends on the detailed timing of runtime events, this effect could not have been captured by any input parameters/features known prior to actually running the job.

We verify our analysis by rerunning our experiments with an extra 50MB of memory which ensures that the promotion estimate remains below the threshold in all runs. This eliminates the high-mode runs, reducing average O-err by 9× from 18.2% to 2.1% in our predictions that involve varying the number of workers.

5.1.3 Other Root Causes

We briefly mention the remaining entries in Table 2; detailed tracing for each is in Appendix 9.

HTTP Redirection and DNS Caching in S3. We found that multiple applications built on Amazon's S3 storage service suffered variable performance that arose from the DNS-based resolution of S3 object names; some name resolution requests experienced HTTP redirects while others didn't depending on the detailed timing of when DNS updates were propagated.

Imperfect Load-Balancing. We observed high irreducible errors when predicting the request throughput of a Tensorflow Serving (TFS) cluster. We run TFS within a Kubernetes cluster and found that this error stems from the randomized load-balancing policy that Kubernetes employs, which leads to an inherent imbalance in the load at each server; when running the cluster at high utilization this imbalance led to some servers being overloaded and the *variability in this imbalance* leads to variations in the overall request throughput.

Non-deterministic Scheduling. We found that independent Spark tasks were being scheduled in different orders across different runs leading to different JCTs. This variability arose because the data structure used to track runnable tasks (Scala's `HashSet`) offers no guarantees on the order of iteration through the set members. Switching to a different data structure eliminated this variability.

Variability in Cloud APIs. Our last two cases of high O-err stemmed from variability across repeated invocations of cloud APIs: memcached was affected by variability in how worker instances were placed (which affects node-to-node latency) while nginx suffered from variability in the default network bandwidth associated with particular instances types. Again, with little visibility into (or control over) cloud API implementations, the impact of this variability could not be predicted by input parameters/features prior to runtime.

5.2 Implications and Next Steps

At a high level, many of the irreducible errors we encountered may be attributed to two common design techniques: the use of randomization (*e.g.,* in load-balancing, scheduling) and the use of system optimizations in which a new mode of behavior is triggered by a threshold parameter (*e.g.,* a promotion estimate, timeouts, a threshold on available workers). Some of these design choices can be altered with little loss to design goals such as performance or resilience (*e.g.,* remov-

Root Cause	**Applications Impacted**	**Trade-off**	**Modification**	**Effort to diagnose**
Spark's "start when 80% of workers are ready" optimization	Terasort	Decreased resilience to stragglers and worker failure	Disable optimization	5 person days
Multi-mode optimization in JVM Garbage Collector	LR1	Slower garbage collection	Avoid triggering, or disable, optimization	39 person days
Non-determinism in Spark sched.	PageRank	None	Use deterministic data structure	14 person days
HTTP redirects and DNS caching in S3's name resolution	KMeans, LR2, FPGrowth, ALS	Decreased flexibility[6] (OR slower name resolutions)	Client-side caching of HTTP redirects (OR always redirect)	10 person days
Imperfect load-balancing at high load	TensorFlow serving	Load imbalance when each server has different numbers of workers	Modified load-balancing policy to always favor local workers	7 person days
Variability in implementations of Cloud APIs (EC2)	memcached, Nginx	Cloud APIs expose more information (less flexibility)	Use AWS placement APIs / include inter-node RTTs as ML feature	5 person days

Table 2: Root causes of the irreducible errors we observed in our test applications. Person hours were calculated using the timestamps in our debugging logs and covers the entire process of reproducing observed behavior, adding instrumentation, processing logs, modifying the system to eliminate suspected causes, running tests for validation, etc.

ing the non-determinism in Spark's scheduler). However, for many others, eliminating them would come at some loss in performance/efficiency (*e.g.,* DNS caching improves scalability, partitioning regions offers faster GC). Moreover, because of the recent emphasis on extracting performance, modern systems now make extensive use of such optimizations.

Given the lower bound set by the O-err (as discussed in §2.4) and the underlying root causes, no amount of model modifications or feature engineering would have resulted in significantly better prediction accuracy. So, where do we go from here? We set out to test an ambitious hypothesis: that ML could serve as a general and easy-to-use predictor of system performance. Unfortunately, we found that most of the applications we studied suffered poor prediction accuracy on a non-trivial number of test cases even under extremely favorable test conditions. Moreover, the design choices that led to low accuracy cannot be easily forsaken without impacting other goals such as performance.

Thus a natural follow-on is to ask whether we can usefully relax our goal of generality - *i.e.,* make some assumptions about application design or prediction use-cases that would still allow us to leverage ML-based prediction in a (mostly) general and easy-to-use manner. As mentioned in §1, the remainder of this paper explores this question from the vantage point of application developers (§6) vs. operators (§7).

6 Results: Modified Applications

We now examine the impact of removing the above root-causes on performance predictability. If doing so improves prediction accuracy, then we can envision a workflow where the application developer identifies the root causes of irreducible errors and makes them configurable. For example, the JVM garbage collector (GC) designer can expose a knob to turn off the optimization for reducing full GCs. This, in turn, would enable system operators to disable such techniques depending on their desired trade-off between predictability and other design goals such as performance.

We emphasize that our goal here is not to provide a mechanism for identifying and eliminating sources of irreducible errors. Developing a systematic approach for this is an interesting problem but beyond the scope of this paper. Instead, we focus on the consequences of doing so: *if* these sources are identified and eliminated, to what extent would it improve performance predictability? Further note that, in line with this objective, we focus on evaluating the resultant impact on performance *predictability*, and not on performance itself.[7]

We now (re)evaluate predictability for our BC (§6.1) and BBC (§6.2) tests after applying the "fixes" to remove the root causes of irreducible errors as discussed in §5.

6.1 BC test results

Fig. 2a shows the CDF of the O-err for our modified applications across all prediction tasks. As expected, removing the sources of irreducible errors results in a dramatic reduction in O-err. All applications now have O-err well within 10% for at least 90% of their prediction tasks. In other words, 90%ile O-err is $< 10\%$ for all 13 applications, and in fact, only two applications have O-err $> 6\%$.

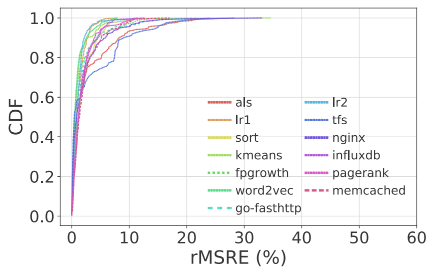
Fig. 2b shows the corresponding CDF for BoM-err. Again, only two applications have 90%ile BoM-err $> 6\%$, of which only one (TFS) has 90%ile BoM-err $> 10\%$. Further observe that the trends for BoM-err closely track those of O-err above, highlighting that ML comes close to achieving the lower bound on prediction errors for our simple BC predictions.

6.2 BBC test results

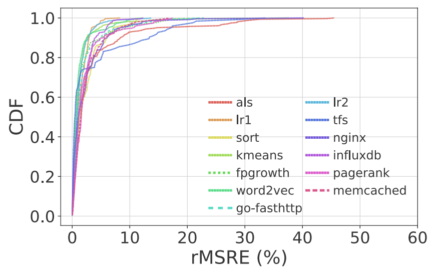
With the promising BC test results, we next turned our attention to BBC tests after application modifications. We systematically relaxed the assumptions of our BC test (as described in §2), and present a subset of results for brevity.

Fig. 2c presents the CDF for BoM-err after we relax the seen-configuration assumption by performing a leave-one-out test as described in §2. Note that, since the test dataset remains unchanged, O-err is the same as in our BC prediction above. Comparing Figures 2b and 2c, we see that the BoM-

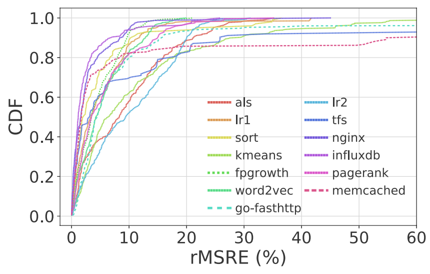
[7]In fact, many of our "fixes" would actually improve performance (*e.g.,* allocating more memory, caching the correct server address) and hence including such results would be misleading!



(a) CDF of O-err in the BC test



(b) CDF of BoM-err in the BC test



(c) CDF of BoM-err in the BBC test

Figure 2: O-err and BoM-err in the BC and BBC test after modifying applications to remove sources of irreducible errors. Note the sharp drop in performance prediction errors, as compared to unmodified applications in Fig. 1.

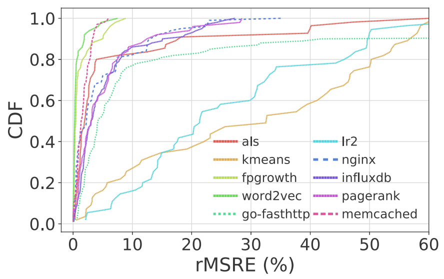


Figure 3: CDF of BoM-err in the BBC leave-one-out test where input data content is not identical, for predictions across varying input data scale. Relaxing the identical-inputs assumption reduces the prediction accuracy for multiple applications.

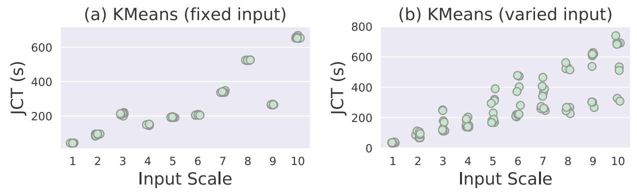


Figure 4: JCTs for KMeans with fixed vs. varied inputs for each value of input scale. KMeans JCT is sensitive to the input dataset.

err is significantly higher for the BBC leave-one-out test than for the BC test. 10 out of the 13 applications exhibit 90%ile BoM-err $>$ 10%. Some degradation in prediction accuracy is expected due to the increased difficulty of the prediction tasks, which now require the ML models to learn the *trend* in performance. What is surprising is that multiple applications experience exceptionally high BoM-err for a significant fraction of prediction tasks. For example, the 90%ile BoM-err of memcached exceeds 60%, while that of KMeans and TFS exceeds 25%. As can be seen, the corresponding 95%ile BoM-err is even higher. We dig deeper into the reasons behind these high errors in §6.3.

While we observe these high prediction errors for only a subset of applications and prediction scenarios, these results emphasize that we cannot simply rely on ML as a general predictor that works across all apps and prediction scenarios.

We next present results from tests in which we relaxed the identical-inputs assumption in addition to relaxing the seen-configuration assumption. In other words, we perform a leave-one-out analysis, but now generate a different input dataset for each datapoint in the training and test set. We do so for 10 applications by varying the random seed in the workload generator, keeping the distribution underlying the input data unchanged. Fig. 3 shows the corresponding CDF for BoM-err across prediction tasks where we vary input scale. We observe that, in general, relaxing the identical-inputs assumption further reduces the prediction accuracy for multiple applications.[8] The applications that are most notably impacted are KMeans and LR2. This is because the performance of these applications is sensitive to the input data. For example, as we show in §6.3, the number of iterations for KMeans, and therefore its JCT, depends on the actual values/content of the input data. This results in a multi-modal behaviour for a given input scale, and thus, high prediction errors. As expected, the corresponding O-err (not shown for brevity) is also high.

Finally, we relaxed our no-contention assumption by repeating the above experiments on non-dedicated (or "shared") instances and found that doing so did not produce statistically meaningful differences in our results. In particular, moving to shared instances resulted in $<$3% degradation in BoM-err across all scenarios (detailed results in Appendix D.2).

6.3 Deep Dive on high BBC prediction errors

Our BBC leave-one-out tests required the ML models to learn the underlying *trend* in performance as a function of the parameter being varied. We observed that in many cases (with high BoM-err), this trend is inherently hard to predict because it changes too fast (i.e. the underlying function has a high Lipschitz constant). This causes our ML models to generalize poorly [24, 42]. We highlight this phenomena for three applications here, and discuss some of the others in Appendix D.4.

(i) KMeans has a high average BoM-err of 40.4% and 30.4% for prediction across varying input-scale with identical inputs and non-identical inputs respectively. We observed that the

[8]Note that Fig. 3 captures a subset of prediction scenarios (i.e. across varied input scale). This is in contrast to Fig. 2c that captures a wider range of prediction scenarios, which explains any observed deviation from this general trend. Appendix D.3 shows the results for our leave-one-out test with only the predictions across varied input scale, for a more direct comparison.

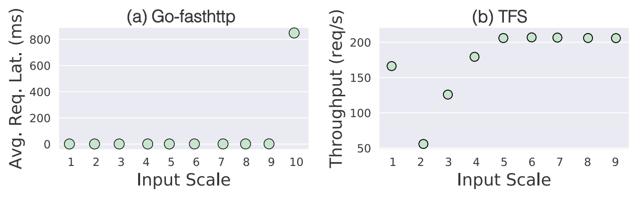


Figure 5: Average request latency of go-fasthttp and TFS at varying input scales. These functions vary too fast, and are hence inherently hard to predict.

performance of KMeans is sensitive to the content of its input dataset. At larger input sizes, Kmeans exhibits multi-modal behavior; specifically, the number of iterations to converge (and hence, the JCT) depends on the input data content (this effect is shown in Fig. 4(b)). This multi-modality impacts O-err and BoM-err in prediction tasks where input content is varied. With identical inputs, this multi-modality impacts the underlying trend as the input scale is varied, making it (surprisingly, even more) difficult to learn. The JCT for an input scale factor of 9 in Fig. 4(a) is an example of this behavior.

*(ii) Go-fasthttp has an exceptionally high average BoM-err of 128.6% for predictions across varying input-scale.*The mean query latency increases dramatically at the highest input scale because the system behavior changes under such high load due to queuing (Fig. 5(a)). This sudden change makes prediction on that value inherently difficult. The BoM-err for go-fasthttp reduces to 4.9% if the test datapoints for the highest input scale are removed from consideration.

(iii) TFS has a higher BoM-err of 52.7% for prediction across varying input-scale. This is again due to the underlying function being difficult to learn, as shown in Fig. 5(b). We are still investigating the root-cause behind this.

6.4 Summary

Application modifications to eliminate sources of irreducible errors do produce a notable increase in prediction accuracy, suggesting that a workflow where application developers provide knobs that give operators the option of disabling these error sources could be a promising direction moving forward. However, there are important concerns that cannot be neglected: (1) From the viewpoint of predictability, as we move to more realistic BBC scenarios, prediction errors do remain high for certain applications due to the underlying trend being difficult to learn (as illustrated in §6.3). Eliminating irreducible errors via application modification is not sufficient in these scenarios. (2) From the viewpoint of generality and ease-of-use, identifying the root causes of errors and making them configurable imposes a non-trivial burden on application developers; the same is true of asking system operators to reason about the trade-offs between predictability and other goals such as performance. In other words, such changes do weaken the black-box nature of performance prediction that we originally hoped ML-based predictors could provide.

7 Probabilistic Predictions

It may not always be possible to identify and eliminate sources of irreducible errors as required by the previous section. For instance, it might be too time-consuming to do so, or the system may be closed-source and not amenable to modifications. Or, operators may not want to compromise on the benefits of the relevant system optimizations (*e.g.,* experiencing slower garbage collection by disabling the GC optimization). Therefore, we now explore an approach that allows operators to *embrace*, rather than eliminate, performance variability.

Our empirical observations in §5 reveal that the optimizations causing irreducible error often lead to bimodal/multimodal performance distributions. This is the key insight that drives our approach, leading us to hypothesize that a way forward could be to extend ML models to predict not just one performance value, but a probability distribution from which we derive k possible values, with the goal that the true value is one of the k predictions, for a low k.

This immediately raises the question of whether such top-k probabilistic predictions would even be useful to an operator? We believe they can be, depending on the use case. For instance, the operator can use the worst among the k predictions when provisioning to meet SLOs; or they can use the average expectations across the k predictions to pick an initial number of workers in a system that anyway dynamically adapts this number over time; or they can use the probability distribution to compare which system configuration will perform better in expectation when purchasing new servers. Note that, as in §6, this approach also comes with a trade-off – using the worst of k predictions may lead to over-provisioning, or using the expected average may lead to sub-optimal choices.

Our goal here is not to design such use cases of probabilistic predictions, or reason about these trade-offs. We instead focus on the following questions: *assuming* operators could make use of top-k probabilistic predictions: (i) how do we extend ML models to enable top-k predictions, and (ii) is there a small enough value of k that results in accurate top-k predictions? If not, exploring use cases of top-k predictions would be moot.

We thus proceed with discussing how we extended two of our models to predict probabilistic outputs (§7.1), and our prediction results (§7.2).

7.1 Extending ML models

We extend random forest and neural network to predict performance as a probability distribution across k possible outputs. We chose these models as they were most natural to extend.

Probabilistic Random Forests (Prob. RF). Instead of using the average JCT of the training points at the leaf node as the prediction (as is done in conventional decision trees), we use the *distribution* of the JCT data points at the leaf node, modeled using a Gaussian Mixture Model (GMM) [34] with k components. We train this Prob. RF as before, still picking splits that minimize the variance of JCT in child nodes.

Mixture Density Networks (MDNs). We adopt MDNs [25],

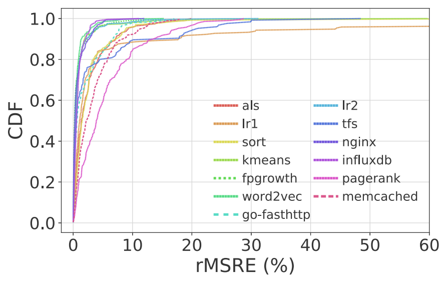


Figure 6: CDF of BoM-err in the BC test where models with prob. outputs are used (k=3). Compare to Fig. 1b.

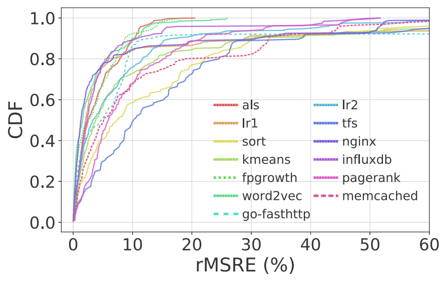


Figure 7: CDF of BoM-err in the BBC leave-one-out test where the models with probabilistic outputs are used (k=3). Compare to Fig 1c.

and modify our neural network to predict parameters for a Gaussian Mixture Model with k components (mean and variance for each component and mixing coefficients). We use negative log-likelihood of the data under the predicted GMM, as the loss function to train the MDN.

We implement Prob. RF based on random forest module in `scikit-learn`, and MDN in TensorFlow [21].

7.2 Top-k Prediction Results

To evaluate predictability with the probabilistic models above, we obtain k predictions from the models as the k different means of the k component GMM, and report the *top-k rMSRE* score, i.e. the rMSRE score of the best prediction among the k predictions made by the model. Such a top-k rMSRE shares the same interpretation as the rMSRE score – in fact, rMSRE scores reported so far can be thought of as top-1 rMSRE scores. In our evaluation, we observed a sharp drop in error rates as we move from a top-1 to top-2 measure and that further improvement plateaus off for $k > 3$. For brevity, we present a subset of our results for $k = 3$.

BC Test Fig.6 shows the top-3 BoM-err under the BC predictions. We see a significant decrease in BoM-err compared to our top-1 prediction presented earlier in Fig.1b. The 90%ile BoM-err is less than 10% for all but two applications. Note that the test data set remains unchanged by our use of probabilistic models, and hence O-err remains as high as in Fig.1a. The reduced BoM-err with top-k predictions shows the promise of this approach in embracing inherent variability.

BBC Test Fig.7 presents the top-3 BoM-err under the BBC test, where we relax the seen-configuration assumption by conducting leave-one-out predictions. While there is an improvement over models that make a single prediction (as in Fig 1c), we note that our multi-modal predictions don't improve performance in cases where the underlying trend is hard to predict for a reason other than multi-modality (*e.g.*, TFS and go-fasthttp).

7.3 Summary

Our findings along this direction are similar to those in §6. While it is possible to reduce prediction errors by extending our ML models to predict top-k performance values, two concerns with regard to generality and ease-of-use remain: (1) Even with top-k predictors, we continue to see scenarios with high error rates when we consider our more realistic BBC tests because the underlying performance trend is difficult to learn. Thus achieving a fully general predictor remains out of reach. (2) The use of top-k predictors complicates the process of applying performance prediction (which compromises ease-of-use) and may lead to sub-optimal decisions; in fact, how to best use such predictions and reason about the resulting trade-offs remains an open question.

8 Takeaways and Next Steps

We set out to evaluate whether ML offers a simpler, more general approach to performance prediction. We showed that: **(1)** Taken "out of the box", many of the applications we studied exhibit a surprisingly high degree of irreducible error, which fundamentally limits the accuracy that *any* ML-based predictor can achieve, and **(2)** We *can* significantly improve accuracy if we relax our goals (accepting the trade offs) and modify applications and/or modify how we use predictions. But even with these relaxations we still see a non-trivial number of predictions with high error rates! E.g., apps where $\sim$10% of the BBC tests have BoM-err $>$ 30-40%.

While ML fails to meet our goal of generality, we did find several scenarios where ML-based prediction *was* effective, showing that we must apply ML in a more nuanced manner by first identifying whether/when ML-based prediction is effective. Our methodology provides a **blueprint** for this, as summarized in Figure 8. Concretely, say that operators want to assess and improve the predictability of a target application. The first step is to run our BC test with the target workloads. If O-err is low, they can continue to the BBC test and check BoM-err. Otherwise, they have two options. The first is to disable any root-causes of variability if possible, rebuild the application, and re-run the BC test. If disabling root-causes is not possible or not desirable, operators can choose to use the top-k predictions. They can also combine both options, reconfiguring the application and using probabilistic predictions.

Even for this more nuanced approach, many open questions remain: (i) do our findings hold beyond the 13 apps and 6 models studied? (ii) how do we design systems to more easily identify sources of irreducible error? (iii) how can developers and operators more easily reason about trade offs between predictability and other design goals? Etc.

Our work provides empirical evidence motivating the above

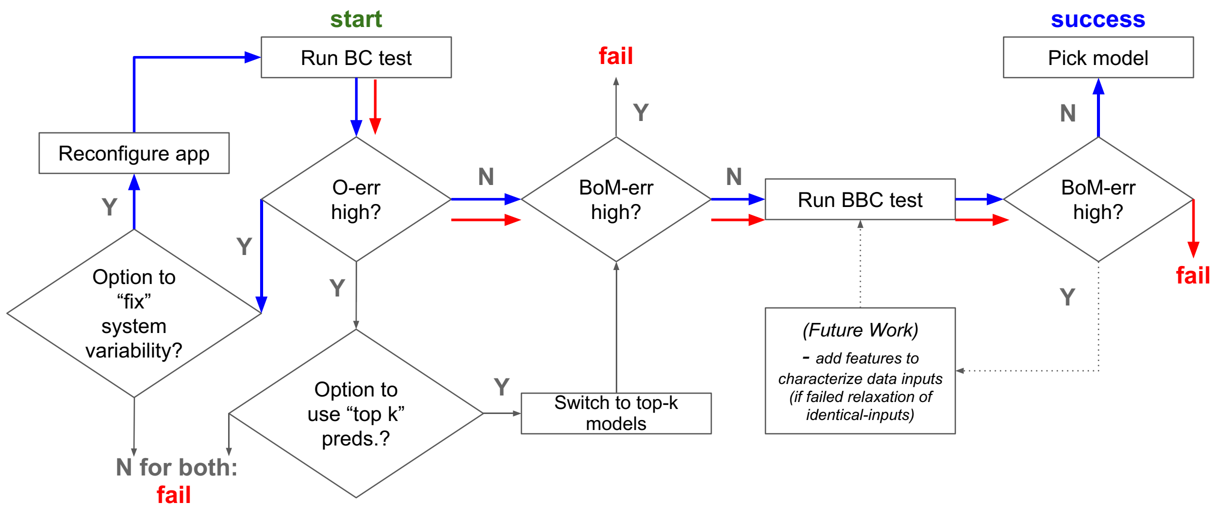


Figure 8: Methodology blueprint. As examples from our findings: Terasort's path (shown in blue) started out with only 73% of prediction tests having an O-err <10% in our BC test but once we eliminated its root-cause of irreducible error (the worker-readiness optimization), 99+% of test cases saw an O-err <10% and the BoM-err even in our BBC test was <10% for ~90% of test cases leading to what we would deem a successful outcome. By contrast, KMeans (shown by the red path) started with a good O-err in our BC test (>90% of test cases had O-err <10%) but ultimately failed when we relaxed our identical-inputs assumption where point >60% of prediction tests had O-err >20%!

questions and a blueprint for how to approach and evaluate them. Overall, we remain cautiously skeptical about the role of ML in predicting system performance. We note that a common thread in the above questions is the need to evaluate predictors in a manner that is systematic and *consistent across studies*. We call on the community to adopt and extend our methodology as the foundation for such evaluation.

9 Related Work

Prior work has explored using ML based performance prediction for tuning and optimizing system configuration. Ernest [57] uses domain expert knowledge to build an analytical model for Spark performance, that is based on transformations and combinations of different features (such as number of cloud instances and input dataset scale), and trains the parameters of this model using ML.

Similar data-driven, gray-box modeling approaches have been applied to predicting and tuning performance for deep learning workloads and scientific computing [47, 48]. Paris [60] is a black-box performance modeling tool for selecting the best instance type by training a Random Forest model for each instance, and profiling unseen test applications on a small subset of instances to feed as input to the model.

Selecta [37] makes innovative use of collaborative filtering to predict performance and select the best-performing storage configuration for data analytics applications. CherryPick [22] explores black-box optimization (Bayesian Optimization) for a guided search towards the optimal cloud configurations without accurately predicting performance. Google's Vizier [31] leverages similar black-box optimization and makes it an internal application service for various workloads. Each of the above focus on answering a specific question with a specific ML technique. Our goal is to understand how ML can be more broadly applied to predicting performance across a range of systems and predictive tasks. We hope that our results, particularly as they relate to our methodology and irreducible error, can be applied to many of the contexts explored in prior work. Monotasks [44] proposes a radically new design for Spark aiming at assisting performance diagnostics and prediction; it does not explore the role of ML for performance prediction.

Similar to our proposal in §7.1, several recent papers recognize that performance is perhaps better represented as a probability distribution. [51] proposes modeling the performance of individual functions/methods as probability distributions and presents automatic instrumentation to capture such distributions. [45] shows how to design cluster schedulers that take as input, distributions derived from historical performance. Our proposal adds a new dimension to this general approach: we show how to extend ML models to generate probabilistic performance predictions.

Google Wide Profiler [36, 50] explores the use of performance counter information collected on an always-on profiling infrastructure in datacenters to provide performance insights and drive job scheduling decisions. As mentioned in §2, our work differs in that we focus on static parameters to enable use-cases where predictions happen before runtime.

Performance variability has been widely reported in contexts from hardware-induced variability [41], to stragglers in batch analytics [59], variability in VM network performance [52], and tail request latencies in microservices [29]. Our work can be thought as complementary: we studied a wide range of applications, report variability even under best-case scenarios, focus on the impact of variability to ML-based performance prediction, and propose systematic approaches to cope with variability.

References

[1] Amazon EC2 dedicated instances. https://aws.amazon.com/ec2/pricing/dedicated-instances/.

[2] Amazon EC2 instance types. https://aws.amazon.com/ec2/instance-types/.

[3] Apache spark. https://spark.apache.org/.

[4] Compute optimized instances. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/compute-optimized-instances.html.

[5] A constant throughput, correct latency recording variant of wrk. https://github.com/giltene/wrk2.

[6] Datasets used in this paper. https://s3.console.aws.amazon.com/s3/buckets/perfd-data.

[7] Fast HTTP package for Go. https://github.com/valyala/fasthttp.

[8] Graphx synthbenchmark. https://github.com/apache/spark/blob/master/examples/src/main/scala/org/apache/spark/examples/graphx/SynthBenchmark.scala/.

[9] High performance load balancer, web server, & reverse proxy. https://www.nginx.com/.

[10] An influxdb benchmarking tool. https://github.com/influxdata/inch.

[11] Kubernetes: Production-grade container orchestration. https://kubernetes.io/.

[12] memcached - a distributed memory object caching system. https://memcached.org/.

[13] Mutilate: high-performance memcached load generator. https://github.com/leverich/mutilate.

[14] Placement groups. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/placement-groups.html.

[15] Scalable datastore for metrics, events, and real-time analytics. https://github.com/influxdata/influxdb.

[16] Spark performance tests. https://github.com/databricks/spark-sql-perf.

[17] Tensorflow serving. https://github.com/tensorflow/serving.

[18] Tensorflow serving on kubernetes. https://www.tensorflow.org/tfx/serving/serving_kubernetes.

[19] Testing the performance of nginx and nginx plus web servers. https://tinyurl.com/yccm2uf6.

[20] Tooling and setup used in this paper. https://github.com/perfd/perfd.

[21] Martın Abadi et al. Tensorflow: Large-scale machine learning on heterogeneous systems, 2015. *Software available from tensorflow. org*, 1(2), 2015.

[22] Omid Alipourfard et al. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *Proc. USENIX NSDI*, 2017.

[23] Vasanth Balasundaram, Geoffrey Fox, Ken Kennedy, and Ulrich Kremer. A static performance estimator to guide data partitioning decisions. In *Proc. ACM PPoPP*, pages 213–223, 1991.

[24] Peter L Bartlett, Dylan J Foster, and Matus J Telgarsky. Spectrally-normalized margin bounds for neural networks. In *Advances in Neural Information Processing Systems*, pages 6240–6249, 2017.

[25] Christopher M Bishop. Mixture density networks. 1994.

[26] Jianhua Cao, Mikael Andersson, Christian Nyberg, and Maria Kihl. Web server performance modeling using an m/g/1/k* ps queue. In *10th International Conference on Telecommunications, 2003. ICT 2003.*, volume 2, pages 1501–1506. IEEE, 2003.

[27] Harshal S Chhaya and Sanjay Gupta. Performance modeling of asynchronous data transfer methods of ieee 802.11 mac protocol. *Wireless networks*, 3(3):217–234, 1997.

[28] Jasmine Collins, Jascha Sohl-Dickstein, and David Sussillo. Capacity and trainability in recurrent neural networks. In *ICLR*, 2017.

[29] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[30] Christina Delimitrou and Christos Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. In *Proc. ACM ASPLOS*, 2014.

[31] Daniel Golovin et al. Google vizier: A service for black-box optimization. In *Proc. ACM KDD*, 2017.

[32] Klaus Greff, Rupesh K Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. LSTM: A search space odyssey. *IEEE transactions on neural networks and learning systems*, 28(10):2222–2232, 2016.

[33] Xinlei Han, Raymond Schooley, Delvin Mackenzie, Olaf David, and Wes J Lloyd. Characterizing public cloud resource contention to support virtual machine co-residency prediction. In *Proc. IEEE IC2E*, 2020.

[34] Trevor Hastie, Robert Tibshirani, Jerome Friedman, and James Franklin. The elements of statistical learning: data mining, inference and prediction. *The Mathematical Intelligencer*, 27(2):83–85, 2005.

[35] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal processing magazine*, 29(6):82–97, 2012.

[36] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Proc. ACM ISCA*, 2015.

[37] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Selecta: heterogeneous cloud storage configuration for data analytics. In *Proc. USENIX ATC*, 2018.

[38] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[39] Jacob Leverich and Christos Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proc. ACM EuroSys*, 2014.

[40] Amiya K Maji, Subrata Mitra, Bowen Zhou, Saurabh Bagchi, and Akshat Verma. Mitigating interference in cloud services by middleware reconfiguration. In *Proc. of the 15th International Middleware Conference*, 2014.

[41] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, and Robert Ricci. Taming performance variability. In *Proc. USENIX OSDI)*, 2018.

[42] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. MIT press, 2018.

[43] Matthias Nicola and Matthias Jarke. Performance modeling of distributed and replicated databases. *IEEE Transactions on Knowledge and Data Engineering*, 2000.

[44] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. Monotasks: Architecting for performance clarity in data analytics frameworks. In *Proc. ACM SOSP*, 2017.

[45] Jun Woo Park, Alexey Tumanov, Angela Jiang, Michael A Kozuch, and Gregory R Ganger. 3sigma: distribution-based cluster scheduling for runtime uncertainty. In *Proc. ACM EuroSys*, 2018.

[46] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[47] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proc. ACM EuroSys*, 2018.

[48] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: scalable analytics on serverless infrastructure. In *Proc. USENIX NSDI*, 2019.

[49] Xing Pu, Ling Liu, Yiduo Mei, Sankaran Sivathanu, Younggyun Koh, and Calton Pu. Understanding performance interference of i/o workload in virtualized cloud environments. In *Proc. IEEE Cloud*, 2010.

[50] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE Micro*, 30(4):65–79, 2010.

[51] Daniele Rogora, Antonio Carzaniga, Amer Diwan, Matthias Hauswirth, and Robert Soulé. Analyzing system performance with probabilistic performance annotations. In *Proc. ACM EuroSys*, 2020.

[52] Ryan Shea, Feng Wang, Haiyang Wang, and Jiangchuan Liu. A deep investigation into network performance in virtual machine based cloud environments. In *Proc. IEEE INFOCOM*, 2014.

[53] Harish Sukhwani, José M Martínez, Xiaolin Chang, Kishor S Trivedi, and Andy Rindos. Performance modeling of pbft consensus process for permissioned blockchain network (hyperledger fabric). In *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*, pages 253–255. IEEE, 2017.

[54] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[55] Omesh Tickoo, Ravi Iyer, Ramesh Illikkal, and Don Newell. Modeling virtual machine performance: challenges and approaches. *ACM SIGMETRICS Performance Evaluation Review*, 37(3):55–60, 2010.

[56] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *Proc. ACM SIGMOD*, 2017.

[57] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: efficient performance prediction for large-scale advanced analytics. In *Proc. USENIX NSDI*, 2011.

[58] Yangyang Wu and Ming Zhao. Performance modeling of virtual machine live migration. In *2011 IEEE 4th International Conference on Cloud Computing*, pages 492–499. IEEE, 2011.

[59] Neeraja J Yadwadkar, Ganesh Ananthanarayanan, and Randy Katz. Wrangler: Predictable and faster jobs using fewer resources. In *Proc. ACM SoCC*, 2014.

[60] Neeraja J Yadwadkar, Bharath Hariharan, Joseph E Gonzalez, Burton Smith, and Randy H Katz. Selecting the best vm across multiple public clouds: a data-driven performance modeling approach. In *Proc. ACM SoCC*, 2017.

[61] Ji Zhang et al. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proc. ACM SIGMOD*, 2019.

[62] Xiaolan Zhang, Giovanni Neglia, Jim Kurose, and Don Towsley. Performance modeling of epidemic routing. *Computer Networks*, 51(10):2867–2891, 2007.

Appendix

A Test Applications: Additional Detail

We include **Memcached**, a popular in-memory key-value store. We use the mutilate [13, 39] memcached load generator which generates realistic memcached request streams and records fine-grained latency measurements. We vary the key size (corresponding to the varying input scale prediction), number of memcached servers (varying number of servers), and the server instance type, while keeping the other workload parameters in mutilate as fixed values[9]. We report results on predicting the average query latency.

We include **Influxdb**, a widely used timeseries database. We use its official benchmarking tool Inch as the client. Inch sends write queries to Influxdb and reports the time it takes to complete each write query. We vary the number of points per timeseries written per query (varying input scale prediction), the number of Influxdb servers, and the server instance type. We report results on predicting the query latency.

We include **nginx**. We use wrk2, an http benchmarking tool used by nginx's official benchmark reports [19]. It sends http requests and reports fine-grain latency and throughput measurements. We vary the per-client request rate, number of worker processes (each worker process is a single-threaded process; the maximum number of worker processes we vary to is equal to the number of cores of the instance), and the server instance type. We report results on predicting the median request latency.

We include **go-fasthttp**, a high-performance HTTP package for building REST services. We use the wrk2 tool to generate HTTP loads again. We vary the request rate, number of server instances, and the server instance type. We report results on predicting the median request latency.

We include multiple **Spark-based applications** spanning diverse forms of computation: sorting, graph computation, classification (two different implementations), data mining, recommendation etc.

We include **Tensorflow Serving (TFS)**, a high performance model serving system, which we orchestrate using Kubernetes (k8s). Our TFS setup resembles that of other (e.g., Web) serving systems in which a front-end load balancer (we use EC2 ELB) spreads client requests across a backend of (model) serving instances.

B Understanding Irreducible Errors – Details

B.1 Spark's "Worker-readiness" Optimization

Details included in the main text.

B.2 Multi-mode Optimization in JVM GC

We elaborate on the details of how the JVM adapative GC leads to variable application performance in the context of our Spark Logistic Regression (LR1) workload.

[9]See https://github.com/perfd/perfd/blob/master/apps/memcached/manifests/perf_predict/suite_1/keySize.yaml for an example setup.

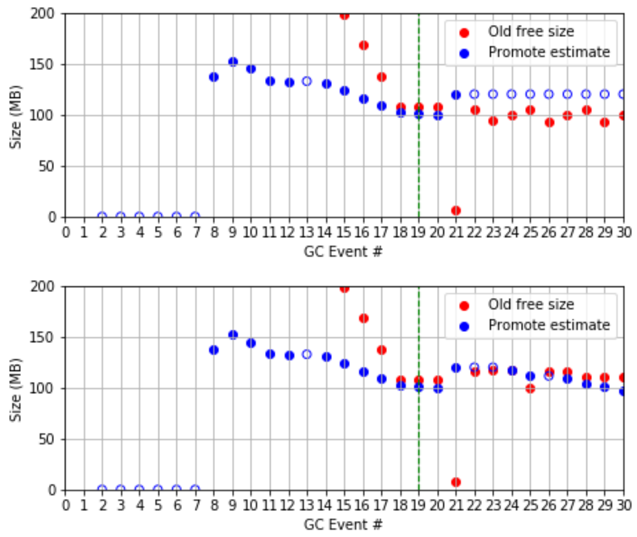


Figure 9: High-mode (top) and Low-mode (bottom) trajectories for promotion estimate and free space in old region during the first 30 garbage collections. Hollow blue dots depict major/full GCs and solid blue dots depict minor GCs.

§5.1.2 revealed a positive correlation between the number of *full* GCs (explained below) and JCT with a distinct bimodal behaviour. We now describe how this bimodality arises. For this, we first explain relevant aspects of the GC mechanism in Java Virtual Machine (JVM) that Spark uses.

JVM divides the Java memory heap into two regions – a *young* region to allocate new objects, and an *old* region to accommodate 'old' objects that have survived multiple GC rounds. It supports three different types of GCs over these regions: (i) *minor GC* on the young region, (ii) *major GC* on the old region, and (iii) *full GC* over the entire heap space (both young and old regions), with the surviving objects residing in the old region.

In the face of heap space shortage, JVM first runs a minor GC on the young region, deleting the unused objects and promoting the old objects (that have survived multiple GC rounds) to the old region. A minor GC triggers a major GC if the old region has too little free space to hold the promoted objects. If a minor GC constantly triggers a major GC, the garbage collector can save time by skipping the minor GC (and the ensuing major GC), and directly running full GC over the entire space. JVM implements this adaptive skipping of minor GC as a performance optimization. To estimate whether a minor GC would trigger a major GC, it maintains a *promotion estimate*, calculated as the moving average of the number of promoted objects after each round of minor GC. If the promotion estimate is higher than the amount of free space in the old region, JVM GC skips the minor GC and runs a full GC directly.

We now show how this adaptive skipping behavior impacts performance predictability. We randomly pick one sample each from the slower "high-mode" runs (whose JCTs sit at the top-right corner of the figure in §5.1.2) and the faster "low-mode" runs (whose JCTs sits in the bottom-left corner of the figure in §5.1.2). The top and bottom plots in Fig.9 show the promotion estimate values and the amount of free space in the old region observed for the first 30 GC events in the two sample runs respectively.

Fig.9(top) reveals that beyond GC event #21, the promotion estimate in the high-mode run remains greater than the amount of free space in the old region (even after event 30, beyond what the plot shows). This results in consecutive full GCs (shown as hollow blue dots). Note that since no objects get promoted in a full GC, the JVM does not update the promotion estimate.

Fig.9(bottom) shows how the low-mode run escapes from such a fate: the promotion estimate is higher than the amount of free old space only for a few GC events, and stays lower than that beyond GC event #26. Consequently, the low mode run sees more minor GCs than full GCs (shown as solid blue dots). Since a full GC, which scans objects across a larger memory space, is significantly more time consuming than a minor GC, the perpetuation of full GCs has a large impact JCT (witnessed in the scatter-plot in §5.1.2).

Fig.10 shows the corresponding time-series of GC events along with events that mark the start and finish of individual tasks (for clarity, we only show the time range of 20-26s, which captures the regime where the two runs start deviating from one another). Careful observation of Fig.10 reveals that the GC behaviour of the two runs begins to diverge around time 24s, when two of tasks finish slightly earlier in the high mode run than in the low mode run. We checked that around this time, the two modes see a difference of about 10MB in the amount of free old space (which is rather small compared to the total heap size of 1GB).

Such subtle differences cannot be determined without knowing the runtime state of the garbage collection and the Spark application.

B.3 Non-determinism in the Spark Scheduler

Our analysis revealed that PageRank's high O-err stems from non-determinism in the Spark scheduler. Specifically, in a job with multiple stages, a stage A may be *independent* of stage B in the sense that A's tasks can be scheduled to run whether or not B's tasks have completed. Our traces showed that independent stages were being scheduled in different orders in different runs leading to different JCTs. This arises because of how the Spark scheduler tracks dependencies between stages. Spark's scheduler maintains a graph that captures the dependencies between stages; a stage v's parents in the graph are those stages that can only be scheduled after v is complete. Spark's scheduler uses the Scala `HashSet` data structure to track the parents of a stage. When the search propagates to the parents, the order of visiting these parents can be inconsistent across runs because `HashSet` makes no guarantees on the order of iteration through the set members.

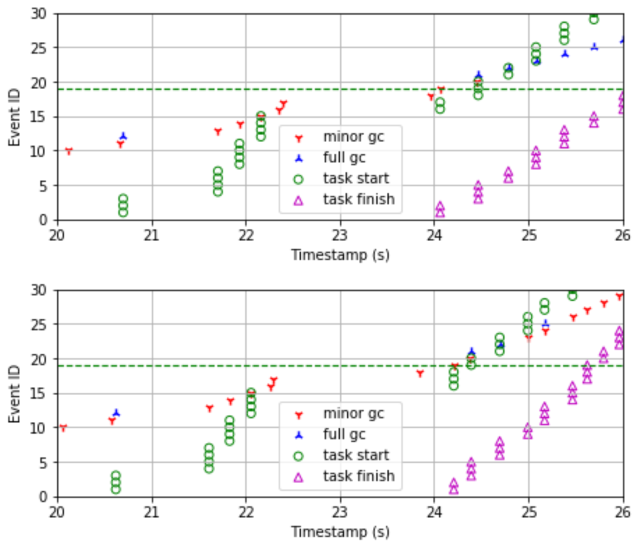


Figure 10: High-mode (top) and Low-mode (bottom) trajectories for the run-time events. Note that unlike Fig.9, the x-axis unit here is the wall time.

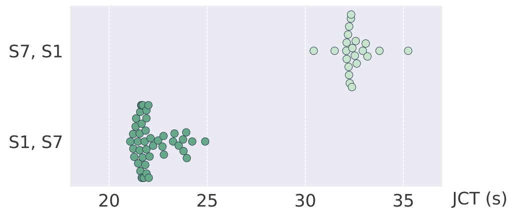


Figure 11: Bimodal performance that results from scheduling independent stages in different orders.

Fig.11 shows the effect of this non-determinism. Stages 1 and 7 are two independent stages and the figure plots the JCT for runs where stage 7 is scheduled before stage 1 (dots on the bottom-left) and vice versa (dots on the top-right). We see that the difference in scheduling order corresponds to bimodality in the JCTs. We emphasize that this bimodality could not be predicted prior to runtime as it depends on the runtime state of the HashSet structure rather than any static input parameters.

Setup	Naive-err	BoM-err	O-err
Baseline	44.7%	19.2%	18.2%
+Sched. mod.	40.7%	5.7%	3.2%

Table 3: Prediction error before vs. after.

To verify our analysis, we modify the Spark scheduler such that the order of scheduled stages is consistent across runs with identical configurations (replacing HashSet with a LinkedHashSet). Table 3 shows that our modification (the "+Sched. mod." row) reduces the O-err by more than 5.7x bringing the error to under 4%.

B.4 HTTP Redirection and DNS Caching in S3

Multiple applications in our study – KMeans, LR2, FPGrowth, and ALS – experienced irreducible errors due to name resolution in Amazon's S3 storage service. We explain this effect in the context of the KMeans application.

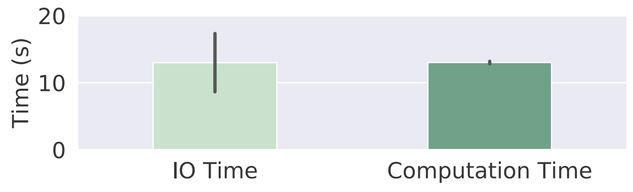


Figure 12: Breakdowns of Spark KMeans job completion time as time spent on cloud storage IO (writing the results to S3) and on computation for 100 experiment runs.

When triaging the overall JCT, we often found that, on average, the computation time was on par with the S3 IO time but that the standard deviation for the latter was 24× times higher than the former (Fig.12). Analysis of the application's runtime logs for KMeans revealed that the *variance* in performance stems from I/O to Amazon's S3 storage service. This is shown in Fig.12 which breaks down the overall JCT into computation time and S3 IO time. We see that the computation time has little variance while the S3 IO time has substantial variance (between 17.5s and 9s).

Further instrumentation revealed a correlation between the time spent on S3 IO and the number of HTTP redirection events, and that the latter varied across multiple runs of identical test configurations.

S3 is a distributed storage service in which objects are spread across multiple datacenters. When a new object "bucket" is created, its DNS entry points to a default datacenter location. If a user creates a bucket in a datacenter other than the default one, then a request to that bucket is first sent to the default server, which responds with an HTTP redirect containing a new URL that resolves to the datacenter where the bucket is located.

Such redirection continues until the DNS entry for the bucket is correctly updated.[10] The S3 buckets in our experiments were created in a different datacenter from the default one, leading to such HTTP redirects.

Fig.13 illustrates the above behavior during one of our experiments. We plot two values over time: (a) the number of distinct S3 servers that our application connected to, and, (b) the time spent on S3 IO. We observe three distinct phases to S3 performance, demarcated by green lines in Fig.13.

Interestingly, we observe a significant intermediate period where the two values oscillate. We found that this oscillation arises because AWS load-balances DNS requests across a pool of DNS servers and the servers in this pool converge to the new DNS entry at different times. In summary, we see

[10]AWS states that DNS entries can take up to 24 hours to be fully propagated; we observed average delays of approx 4 hours. AWS also recommends that clients not cache the redirect URL as its validity is only temporary.

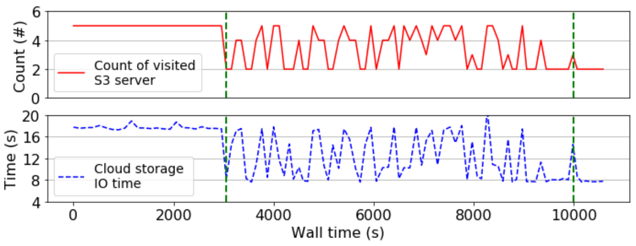


Figure 13: The number of S3 servers visited during the cloud storage IO (top) and the total IO time (bottom). This is plotted over wall-clock time during multiple rounds of experiments.

variable performance even for identical test configurations due to the distributed and eventually-consistent nature of object name resolution in S3. To validate our analysis, we repeated our experiments after modifying the KMeans application to cache the correct bucket location after the first redirection event (yes, ignoring AWS' recommendation). Table 4 shows that this modification dramatically reduces prediction error to an O-err of 1.0% and a BoM-err of 1.1%.

Setup	Naive-err	BoM-err	O-err
Baseline	22.7%	16.0%	15.2%
Correct bucket loc.	5.6%	1.1%	1.0%

Table 4: Prediction error before vs. after.

One might ask whether it is possible to predict the impact of AWS's DNS service on applications. An in-depth exploration of this question is beyond the scope of this paper. However, we note that doing so appears impractical, if not infeasible, since we would have to know how an application's lifetime overlaps with the DNS timers not just for a single server but for an entire pool of servers, as well as precisely knowing how DNS requests are load-balanced across this pool.

B.5 Imperfect load-balancing at high load

We observed high irreducible errors when predicting the request throughput provided by a TFS cluster under increasing numbers of workers (servers). Our analysis revealed that this high error stems from how client requests are load-balanced across TFS servers. We run TFS servers within a Kubernetes cluster and hence incoming client requests undergo two levels of load balancing. First, the AWS Elastic Load Balancer round-robins incoming client connections across the k8s nodes. Next, each k8s node load balances incoming RPCs across the TFS instances.

We found that the irreducible errors in TFS stem from the second stage of load-balancing. The load-balancing within k8s is based on selecting a TFS server at random, leading to some inherent imbalance in the load at each server. When running the TFS cluster at high utilization (as our experiments do), this imbalance means that some servers are already running at capacity while others are running below the maximum request rate they can sustain. The *variability in this imbalance* leads to variations in the overall request throughput; e.g., in repeated runs of one identical test setup, we saw aggregate throughput vary between 80-140 req/sec.

Setup	Naive-err	BoM-err	O-err
Baseline tput.	123.5%	26.7%	11.8%
-Rnd LB tput.	163.2%	7.6%	6.7%

Table 5: Prediction error before vs. after.

To verify this effect, we reconfigure the k8s load-balancer to always direct client RPCs to the local TFS server instance. In effect, this disables the k8s load-balancer which is acceptable for our test purposes. Table 5 shows that this modification substantially reduces the prediction error. We found a similar effect when predicting request latency and also found that incorporating heavy-hitter clients exacerbated the error due to randomized load-balancing.

B.6 Variability in Implementations of Cloud APIs

We observed high O-err in the memcached and ngnix applications, both stemming from variability in how the cluster is configured and the limited control/visibility that default cloud APIs provide for this process.

In the case of memcached, the variability stemmed from how our worker instances were being placed within the cloud infrastructure. Our experiments used EC2's default VM placement which offers no guarantee on the proximity between our allocated instances and hence our node-to-node latency varied across runs. This in turn led to variable memcached read latencies, as we found memcached read times are dominated by the network latency between the client and server in our setup. We found that switching to an API that consistently places a set of instances close together reduced the O-err from 36.9% to 2.4% (varying input scale) [14].

For nginx, the high O-err stemmed from variability in the default network bandwidth associated with smaller instance types. Specifically, with EC2's "c5" instances types, those smaller than c5.9xlarge are by default assigned "up to" 10Gbps network bandwidth while c5.9xlarge instances are assigned exactly 10Gbps. For smaller instance types we found that EC2 *occasionally* throttles network bandwidth and the degree of throttling varies across runs (see Appendix B.6). The response time in nginx is also dominated by the network latency between client and server and hence this variability in throttling leads to unpredictable request latencies. Repeating our scale-out and input-scaling tests with a c5.9xlarge instance (instead of our previous default of c5.xlarge for the client and c5.4xlarge for the server) reduced the O-err from 15.6% to 2.0% under varying number of workers.

Fig.20 depicts the varying available bandwidth across different runs of the same configuration (specifically, the number of worker processes in varying number of worker process scenario). We measured the node-to-node bandwidth using iperf between the instance running the client (wrk2) and the instance running the nginx right before the experiments were run. The results show that despite the configurations (instance types, number of worker processes) remaining the same, the available bandwidth can vary. We conjecture this is a result of the bandwidth allocation mechanism AWS employs, which throttles bandwidth usage based on an estimation of the average network utilization of the instances [4].

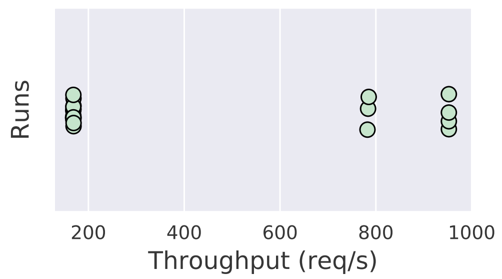


Figure 14: Available bandwidth (measured with `iperf`) between client and server nodes across different runs of the same configuration.

Note that it is arguable whether the above sources of error are "irreducible". On the one hand, it might be possible to augment cloud APIs or incorporate the parameters of cloud APIs as a feature in our ML models, however, doing so is likely to come at a loss in flexibility and efficiency for cloud operators. E.g., one might envisage placement APIs that guarantee *consistent* proximity between instances, but the achievable proximity is likely to vary depending on the number and type of instances (we will see an example of this in §**??**). Similarly, if the AWS throttling that we observed is determined by current network load,[11] then it is not clear how one might capture the impact of throttling prior to runtime. We leave the question of how cloud providers might augment their APIs to aid performance predictability as a topic for future work.

C Best-Case Analysis - More Details

Fig.15 and Fig.16 describe the per-app average prediction error rates for best-case experiments.

D Beyond Best-Case Analysis - More Details

D.1 Leave-one-out - More Details

Fig.17 and Fig.18 describe the per-app average prediction error rates for leave-one-out experiments.

D.2 Dedicated Instance vs. Shared Instance

Fig.20 shows the impact of changing dedicated instance type to shared instance type for the Memcached and Spark sorting applications. We see that overall the relaxation does not have a statistically meaningful impact on the predictability of these two applications.

[11] AWS documentation does not elaborate on the exact conditions under which they implement throttling [4]

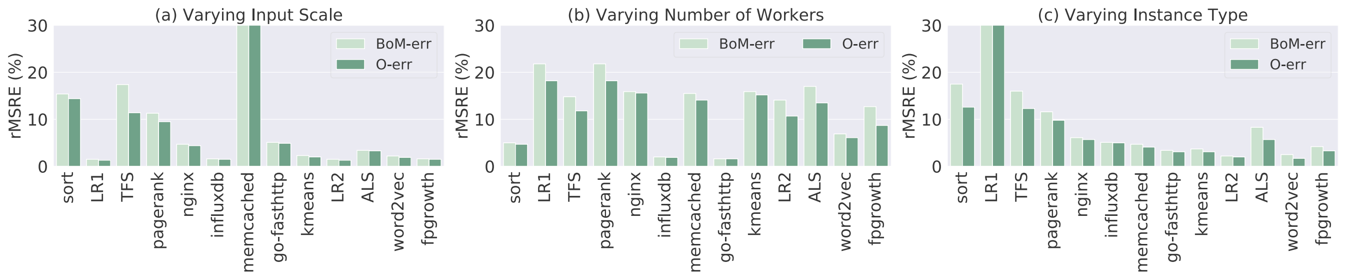


Figure 15: Best-case prediction errors for different applications under (a) changing input-scale, (b) scaling-out the number of workers, and (c) scaling-up the instances; before system modifications.

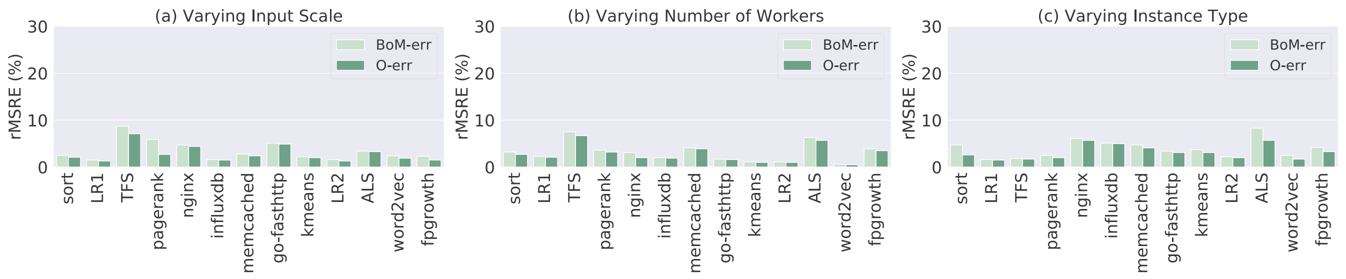


Figure 16: Best-case prediction errors for different applications under (a) changing input-scale, (b) scaling-out the number of workers, and (c) scaling-up the instances; after system modifications.

D.3 Leave-one-out Test with only Varied Input Scale

Fig.21 shows the CDFs of BoM-err in the BBC leave-one-out test, for only the predicitons across varying input data scale.

D.4 High BoM-err in BBC after App. Modifications - Other Applications

(i) TFS has a higher BoM-err of 52.7% for prediction across varying input-scale. This proved to be because the trend in TFS throughput under varying input scale is inherently hard to predict. The underlying function – shown in Fig.5(a) – has a high Lipschitz constant (*i.e.,* it changes too fast), which causes the trained model to have poor generalization [24, 42]. We have not yet been able to determine the root cause for this odd trend in TFS.

(ii) Memcached has a high BoM-err of 50.9% for prediction across scaling up instance types. Recall from §5 that memcached performance is sensitive to the node-to-node latency and hence we modified our experiments to use AWS's cluster placement group API that ensures nodes in a cluster are consistently placed close to each other. This avoids variability in placement across multiple runs of an identical test configuration. However, what we discovered is that while AWS's placement API is consistent in the proximity at which it places instances of a particular type, this proximity may vary across *different* instance types, making it hard for our ML models to learn a trend across instance types (as is needed for our leave-one-out analysis).

E Comparing ML Models

So far, we focused on the error of the best-performing ML model for a given task. We now compare across our six ML models. We do so in the context of our leave-one-out analysis since it is both more challenging and realistic. Table 6(top) reports the error rate that each model achieves for each of the three prediction scenarios, averaged across all applications. We normalize each of these error rates by the lowest average error for the corresponding prediction scenario. Therefore, a normalized value of 1.0 indicates that the corresponding model performed the best (on average) for the corresponding prediction scenario. Table 6(bottom) similarly reports the normalized error rates for each model and each application, but now averaged across all prediction tasks.

It is important to note that our analysis uses only out-of-the-box versions of each ML model and applies the same hyperparameter tuning approach to each of them. This is in contrast to many prior studies (in systems contexts and beyond) in which a particular model is often specialized and carefully tuned to achieve high accuracy for a given prediction task [28, 32, 60]. In this sense, our results can be viewed as a "fair" comparison of models while at the same time we acknowledge that the performance of any individual model/prediction could probably be improved through careful tuning. We view our approach as establishing a useful baseline and conjecture that there is value to a low-effort ML predictor, especially given the rapid pace of evolution in modern software services.

We draw the following conclusions from our results:

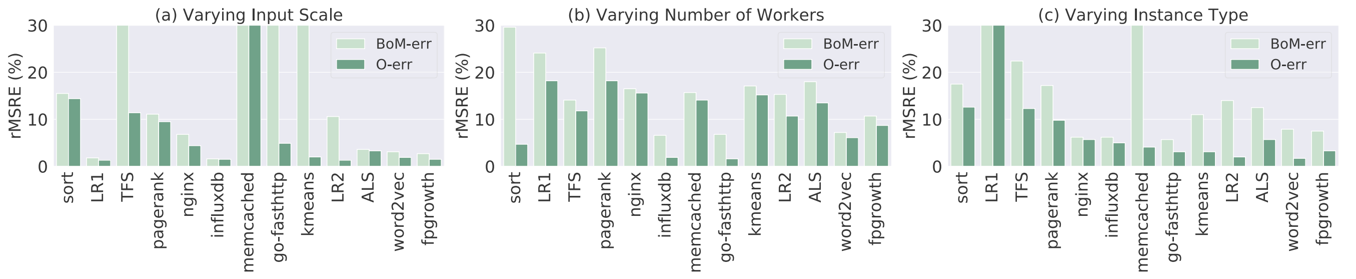


Figure 17: Leave-one-out prediction errors for different applications under (a) changing input-scale, (b) scaling-out the number of workers, and (c) scaling-up the instances; before system modifications.

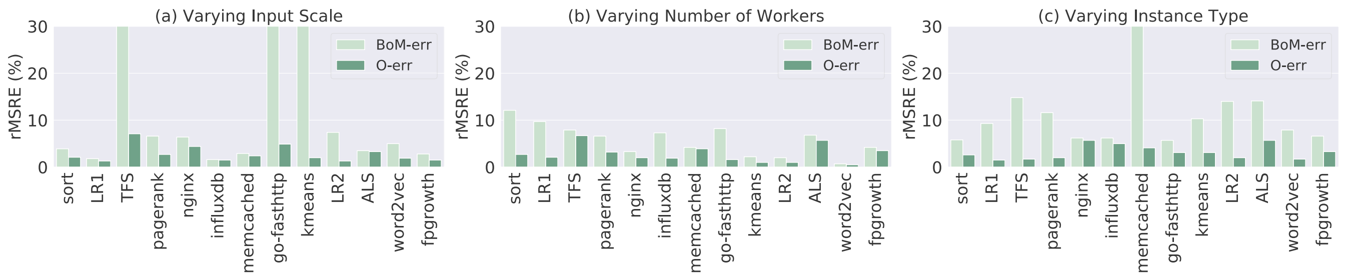


Figure 18: Leave-one-out prediction errors for different applications under (a) changing input-scale, (b) scaling-out the number of workers, and (c) scaling-up the instances; after system modifications.

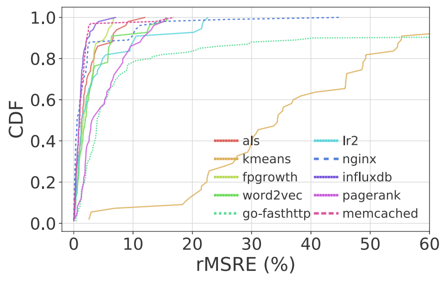


Figure 19: CDF of BoM-err in the BBC leave-one-out test where input data content is identical, for predictions across varying input data scale

(i) The key takeaway from our results is that there is no clear winner: no ML model performs the best across all prediction scenarios and across all applications. This validates the merits of studying a range of ML models and applications so that we can understand how to best apply or combine various models.

(ii) There is no clear loser either: each model performs the best for at least one prediction scenario or application.

(iii) Even though there is no clear winner, neural network often results in the best performance or has error rates close to the best performing model. There were a few exceptions where neural network resulted in 2× higher error rates than the lowest error. We found that using a different hyper-parameter tuning methodology reduced the neural network error in these cases. This confirms the common wisdom that neural net-

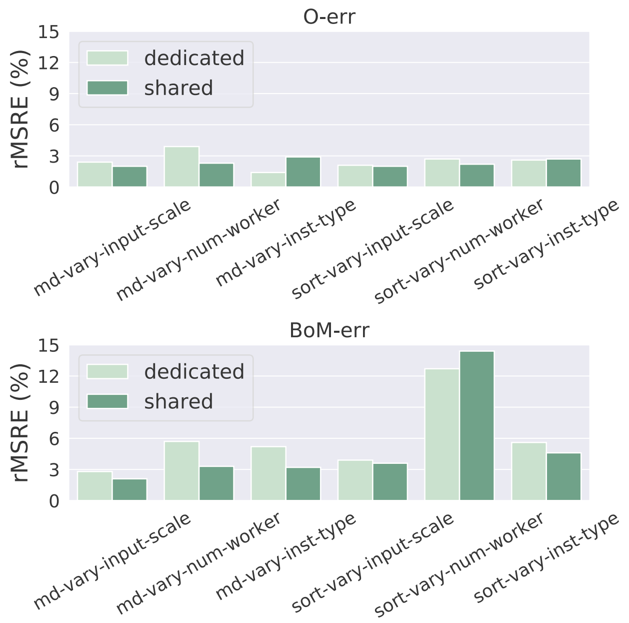


Figure 20: Memcached (md) and Spark sorting (sort) prediction accuracy in the three prediction scenarios with dedicated instances and shared instances.

works, while powerful, can be difficult to train and tune.
(iv) For cases where neural network performed the best, there was often at least one other simpler model performed as well.

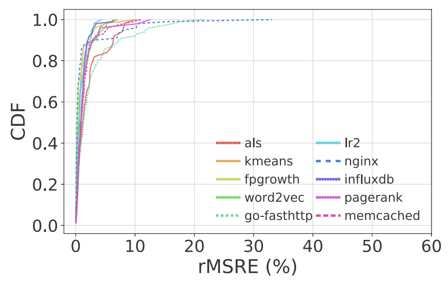


Figure 21: CDF of O-err in the BBC leave-one-out test where input data content is not identical, for predictions across varying input data scale

Prediction Type	**Linear Regression**	**kNN**	**Random Forest**	**SVM**	**SVM kernelized**	**Neural Network**
Input scale	**1.0**	3.5	3.5	**1.0**	2.4	**1.0**
# workers	2.0	2.0	2.1	1.6	1.4	**1.0**
Inst. type	4.0	1.2	**1.0**	1.4	1.4	1.6

Application	**Linear Regression**	**kNN**	**Random Forest**	**SVM**	**SVM kernelized**	**Neural Network**
sort	4.8	2.9	3.3	2.2	1.8	**1.0**
LR1	1.3	1.6	2.0	1.2	1.1	**1.0**
TFS	4.3	2.4	2.2	1.3	**1.0**	1.1
pagerank	2.5	1.7	1.8	1.8	3.4	**1.0**
nginx	6.6	2.6	1.1	1.1	**1.0**	1.7
influxdb	1.2	2.9	2.9	**1.0**	1.2	2.6
memcached	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	1.4
go-fasthttp	1.5	1.1	1.2	1.5	**1.0**	1.1
kmeans	1.3	1.4	**1.0**	1.3	1.6	1.2
LR2	2.3	2.6	2.2	2.4	**1.0**	1.9
word2vec	1.2	5.0	5.1	**1.0**	4.1	2.1
fpgrowth	6.0	2.8	2.7	**1.0**	2.0	1.1
ALS	1.7	1.2	1.1	1.3	2.5	**1.0**

Table 6: Comparison across ML models. The top table reports the error rates for each type of prediction scenario and ML model, averaged across all applications, and normalized by the lowest average error for that scenario. The bottom table similarly reports normalized error rates for each application, averaged across different prediction scenarios.

Scaling Distributed Machine Learning with In-Network Aggregation

Amedeo Sapio* *KAUST*
Marco Canini* *KAUST*
Chen-Yu Ho *KAUST*
Jacob Nelson *Microsoft*
Panos Kalnis *KAUST*
Changhoon Kim *Barefoot Networks*
Arvind Krishnamurthy *University of Washington*
Masoud Moshref *Barefoot Networks*
Dan R. K. Ports *Microsoft*
Peter Richtárik *KAUST*

Abstract

Training machine learning models in parallel is an increasingly important workload. We accelerate distributed parallel training by designing a communication primitive that uses a programmable switch dataplane to execute a key step of the training process. Our approach, SwitchML, reduces the volume of exchanged data by aggregating the model updates from multiple workers in the network. We co-design the switch processing with the end-host protocols and ML frameworks to provide an efficient solution that speeds up training by up to 5.5× for a number of real-world benchmark models.

1 Introduction

Today's machine learning (ML) solutions' remarkable success derives from the ability to build increasingly sophisticated models on increasingly large data sets. To cope with the resulting increase in training time, ML practitioners use distributed training [1, 22]. Large-scale clusters use hundreds of nodes, each equipped with multiple GPUs or other hardware accelerators (e.g., TPUs [48]), to run training jobs on tens of workers that take many hours or days.

Distributed training is increasingly a *network-bound* workload. To be clear, it remains computationally intensive. But the last seven years have brought a 62× improvement in compute performance [64, 78], thanks to GPUs [74] and other hardware accelerators [11, 34, 35, 48]). Cloud network deployments have found this pace hard to match, skewing the ratio of computation to communication towards the latter. Since parallelization techniques like mini-batch stochastic gradient descent (SGD) training [37, 43] alternate computation with synchronous model updates among workers, network performance now has a substantial impact on training time.

Can a new type of accelerator *in the network* alleviate the network bottleneck? We demonstrate that an *in-network aggregation* primitive can accelerate distributed ML workloads, and can be implemented using programmable switch hardware [5, 10]. Aggregation reduces the amount of data transmitted during synchronization phases, which increases throughput, diminishes latency, and speeds up training time.

Building an in-network aggregation primitive using programmable switches presents many challenges. First, the per-packet processing capabilities are limited, and so is on-chip memory. We must limit our resource usage so that the switch can perform its primary function of conveying packets. Second, the computing units inside a programmable switch operate on integer values, whereas ML frameworks and models operate on floating-point values. Finally, the in-network aggregation primitive is an all-to-all primitive, unlike traditional unicast or multicast communication patterns. As a result, in-network aggregation requires mechanisms for synchronizing workers and detecting and recovering from packet loss.

We address these challenges in SwitchML, showing that it is indeed possible for a programmable network device to perform in-network aggregation at line rate. SwitchML is a co-design of in-switch processing with an end-host transport layer and ML frameworks. It leverages the following insights. First, aggregation involves a simple arithmetic operation, making it amenable to parallelization and pipelined execution on programmable network devices. We decompose the parameter updates into appropriately-sized chunks that can be individually processed by the switch pipeline. Second, aggregation for SGD can be applied separately on different portions of the input data, disregarding order, without affecting the correctness of the final result. We tolerate packet loss through the use of a light-weight switch scoreboard mechanism and a retransmission mechanism driven solely by end hosts, which together ensure that workers operate in lock-step without any decrease in switch aggregation throughput. Third, ML training is robust to modest approximations in its compute operations. We address the lack of floating-point support in switch dataplanes by having the workers scale and convert floating-point values to fixed-point using an adaptive scaling factor with negligible approximation loss.

*Equal contribution. Amedeo Sapio is affiliated with Barefoot Networks, but was at KAUST during much of this work.

SwitchML integrates with distributed ML frameworks, such as PyTorch and TensorFlow, to accelerate their communication, and enable efficient training of deep neural networks (DNNs). Our initial prototype targets a *rack-scale architecture*, where a single switch centrally aggregates parameter updates from serviced workers. Though the single switch limits scalability, we note that commercially-available programmable switches can service up to 64 nodes at 100 Gbps or 256 at 25 Gbps. As each worker is typically equipped with multiple GPUs, this scale is sufficiently large to push the statistical limits of SGD [32, 43, 50, 98].

We show that SwitchML's in-network aggregation yields end-to-end improvements in training performance of up to 5.5× for popular DNN models. Focusing on a communication microbenchmark, compared to the best-in-class collective library NCCL [77], SwitchML is up to 2.9× faster than NCCL with RDMA and 9.1× than NCCL with TCP. While the magnitude of the performance improvements is dependent on the neural network architecture and the underlying physical network speed, it is greater for models with smaller compute-to-communication ratios – good news for future, faster DNN training accelerators.

Our approach is not tied to any particular ML framework; we have integrated SwitchML with Horovod [89] and NCCL [77], which support several popular toolkits like TensorFlow and PyTorch. SwitchML is openly available at https://github.com/p4lang/p4app-switchML.

2 Network bottlenecks in ML training

In the distributed setting, ML training yields a high-performance networking problem, which we highlight below after reviewing the traditional ML training process.

2.1 Training and all to all communication

Supervised ML problems, including logistic regression, support vector machines and deep learning, are typically solved by iterative algorithms such as stochastic gradient descent (SGD) or one of its many variants (e.g., using momentum, mini-batching, importance sampling, preconditioning, variance reduction) [72, 73, 83, 90]. A common approach to scaling to large models and datasets is data-parallelism, where the input data is partitioned across workers.[1] Training in a data-parallel, synchronized fashion on n workers can be seen as learning a model $x \in \mathbb{R}^d$ over input/training data D by performing iterations of the form $x^{t+1} = x^t + \sum_{i=1}^{n} \Delta(x^t, D_i^t)$, where x^t is a vector of model parameters[2] at iteration t, $\Delta(\cdot,\cdot)$ is the model update function[3] and D_i^t is the data subset used at worker i during that iteration.

The key to data parallelism is that each worker i, in parallel, locally computes the update $\Delta(x^t, D_i^t)$ to the model parameters based on the current model x^t and a mini-batch, i.e., a subset of the local data D_i^t. Typically, a model update contributed by worker i is a multiple of the stochastic gradient of the loss function with respect to the current model parameters x^t computed across a mini-batch of training data, D_i^t. Subsequently, workers communicate their updates, which are aggregated ($\sum$) and added to x^t to form the model parameters of the next iteration. Importantly, each iteration acts only on a mini-batch of the training data. It requires many iterations to progress through the entire dataset, which constitutes a training epoch. A typical training job requires multiple epochs, reprocessing the full training data set, until the model achieves acceptable error on a validation set.

From a networking perspective, the challenge is that data-parallel SGD requires computing the sum of model updates across all workers after every iteration. Each model update has as many parameters as the model itself, so they are often in 100s-of-MB or GB range. And their size is growing exponentially: today's largest models exceed 32 GB [84]. These aggregations need to be performed frequently, as increasing the mini-batch size hurts convergence [66]. Today's ML toolkits implement this communication phase in one of two ways:

The parameter server (PS) approach. In this approach, workers compute model updates and send them to *parameter servers* [45, 56, 64]. These servers, usually dedicated machines, aggregate updates to compute and distribute the new model parameters. To prevent the PS from becoming a bottleneck, the model is sharded over multiple PS nodes.

The all-reduce approach. An alternate approach uses the workers to run an all-reduce algorithm – a collective communication technique common in high-performance computing – to combine model updates. The workers communicate over an overlay network. A *ring* topology [6], where each worker communicates to the next neighboring worker on the ring, is common because it is bandwidth-optimal (though its latency grows with the number of workers) [79]. *Halving and doubling* uses a binary tree topology [93] instead.

2.2 The network bottleneck

Fundamentally, training alternates compute-intensive phases with communication-intensive model update synchronization. Workers produce intense bursts of traffic to communicate their model updates, whether it is done through a parameter server or all-reduce, and training stalls until it is complete.

Recent studies have shown that performance bottleneck in distributed training is increasingly shifting from compute to communication [64]. This shift comes from two sources. The first is a result of advances in GPUs and other compute accelerators. For example, the recently released NVIDIA A100 offers 10× and 20× performance improvements for floating-point

[1]In this paper, we do not consider model-parallel training [28, 82], although that approach also requires efficient networking. Further, we focus exclusively on widely-used distributed synchronous SGD [1, 37].

[2]In applications, x is typically a 1, 2, or 3 dimensional tensor. To simplify notation, we assume its entries are vectorized into one d dimensional vector.

[3]We abstract learning rate (step size) and model averaging inside Δ.

and mixed-precision calculations, respectively [74] compared to its predecessor, the V100 – released just 2.5 years previously. This pace far exceeds advances in network bandwidth: a 10× improvement in Ethernet speeds (from 10 Gbps to 100 Gbps) required 8 years to standardize.

Second, the ratio of communication to computation in the workload itself has shifted. The current trend towards ever-larger DNNs generally exacerbates this issue. However, this effect is highly application-dependent. In popular ML toolkits, communication and computation phases can partially overlap. Since back-prop proceeds incrementally, communication can start as soon as the earliest partial results of back-prop are available. The effectiveness of this technique depends on the structure of the DNN. For DNNs with large initial layers, its effectiveness is marginal, because there is little to no opportunity to overlap communication with computation.

When is the network a bottleneck? To answer this quantitatively, we profile the training of 8 common DNNs on a cluster with 8 workers using NVIDIA P100 GPUs. To precisely factor out the contribution of communication to the processing time of a mini-batch, we emulate communication time at 10 Gbps or 100 Gbps Ethernet assuming transmission at line rate. We record the network-level events, which allows us to report the fraction of time spent in communication as well as how much can overlap with computation (Table 1). At 10 Gbps, all but three workloads spend more than 50% of their time in communication, usually with little computation-phase overlap. These workloads benefit greatly from 100 Gbps networking, but even so communication remains a significant share (at least 17%) of batch processing time for half of the workloads.

What happens when GPUs become faster? Our profile uses P100 GPUs, now two generations old. Faster GPUs would reduce the computation time, increasing the relative communication fraction. Our measurement of non-overlappable communication time allows us to determine the scaling factor α applied to GPU computation time at which point the network is saturated. There is still some speedup beyond an $\alpha\times$ faster GPU, but it is limited to the initial phase, before communication begins. Note $\alpha < 4$ for half the workloads (Table 1), suggesting that network performance will be a serious issue when using the latest GPUs with a 100 Gbps network.

3 In-network aggregation

We propose an alternative approach to model update exchange for ML workloads: *in-network aggregation*. In this approach, workers send their model updates over the network, where an aggregation primitive in the network sums the updates and distributes only the resulting value. Variations on this primitive have been proposed, over the years, for specialized supercomputer networks [2, 26] and InfiniBand [33]. We demonstrate

Model	Size	Batch	10 Gbps		100 Gbps		
	[MB]	size	Batch [ms]	Comm [%]	Batch [ms]	Comm [%]	α
DeepLight	2319	2^{13}	2101 ± 1.4	97% (2%)	258 ± 0.4	79% (20%)	1.0
LSTM	1627	64	1534 ± 8.3	94% (10%)	312 ± 6.8	46% (56%)	1.5
BERT	1274	4	1677 ± 7.1	67% (3%)	668 ± 3.1	17% (35%)	3.5
VGG19	548	64	661 ± 1.9	73% (67%)	499 ± 1.1	10% (99%)	6.7
UGATIT	511	2	1612 ± 2.5	28% (84%)	1212 ± 3.5	4% (99%)	17.6
NCF	121	2^{17}	149 ± 0.6	72% (4%)	46 ± 0.1	23% (27%)	1.2
SSD	98	16	293 ± 0.6	26% (99%)	293 ± 1.6	3% (99%)	15.2
ResNet-50	87	64	299 ± 10.9	29% (67%)	270 ± 1.2	3% (94%)	19.8

Table 1: Profile of benchmark DNNs. "Batch [ms]" reports the average batch processing time and its standard deviation. "Comm" reports the proportion of communication activity as % of batch time. The figure in parentheses is the percentage of *that* time that overlaps with computation. For example, DeepLight at 10 Gbps spends 97% of its batch time in communication; only 2% of this 97% communication overlaps with computation. The table lists a scaling factor for an hypothetical $\alpha\times$ faster GPU that implies communication is contiguous and saturates the 100 Gbps bandwidth once communication begins.

that it is possible to realize in-network aggregation in an Ethernet network and benefit ML applications.

In-network aggregation offers a fundamental advantage over both all-reduce and PS. It achieves the minimum possible latency and the minimum communication cost, quantified in data volume each worker sends and receives: $2|U|$ bytes, where $|U|$ is the total number of bytes to be aggregated. This is a significant improvement over the equivalent cost for bandwidth-optimal all-reduce, which is $4|U|\frac{n-1}{n}$ [79]. The PS approach can match this communication cost of $2|U|$ bytes, at the expense of more resource cost; in the limit, it doubles the number of required machines and network bandwidth.[4] Regardless of resource costs, in-network aggregation avoids end-host processing required to perform aggregation and, therefore, provides "sub-RTT" latency [46], which the contrasted approaches cannot achieve.

Illustrating the advantages of in-network aggregation. To characterize the extent to which communication is a bottleneck for training performance, we use our profile of eight DNN models from §2.2. We evaluate the impact of communication performance using a trace of network-level events recorded during training. This trace captures real compute times and memory access latency, including the latency for barrier events that precede each synchronization, but allows us to emulate different network speeds and computation patterns. In particular, our trace records the detailed timing of individual all-reduce invocations, so it faithfully accounts for potential overlap between communication and computation.[5]

We compare the performance of in-network aggregation (INA) with the current best practice, ring all-reduce (RAR). Table 2 summarizes the batch processing speedup over the

[4]If the PS nodes are co-located with the worker nodes, then the effective bandwidth per node is halved, doubling latency.

[5]The ML toolkit adopts an optimization known as *tensor fusion* or *bucketing* that coalesces multiple all-reduce invocations to amortize setup overhead. Our traces reflect the effect of this optimization.

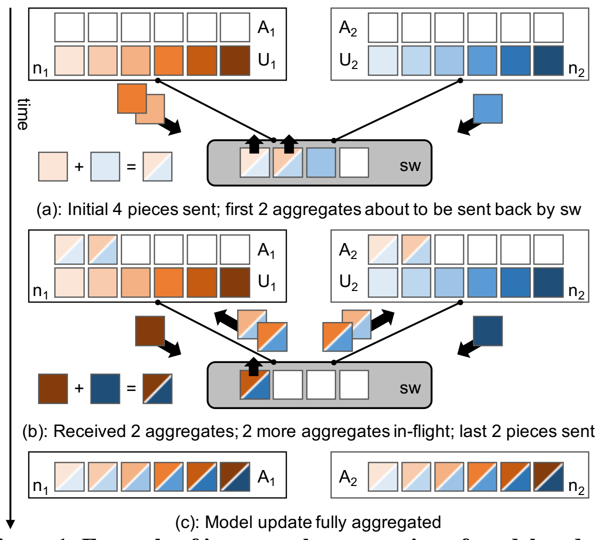

time
A₁
U₁
n₁
A₂
U₂
n₂
sw
+
=
(a): Initial 4 pieces sent; first 2 aggregates about to be sent back by sw
(b): Received 2 aggregates; 2 more aggregates in-flight; last 2 pieces sent
(c): Model update fully aggregated

Figure 1: Example of in-network aggregation of model updates. U_i is the model update computed by worker i. Workers stream pieces of model updates in a coordinated fashion. In the example, each workers can have at most 4 outstanding packets at any time to match the slots in the switch. The switch aggregates updates and multicasts back the values, which are collected into the aggregated model update A_i, then used to form the model parameters of the next iteration.

ring all-reduce performance. INA is consistently superior to RAR. For communication-bound models (the four models in the 100 Gbps case), INA is up to 80% and up to 67% faster at 10 and 100 Gbps, respectively. Note that this analysis reflects a *theoretically optimal* implementation of RAR. The measured speedups (§6) of our real INA implementation are higher, because real RAR implementations do not achieve optimal performance; it is difficult to fully exploit all available bandwidth and avoid system overheads.

We also note that our profiling environment uses NVIDIA P100 devices. These are currently two-generation old GPU accelerators. We investigate with real benchmarks in §6 the impact of faster GPUs, which increases the relative impact of communication overheads.

Alternative: gradient compression. Another way to reduce communication costs is to reduce the data volume of model updates using lossy compression. Proposed approaches include reducing the bit-width of gradient elements (quantization) or transmitting only a subset of elements (sparsification). These approaches come with tradeoffs: too much compression loss can impact the resulting model accuracy.

We adopt the results of a recent survey of gradient compression methods [96] to emulate the behavior of Top-*k* [3] and QSGD [4] as two representative sparsification and quantization compressors, respectively. We use data from that study to identify the compression overhead and data reduction achieved. Our synthetic communication time, then, includes both the computational cost of compression and the communication cost of the all-gather operation used to exchange model

Model	INA	QSGD		Top-*k*	
		64	256	1%	10%
			10 Gbps		
DeepLight	1.80	1.27	0.97	9.24 (-1.1%)	1.05 (-0.9%)
LSTM	1.77	1.27	0.97	7.49	1.05
NCF	1.54	1.22	0.96	4.07	1.05 (-2.2%)
BERT	1.54	1.20	0.98	3.45 (†)	1.04 (†)
VGG19	1.60	1.22	0.97	2.13 (-10.4%)	1.04 (-3.3%)
UGATIT	1.22	1.12	0.99	1.58	1.02
ResNet-50	1.05	1.07	0.95	1.15 (-1.7%)	1.02 (+0.2%)
SSD	1.01	1.00	1.00	1.01 (-2.4%)	1.00 (-0.6%)
			100 Gbps		
DeepLight	↗1.67	0.93	0.78	2.96 (-1.1%)	0.47 (-0.9%)
LSTM	↗1.20	0.98	0.84	1.37	0.54
NCF	1.22	1.00	0.85	1.22	0.65 (-2.2%)
BERT	↗1.14	0.98	0.92	1.27 (†)	0.74 (†)

† The BERT task is fine-tuning from a pre-trained model, for which compression does not have a noticeable impact. The impact during pretraining is analyzed in Appendix E.

Table 2: Analysis of batch processing speedup relative to ring all-reduce based on synthetic communication. For Top-*k* compression, impact on model quality is shown in parentheses. Accuracy penalties greater than 1% are shaded in gray; red indicates failure to converge. At 100 Gbps, only the models that are network bottlenecked are shown. ↗ indicates 100 Gbps cases where SwitchML achieves a higher batch processing speedup due to practical system overheads.

updates (following their implementation [96]).

We observe (Table 2) that, although gradient compression decreases data volume, it is not necessarily superior to INA. In general, the computational cost of compression and decompression is non-negligible [58,96]; in some cases, it outweighs the communication-reduction benefits. In particular, INA outperforms QSGD on all workloads for both the 64 and 256 levels (6 and 8 bits). Similarly, Top-*k* underperforms INA at the 10% compression level, and even *reduces* performance relative to RAR in the 100 Gbps setting. These observations agree with recent work [58, 96]. In particular, Li et al. [58] proposed additional hardware offloading, using an FPGA at every worker, to mitigate compression costs. As this requires additional hardware, our analysis does not consider it.

Gradient compression *does* outperform INA when it can achieve high compression ratios, as with Top-*k* at 1%. However, in many cases, this level of compression either requires more training iterations to converge, or hurts the accuracy of the resulting model [96]. For example, the NCF model achieves 95.8% hit rate without compression after 20 epochs of training, while with Top-*k* compression at 10% it achieves 93.6%. It fails to converge at 1% compression. We report convergence comparisons for various models in Appendix D.

4 Design

Our system, SwitchML, implements the aggregation primitive in a programmable dataplane switch. Such switches are now commercially available, with only a small cost premium compared to fixed-function switches [5]. In-network aggregation is conceptually straightforward, but implementing it inside a programmable switch, however, is challenging. Although programmable switches allow placing computation into the network path, their limited computation and storage capabilities impose constraints on implementing gradient aggregation. The system must also tolerate packet loss, which, although uncommon in the rack-scale cluster environment, is nevertheless possible for long-running DNN training jobs. SwitchML addresses these challenges by appropriately dividing the functionality between the hosts and the switches, resulting in an efficient and reliable streaming aggregation protocol.

4.1 Challenges

Limited computation. Mathematically, gradient aggregation is the average over a set of floating-point vectors. While a seemingly simple operation, it exceeds the capabilities of today's programmable switches. As they must maintain line rate processing, the number of operations they can perform on each packet is limited. Further, the operations themselves can only be simple integer arithmetic/logic operations; neither floating-point nor integer division operations are possible.

Limited storage. Model updates are large. In each iteration, each worker may supply hundreds of megabytes of gradient values. This volume far exceeds the on-switch storage capacity, which is limited to a few tens of MB and must be shared with forwarding tables and other core switch functions. This limitation is unlikely to change in the future [10], given that speed considerations require dataplane-accessible storage to be implemented using on-die SRAM.

Packet loss. SwitchML must be resilient to packet loss, without impact on efficiency or correctness (e.g., discarding part of an update or applying it twice because of retransmission).

4.2 SwitchML overview

SwitchML aims to alleviate communication bottlenecks for distributed ML training applications using in-network aggregation, in a practical cluster setting.[6] SwitchML uses the following techniques to reduce communication costs while meeting the above challenges.

Combined switch-host architecture. SwitchML carefully partitions computation between end-hosts and switches to circumvent the restrictions of the limited computational power at switches. The switch performs integer aggregation, while end-hosts are responsible for managing reliability and performing more complex computations.

[6]For simplicity, we assume dedicated bandwidth for the training jobs. We also assume that worker, link or switch failures are handled by the ML framework, as it is common in practice [1,56].

Algorithm 1 Switch logic.

```
1:  Initialize State:
2:      n = number of workers
3:      pool[s], count[s] := {0}
4:  upon receive p(idx, off, vector)
5:      pool[p.idx] ← pool[p.idx] + p.vector        {+ is the vector addition}
6:      count[p.idx]++
7:      if count[p.idx] = n then
8:          p.vector ← pool[p.idx]
9:          pool[p.idx] ← 0; count[p.idx] ← 0
10:         multicast p
11:     else
12:         drop p
```

Pool-based streaming aggregation. A complete model update far exceeds the storage capacity of a switch, so it cannot aggregate entire vectors at once. SwitchML instead *streams* aggregation through the switch: it processes the aggregation function on a limited number of vector elements at once. The abstraction that makes this possible is a pool of integer aggregators. In SwitchML, end hosts handle the management of aggregators in a pool – determining when they can be used, reused, or need more complex failure handling – leaving the switch dataplane with a simple design.

Fault tolerant protocols. We develop lightweight schemes to recover from packet loss with minimal overheads and adopt traditional mechanisms to solve worker or network failures.

Quantized integer-based aggregation. Floating-point operations exceed the computational power of today's switches. We instead convert floating-point values to 32-bit integers using a block floating-point-like approach [25], which is done efficiently at end hosts without impacting training accuracy.

We now describe each of these components in turn. To ease the presentation, we describe a version of the system in which packet losses do not occur. We remove this restriction later.

4.3 Switch-side aggregation protocol

We begin by describing the core network primitive provided by SwitchML: in-switch integer aggregation. A SwitchML switch provides a pool of s integer aggregators, addressable by index. Each slot in the pool aggregates a vector of k integers, which are delivered all at the same time in one update packet. The aggregation function is the addition operator, which is commutative and associative – meaning that the result does not depend on the order of packet arrivals. Note that addition is a simpler form of aggregation than ultimately desired: model updates need to be *averaged*. As with the all-reduce approach, we leave the final division step to the end hosts, as the switch cannot efficiently perform this.

Algorithm 1 illustrates the behavior of the aggregation primitive. A packet p carries a pool index, identifying the particular aggregator to be used, and contains a vector of k integers to be aggregated. Upon receiving a packet, the switch aggregates the packet's vector ($p.vector$) into the slot addressed by the packet's pool index ($p.idx$). Once the slot has aggregated vectors from each worker,[7] the switch outputs the result – by rewriting the packet's vector with the aggregated value from that particular slot, and sending a copy of the packet to each worker. It then resets the slot's aggregated value and counter, releasing it immediately for reuse.

The pool-based design is optimized for the common scenario where model updates are larger than the memory capacity of a switch. It addresses two major limitations of programmable switch architectures. First, because switch memory is limited, it precludes the need to store an entire model update on a switch at once; instead, it aggregates pieces of the model in a streaming fashion. Second, it allows processing to be done at the packet level by performing the aggregation in small pieces, at most k integers at a time. This is a more significant constraint than it may appear; to maintain a very high forwarding rate, today's programmable switches parse only up to a certain amount of bytes in each packet and allow computation over the parsed portion. Thus, the model-update vector and all other packet headers must fit within this limited budget, which is today on the order of a few hundred bytes; ASIC design constraints make it unlikely that this will increase dramatically in the future [10, 16, 92]. In our deployment, k is 64 or 256.

4.4 Worker-side aggregation protocol

The switch-side logic above does not impose any constraints on which aggregator in the pool to use and when. Workers must carefully control which vectors they send to which pool index and, since pool size s is limited, how they reuse slots.

There are two considerations in managing the pool of aggregators appropriately. For correctness, every worker must use the same slot for the same piece of the model update, and no slot can be simultaneously used for two different pieces. For performance, every worker must work on the same slot at roughly the same time to avoid long synchronization delays. To address these issues, we design a custom aggregation protocol running at the end hosts of ML workers.

For now, let us consider the non-failure case, where there is no packet loss. The aggregation procedure, illustrated in Algorithm 2, starts once every worker is ready to exchange its model update. Without loss of generality, we suppose that the model update's size is a multiple of k and is larger than $k \cdot s$, where k is the size of the vector aggregated in each slot and s denotes the pool size. Each worker initially sends s packets containing the first s pieces of the model update – each piece being a contiguous array of k values from offset *off* in that worker's model update U. Each of these initial packets is assigned sequentially to one of the s aggregation slots.

After the initial batch of packets is sent, each worker awaits the aggregated results from the switch. Each packet received indicates that the switch has completed the aggregation of a particular slot. The worker consumes the result carried in the packet, copying that packet's vector into the aggregated model update A at the offset carried in the packet ($p.off$). The worker then sends a new packet with the *next* piece of update to be aggregated. This reuses the same pool slot as the one just received, but contains a new set of k parameters, determined by advancing the previous offset by $k \cdot s$.

Algorithm 2 Worker logic.

```
1: for i in 0 : s do
2:     p.idx ← i
3:     p.off ← k · i
4:     p.vector ← U[p.off : p.off + k]
5:     send p
6: repeat
7:     receive p(idx, off, vector)
8:     A[p.off : p.off+k] ← p.vector
9:     p.off ← p.off + k · s
10:    if p.off < size(U) then
11:        p.vector ← U[p.off : p.off + k]
12:        send p
13: until A is incomplete
```

A key advantage of this scheme is that it does not require any explicit coordination among workers and yet achieves agreement among them on which slots to use for which parameters. The coordination is implicit because the mapping between model updates, slots, and packets is deterministic. Also, since each packet carries the pool index and offset, the scheme is not influenced by reorderings. A simple checksum can be used to detect corruption and discard corrupted packets.

This communication scheme is self-clocked after the initial s packets. This is because a slot cannot be reused until all workers have sent their contribution for the parameter update for the slot. When a slot is completed, the packets from the switch to the workers serve as flow-control acknowledgments that the switch is ready to reuse the slot, and the workers are free to send another packet. Workers are synchronized based on the rate at which the system aggregates model updates. The pool size s determines the number of concurrent in-flight aggregations; as we elaborate in Appendix §C, the system achieves peak bandwidth utilization when $k \cdot s$ (more precisely, $b \cdot s$ where b is the packet size – 1100 bytes in our setting) matches the bandwidth-delay product of the inter-server links.

4.5 Dealing with packet loss

Thus far, we have assumed packets are never lost. Of course, packet loss can happen due to either corruption or network congestion. With the previous algorithm, even a single packet loss would halt the system. A packet loss on the "upward" path from workers to the switch prevents the switch from completing the corresponding parameter aggregations. The loss of one of the result packets that are multicast on the "downward" paths not only prevents a worker from learning the result but also prevents it from ever completing A.

We tolerate packet loss by retransmitting lost packets. In

[7]For simplicity, we show a simple counter to detect this condition. Later, we use a bitmap to track which workers have sent updates.

Algorithm 3 Switch logic with packet loss recovery.

```
1:  Initialize State:
2:      n = number of workers
3:      pool[2, s], count[2, s], seen[2, s, n] := {0}
4:  upon receive p(wid, ver, idx, off, vector)
5:      if seen[p.ver, p.idx, p.wid] = 0 then
6:          seen[p.ver, p.idx, p.wid] ← 1
7:          seen[(p.ver+1)%2, p.idx, p.wid] ← 0
8:          count[p.ver, p.idx] ← (count[p.ver, p.idx]+1)%n
9:          if count[p.ver, p.idx] = 1 then
10:             pool[p.ver, p.idx] ← p.vector
11:         else
12:             pool[p.ver, p.idx] ← pool[p.ver, p.idx] + p.vector
13:         if count[p.ver, p.idx] = 0 then
14:             p.vector ← pool[p.ver, p.idx]
15:             multicast p
16:         else
17:             drop p
18:     else
19:         if count[p.ver, p.idx] = 0 then
20:             p.vector ← pool[p.ver, p.idx]
21:             forward p to p.wid
22:         else
23:             drop p
```

Algorithm 4 Worker logic with packet loss recovery.

```
1:  for i in 0 : s do
2:      p.wid ← Worker ID
3:      p.ver ← 0
4:      p.idx ← i
5:      p.off ← k · i
6:      p.vector ← U[p.off : p.off + k]
7:      send p
8:      start_timer(p)
9:  repeat
10:     receive p(wid, ver, idx, off, vector)
11:     cancel_timer(p)
12:     A[p.off : p.off+k] ← p.vector
13:     p.off ← p.off + k · s
14:     if p.off < size(U) then
15:         p.ver ← (p.ver+1)%2
16:         p.vector ← U[p.off : p.off + k]
17:         send p
18:         start_timer(p)
19: until A is incomplete

20: upon timeout p /* Timeout Handler */
21:     send p
22:     start_timer(p)
```

order to keep switch dataplane complexity low, packet loss detection is done by the workers if they do not receive a response packet from the switch in a timely manner. However, naïve retransmission creates its own problems. If a worker retransmits a packet that was actually delivered to the switch, it can cause a model update to be applied twice to the aggregator. On the other hand, if a worker retransmits a packet for a slot that was actually already fully aggregated (e.g., because the response was lost), the model update can be applied to the wrong data because the slot could have already been reused by other workers who received the response correctly. Thus, the challenges are (1) to be able to differentiate packets that are lost on the upward paths versus the downward ones; and (2) to be able to retransmit an aggregated response that is lost on the way back to a worker.

We modify the algorithms to address these issues by keeping two additional pieces of switch state. First, we explicitly maintain information as to which workers have already contributed updates to a given slot. This makes it possible to ignore duplicate transmissions. Second, we maintain a *shadow copy* of the *previous* result for each slot. That is, we have two copies or versions of each slot, organized in two pools; workers alternate between these two copies to aggregate successive chunks that are assigned to the same slot. The shadow copy allows the switch to retransmit a dropped result packet for a slot even when the switch has started reusing the slot for the next chunk.

The key insight behind this approach's correctness is that, even in the presence of packet losses, our self-clocking strategy ensures that no worker node can ever lag *more than one chunk* behind any of the others for a particular slot. This invariant is because the switch will not release a slot to be reused, by sending a response, until it has received an update packet from *every* worker for that slot. Furthermore, a worker will not send the next chunk for a slot until it has received the response packet for the slot's previous chunk, preventing the system from moving ahead further. As a result, it is sufficient to keep only one shadow copy.

Besides obviating the need for more than one shadow copy, this has a secondary benefit: the switch does not need to track full phase numbers (or offsets); a single bit is enough to distinguish the two active phases for any slot.

In keeping with our principle of leaving protocol complexity to end hosts, the shadow copies are kept in the switch but managed entirely by the workers. The switch simply exposes the two pools to the workers, and the packets specify which slot acts as the active copy and which as the shadow copy by indicating a single-bit pool version (*ver*) field in each update packet. The pool version starts at 0 and alternates each time a slot with the same *idx* is reused.

Algorithms 3 and 4 show the details of how this is done. An example illustration is in Appendix A. In the common case, when no losses occur, the switch receives updates for slot *idx*, pool *ver* from all workers. When workers receive the response packet from the switch, they change the pool by flipping the *ver* field – making the old copy the shadow copy – and send the next phase updates to the other pool.

A timeout detects packet loss at each worker. When this occurs, the worker does not know whether the switch received its previous packet or not. Regardless, it retransmits its earlier update with the same slot *idx* and *ver* as before. This slot is guaranteed to contain the state for the same aggregation in the switch. The *seen* bitmask indicates whether the update has already been applied to the slot. If the aggregation is already complete for a slot, and the switch yet receives an update packet for the slot, the switch recognizes the packet as a retransmitted packet and replies with a unicast packet containing the result. The result in one slot is overwritten for reuse only when there is the certainty that all the workers have received the slot's aggregated result. Slot reuse happens when all the workers have sent their updates to the same slot of the other pool, signaling that they have all moved forward. Note this scheme works because the completion of aggregation for a slot *idx* in one pool *safely and unambiguously* confirms that the previous aggregation result in the shadow copy of slot *idx* has indeed been received by every worker.

This mechanism's main cost is switch memory usage: keeping a shadow copy doubles the memory requirement, and tracking the *seen* bitmask adds additional cost. This may appear problematic, as on-switch memory is a scarce resource. In practice, however, the total number of slots needed – tuned based on the network bandwidth-delay product (Appendix C) – is much smaller than the switch's memory capacity.

4.6 Dealing with floating-point numbers

DNN training commonly uses floating-point numbers, but current programmable switches do not natively support them. We explored two approaches to bridging this gap.

Floating-point numbers are already an approximation. SGD and similar algorithms are defined over real numbers. Floating-point numbers approximate real numbers by trading off range, precision, and computational overhead to provide a numerical representation that can be broadly applied to applications with widely different properties. However, many other approximations are possible. An approximation designed for a specific application can obtain acceptable accuracy with lower overhead than standard floating-point offers.

In recent years, the community has explored many specialized numerical representations for DNNs. These representations exploit the properties of the DNN application domain to reduce the cost of communication and computation. For instance, NVIDIA Volta and Ampere GPUs [17, 74] include mixed-precision (16-/32-bit) TPUs that can train with accuracy matching full-precision approaches. Other work has focused on gradient exchange for SGD, using fixed-point quantization, dithering, or sparsification to reduce both the number of bits and the gradient elements transmitted [7, 8, 60, 69, 88, 95, 99]. Further, others have explored block floating-point representations [25, 53], where a single exponent is shared by multiple tensor elements, reducing the amount of computation required to perform tensor operations. This innovation will continue (as work [40, 70] that builds upon our architecture demonstrates); our goal is not to propose new representations but to demonstrate that techniques like those in the literature are practical with programmable switches.

We use a numeric representation, inspired by block floating-point, that combines 32-bit fixed-point addition in the switch with adaptive scaling on the workers. This representation is used only when aggregating gradients; all other data (weights, activations) remain in 32-bit floating-point representation.

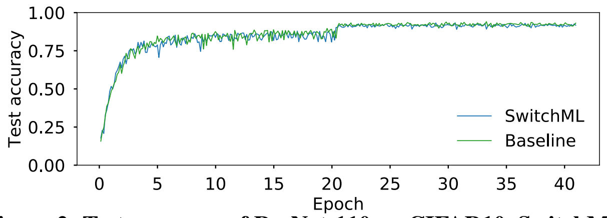


Figure 2: Test accuracy of ResNet-110 on CIFAR10. SwitchML achieves similar accuracy to the baseline.

To implement our representation, we scale gradient values using a per-packet scaling factor f, which is automatically determined for each use of an aggregator slot in the switch. The scaling factor is set so that the maximum aggregated floating point value within a block of k gradients is still representable as a 32-bit fixed point value. Namely, let h be the largest absolute value of a block of gradients; f is set to $(2^{31}-1)/(n \cdot 2^m)$, where m is the exponent of h rounded up to a power of 2 and n is the number of workers. Appendix E formally analyzes the precision of this representation.

To realize this quantization of floating-point values, workers need to agree on a global value of m prior to sending the corresponding block of gradients. We devise a simple look-ahead strategy: when workers send the j-th block to slot i, they include their local block $j+1$'s maximum gradient (rounded up to a power of 2). The switch identifies the global maximum m and piggy-backs that value when sending the aggregated gradients of the j-th block.

We verify experimentally that this communication quantization allows training to similar accuracy in a similar number of iterations as an unquantized network. We illustrate the convergence behavior by training a ResNet-110 model on CIFAR10 dataset for 64,000 steps (about 41 epochs) using 8 workers. Figure 2 shows the test accuracy over time. The accuracy obtained by SwitchML (about 91-93% in the last 5 points) is similar to that obtained by training with TensorFlow on the same worker setup, and it matches prior results [38] with the same hyperparameter settings. The training loss curves (not shown) show the same similarity. In Appendix E, we further give a detailed convergence analysis for the aforementioned representation on models in Table 1.

While the above representation is used in the remainder of the paper, we also explored the implementation of a restricted form of 16-bit floating-point. In this version, the switch converts each 16-bit floating-point value in the incoming packets into a fixed-point value and then performs aggregation. When generating responses, the switch converts fixed-point values back into floating-point values. Due to resource limitations in the switch, we were only able to support half the dynamic range of the 16-bit floating-point format; we expect this to lead to poor convergence during training. Conversely, our 32-bit integer format uses minimal switch resources, provides good dynamic range, and has a minimal overhead on workers. A 16-bit format would provide a bandwidth benefit (§6.3).

5 Implementation

We build SwitchML as a collective library which we integrate in PyTorch's DistributedDataParallel module and in TersorFlow via Horovod [89]. SwitchML is implemented as a worker component written as $\sim$3,100 LoCs in C++ and a switch component realized in P4 [9] with $\sim$3,700 LoCs. The worker is built atop Intel DPDK. We have also built a RDMA-capable implementation, but it is not yet integrated with the training frameworks. Here, we highlight a few salient aspects of our implementation. Appendix B describes more details.

Our P4 program distributes aggregation across multiple stages of the ingress pipeline, and also implements flow control, retransmission, and exponent-calculation logic. It uses the traffic manager subsystem to send multiple copies of result packets. It can process 64 elements per packet using one switch pipeline, and 256-element (1024-byte) packets using all four switch pipelines. On the worker side, we process each packet in a run-to-completion fashion and scale to multiple CPU cores using DPDK and Flow Director. We use up to 8 cores per worker. This scales well because we shard slots and chunks of tensors across cores without any shared state. The ML framework invokes our synchronous API whenever model updates are ready. In practice, model updates consist of a set of tensors that are aggregated independently but sequentially.

Supporting large packets. Good bandwidth efficiency requires processing enough integer elements in each packet to offset the network framing overhead. Our P4 program can parse and aggregate 64 $\times$ 4-byte elements per packet, but can only read 32 elements per packet when aggregation is complete. With framing overheads, a 32-element payload would limit goodput to 63% of line rate. Our P4 program supports larger packets in two additional configurations for better efficiency: a 64-element configuration with 77% goodput, and a 256-element one with 93% goodput.

We support larger packets through *recirculation*: sending packets through the switch pipelines multiple times. Our 64-element design uses a single pipeline. It makes one additional pass through the pipeline *only* when an output packet is broadcast in order to read the results: this separation of reads and writes allows us to write 64 elements in a single pass. The internal recirculation ports provided by the chip provide sufficient bandwidth. To support 256 elements, we recirculate packets through all four switch pipelines. This requires placing switch ports into loopback mode for more recirculation bandwidth, leaving 16 $\times$ 100 Gbps bandwidth available for workers. When a slot is complete, we recirculate again through all the pipelines to read the results. Tofino has sufficient bandwidth to do this recirculation at 1.6 Tbps, and the latency scales deterministically with the number of pipeline passes: we measure an additional 520 ns per pass.

Supporting RDMA. Our host-side framework, even using DPDK and multiple cores, has difficulty achieving 100 Gbps throughput due to packet processing costs. We address this by implementing a subset of RDMA in the switch. This allows workers to offload packet processing: the RDMA NIC breaks large messages into individual packets. Specifically, we use RoCE v2 [42] in Unreliable Connected (UC) mode [67]. This mode, which does not require any of RoCE's link-level flow control mechanisms, supports multi-packet messages and detects packet drops, but does not implement retransmission. SwitchML continues to rely on its existing reliability mechanism. Timeouts and duplicate packets are handled as before, except that a timeout forces a client to retransmit the entire multi-packet message. To balance the benefit of offload with the cost of retransmission, we use small, multi-packet messages (generally 16 packets per message). Although retransmissions are more expensive, the common case is much faster, even though we use a single CPU core.

RDMA Write Immediate messages are used for all communication, allowing data to move directly between the switch and GPUs, with client CPUs handling protocol operations. SwitchML metadata is encoded in RDMA headers. Concurrent messages are sent on separate queue pairs to allow packets to interleave; queue pair IDs and access keys are negotiated with the switch control plane during job setup. The switch sends aggregated results by generating RDMA Write messages to the destination buffer.

6 Evaluation

We analyze the performance benefits of SwitchML by using standard benchmarks on popular models in TensorFlow and PyTorch and by using microbenchmarks to compare it to state-of-the-art collective communications libraries and PS scenarios.

Testbed. We conduct most of our experiments on a testbed of 8 machines, each with 1 NVIDIA P100 16 GB GPU, dual 10-core CPU Intel Xeon E5-2630v4 at 2.20 GHz, 128 GB of RAM, and 3 $\times$ 1 TB disks for storage (as single RAID). To demonstrate scalability with 16 nodes, we further use 8 machines with dual 8-core CPU Intel Xeon Silver 4108 at 1.80 GHz. Moreover, we use a Wedge100BF-65X programmable switch with Barefoot Networks' Tofino chip [5]. Every node is networked at both 10 and 100 Gbps.

Performance metrics. We mostly focus on two performance metrics. We define *tensor aggregation time* (TAT) as the time to aggregate a tensor starting from the time a worker is ready to send it till the time that worker receives the aggregated tensor; lower is better. We also report *aggregated tensor elements* (ATE) per unit of time, for presentation clarity; higher is better. For these metrics, we collect measurements at each worker for aggregating 100 tensors of the same size, after 10 warmups. We measure *training throughput* defined in terms of the numbers of training samples processed per second. We measure throughput for 100 iterations that follow 100 warmups. A variant of training throughput is the batch processing throughput,

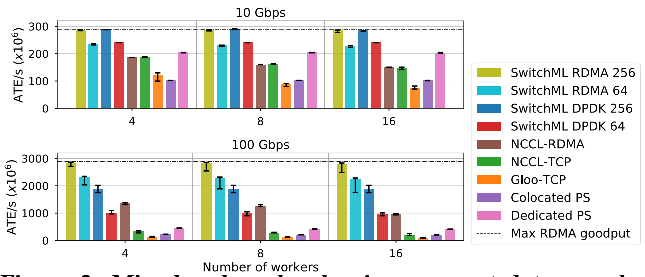


Figure 3: Microbenchmarks showing aggregated tensor elements per second on a 10 (top) and 100 (bottom) Gbps network as workers increase.

which we use to analyze performance by replaying profile traces. This throughput metric includes communication and computation costs, but excludes the time to load data.

Benchmarks. We evaluate SwitchML by training with 8 DNNs introduced in Table 1. The detailed configuration of the benchmarks is in Table 3 in Appendix B. Half of the benchmarks execute on PyTorch and half on TensorFlow.

Setup. As a baseline, we run both PyTorch with native distributed data-parallel module and TensorFlow with Horovod. By default, we use NCCL as the communication library, and use both TCP and RDMA as the transport protocol. Our default setup is to run experiments on 8 workers.

6.1 Tensor aggregation microbenchmarks

To illustrate SwitchML's efficiency in comparison to other communication strategies, we devise a communication-only microbenchmark that performs continuous tensor aggregations, without any gradient computation on the GPU. We verify that the tensors – initially, all ones – are aggregated correctly. We test with various tensor sizes from 50 MB to 1.5 GB. We observe that the number of aggregated tensor elements per time unit (ATE/s) is not very sensitive to the tensor size. Thus, we report results for 100 MB tensors only.

For these experiments, we benchmark SwitchML against the popular all-reduce communication libraries (Gloo [31] and NCCL [77]). We further compare against a parameter server-like scenario, i.e., a set of worker-based processes that assist with the aggregation. To this end, we build a DPDK-based program that implements streaming aggregation as in Algorithm 1. To capture the range of possible PS performance, we consider two scenarios: (1) when the PS processes run on dedicated machines, effectively doubling the cluster size, and (2) when a PS process is co-located with every worker. We choose to run as many PS processes (each using 8 cores) as workers so that the tensor aggregation workload is equally spread among all machines (uniformly sharded) and avoids introducing an obvious performance bottleneck due to over-subscribed bandwidth, which is the case when the ratio of workers to PS nodes is greater than one.

Figure 3 shows the results at 10 and 100 Gbps on three cluster sizes. The results demonstrate the efficiency of SwitchML:

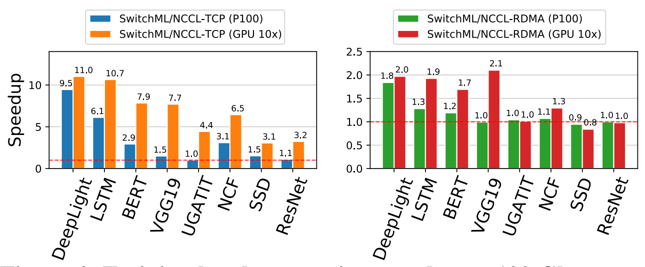


Figure 4: Training batch processing speedup at 100 Gbps considering a P100 GPU and a 10× faster GPU.

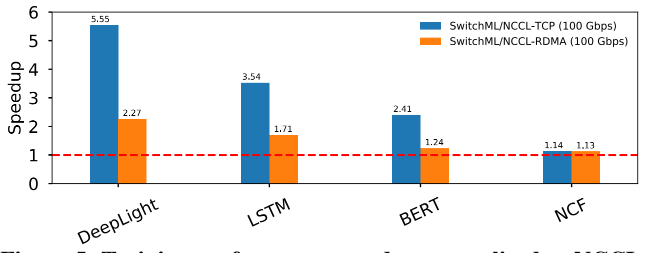


Figure 5: Training performance speedup normalized to NCCL with TCP and RDMA transport protocols.

its highest-performing variant, which uses RDMA with 256-value (1024-byte payload) packets, is within 2% of the maximum achievable goodput. Using smaller packets ($k = 64$ instead of 256) has a noticeable performance impact, underscoring the importance of our multi-pipeline design. The DPDK implementation has additional host-side overhead that prevents it from achieving full link utilization at 100 Gbps. In spite of this, SwitchML can still outperform the best current all-reduce system, NCCL, even when it uses RDMA and SwitchML does not. Moreover, SwitchML always maintains a predictable rate of ATE/s regardless of the number of workers. This trend should continue with larger clusters.

The Dedicated PS approach (with 256 values per packet) – while using *twice* the number of machines *and* network capacity – falls short of matching SwitchML DPDK performance. Unsurprisingly, using the same number of machines as SwitchML, the Colocated PS approach reaches only half of Dedicated PS performance. Our PS implementation is simpler than (and should outperform) a traditional PS, as we do not store the entire model in memory. It demonstrates that, in principle, our aggregation protocol could be run entirely in software on a middlebox, but with lower performance: in-network aggregation inherently requires fewer resources than host-based aggregation.

6.2 SwitchML improves training speed

We analyze training performance on eight DNN benchmarks. We normalize results to NCCL as the underlying communication backend of PyTorch and TensorFlow.

Figure 4 reports the speedup for processing a training batch for SwitchML compared to NCCL at 100 Gbps. SwitchML uses the DPDK implementation with 256-value packets. These results replay the profile traces collected on our cluster

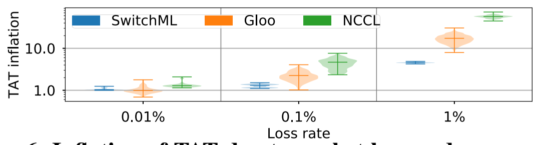


Figure 6: Inflation of TAT due to packet loss and recovery. Results are normalized to a baseline scenario where no loss occurs and the worker implementation does not incur any timer-management overhead.

(§2.2), allowing us to report both the speedup on our testbed GPUs (P100s, which are two generations old) and hypothetical GPUs that are 10× faster (by uniformly scaling the traces by this factor). This emulation lets us evaluate the setting where a fast network is paired with fast GPUs. While it is hard to predict future evolution of GPU speed vs. network bandwidth, we reason that this scaling factor currently corresponds to the span of two to three GPU generations (the A100 benchmarks at 4.2× the V100 [75], which in turn is 1.4-2.2× faster than our P100 [97]) and represents a likely bound on the real-world speedups achievable, which are anyway dependent on the model (e.g., the ResNet50 model sees a nearly 10× speedup from an NVIDIA V100 GPU compared to a K80 GPU [71]) and the other infrastructure specifics.

As expected, SwitchML accelerates batch processing especially for the larger DNNs. The speedup over NCCL-RDMA is at most 2.1×, which is in line with the fundamental 2× advantage of INA over RAR (§3). In most cases, the measured speedup is higher than the emulated communication results (Table 2) predict, because NCCL's RAR implementation does not achieve the theoretical maximum efficiency. The speedup relative to NCCL-TCP is larger (up to one order of magnitude), which is attributable primarily to DPDK's kernel-bypass advantage.

SwitchML provides significant benefits for many, but not all, real-world DNNs, even with 100 Gbps networks. For example, DeepLight and LSTM enjoy major improvements. BERT sees a somewhat lower speedup, in part because its gradient consists of many relatively small (∼60 MB) tensors. Similarly, NCF, a relatively small model, has a modest speedup. Other models, like UGATIT, SSD, and ResNet are simply not network-bound at 100 Gbps. SSD is a particularly challenging case: not only is it a small model that would require an $\alpha = 15.2\times$ faster GPU to become network-bound (Table 1), it also makes many aggregation invocations for small gradients. The overheads of starting an aggregation are not well amortized, especially in the 10× scaled scenario.

Finally, we consider the end-to-end speedup on a complete training run with 16 workers. We focus on the four models that are network-bottlenecked at 100 Gbps. Figure 5 shows the training performance speedup compared to NCCL using RDMA and TCP. These measurements use SwitchML's DPDK implementation, with 256-value packets; we expect a larger speedup once SwitchML's RDMA implementation is integrated with the training framework. Even so, SwitchML's speedups range between 1.13-2.27× over NCCL-RDMA and

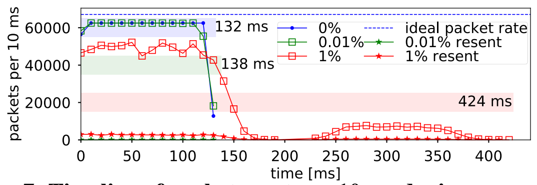


Figure 7: Timeline of packets sent per 10 ms during an aggregation with 0%, 0.01% and 1% packet loss probability. Horizontal bars denote the TAT in each case.

2.05-5.55× over NCCL-TCP. The results are not directly comparable to Figure 4, because (1) they use a larger 16-node cluster, and (2) they report total end-to-end iteration time, which also includes data loading time. Our deployment does not use any optimized techniques for data loading, an orthogonal problem being addressed by other work (e.g., DALI [76]).

6.3 Overheads

Packet loss recovery. We study how packet loss affects TAT. To quantify the change in TAT due to packet loss, we experiment with a uniform random loss probability between 0.01% and 1% applied on every link. The retransmission timeout is set to 1 ms. We run microbenchmark experiments in similar scenarios as §6.1. We report a few representative runs.

Figure 6 measures the inflation in TAT with different loss probabilities. SwitchML completes tensor aggregation significantly faster than Gloo or NCCL when the loss is 0.1% or higher. A loss probability of 0.01% minimally affects TAT in either case. To better illustrate the behavior of SwitchML, we show in Figure 7 the evolution of packets sent per 10 ms at a representative worker for 0.01% and 1% loss. We observe that SwitchML generally maintains a high sending rate – relatively close to the ideal rate – and quickly recovers by retransmitting dropped packets. The slowdown past the 150 ms mark with 1% loss occurs because some slots are unevenly affected by random losses and SwitchML does not apply any form of work-stealing to rebalance the load among aggregators. This presents a further opportunity for optimization.

Tensor scaling and type conversion. We analyze whether any performance overheads arise due to the tensor scaling operations (i.e., multiply updates by f and divide aggregates by f) and the necessary data type conversions: float32-to-int32 → htonl → ntohl → int32-to-float32.

To quantify overheads, we use int32 as the native data type while running the microbenchmarks. This emulates a native float32 scenario with no scaling and conversion operations. We also illustrate the potential improvement of quantization to single-precision (float16) tensors, which halves the volume of data to be sent to the network. (We include a conversion from/to float32.) This setting is enabled by the ability to perform at line rate, in-switch type conversion (float16 ↔ int32), which we verified with the switch chip vendor. However, for this experiment, we emulate this by halving the tensor size.

We find that these overheads are negligible at 10 Gbps.

This is due to our use of x86 SSE/AVX instructions. When we use float16, performance doubles, as expected. However, these overheads become more relevant as data rates increase, requiring to offload type conversion operations to the GPU at 100 Gbps and scaling up the number of cores; RDMA alleviates the pressure for CPU cycles used for I/O (see the gap between DPDK- and RDMA-based performance in Figure 3).

Switch resources. We comment on SwitchML's usage of switch resources in relation to network bandwidth and number of workers. As discussed in Appendix B, our implementation only makes use of the switch ingress pipeline in maximizing the number of vector elements that is processed at line rate. Aggregation bandwidth affects the required pool size. We verified that the memory requirement is less than 10% of switch resources. The number of workers does not influence the resource requirements to perform aggregation at line rate. That said, the number of switch ports and pipelines obviously pose a cap on how many directly-attached workers are supported. A single pipeline in our testbed supports 16-64 workers depending on network speed. We describe how to move beyond a single rack scale in the next section.

7 Extensions

Scaling beyond a rack. We described SwitchML in the context of a rack. However, large scale ML jobs could span beyond a single rack. SwitchML's design can support multiple racks by hierarchically composing several instances of our switch logic, although we do not have a testbed large enough to test (or require) such a design. Each worker is connected to a top-of-rack switch, which aggregates updates from the workers in the rack. Rather than broadcast the result packet to the workers, it instead sends it to a tier-1 aggregation switch, which aggregates updates from multiple racks. This can continue with as many levels as are needed to support the desired network topology. Ultimately, a root switch completes the aggregation of partial aggregates and multicasts a result packet downstream. At each level, the switches further multicast the packet, ultimately reaching the workers.

The hierarchical approach also allows us to support switches with multiple processing pipelines. Suppose a switch has pipelines that can aggregate up to p ports (for the switches we use, $p = 16$). In this setting, each switch aggregates tensors from d downstream ports and forwards partial aggregates via $u = \lceil \frac{d}{p} \rceil$ upstream ports. In other words, the switch operates as u virtual switches, one for each pipeline in the switch.

This hierarchical composition is bandwidth-optimal, as it allows n workers to fully utilize their bandwidth while supporting all-to-all communication with a bandwidth cost proportional to u instead of n. That is, every switch aggregates data in a $p:1$ ratio. As a result, the system naturally supports oversubscription of up to this ratio at the aggregation or core layers. This allows it to support large clusters with relatively shallow hierarchies; using the current generation of 64-port, 4-pipeline 100 Gbps switches, a two-layer hierarchy can support up to 240 workers and a three-layer one manages up to 3,600.

Importantly (and by design), the packet loss recovery algorithm described in §4.5 works in the multi-rack scenario. Thanks to the use of bitmaps and shadow copies, a retransmission originated from a worker will be recognized as a retransmission on all switches that have already processed that packet. This triggers the retransmission of the aggregated packet toward the upper layer switch, ensuring that the switch affected by the packet loss is always ultimately reached.

Congestion control. We have not implemented an explicit congestion control algorithm; the self-clocking streaming protocol is a flow control mechanism to control access to the switch's aggregator slots. It also serves as a rudimentary congestion control mechanism, in that if one worker's link is congested and it cannot process aggregation results at full speed, the self-clocking mechanism will reduce the sending rate of all workers. This is sufficient for dedicated networks (which is common for ML clusters in practice). For more general use, a congestion control scheme may be needed; concurrent work has been developing such protocols [29].

Deployment model. Thus far, we presented SwitchML as an in-network computing approach, focusing on the mechanisms to enable efficient aggregation of model updates at line rate on programmable switching chips with very limited memory. While that might be a viable deployment model in some scenarios, we highlight that our design may have more ample applicability. In fact, one could use a similar design to create a dedicated "parameter aggregator," i.e., a server unit that combines a programmable switching chip with a typical server board, CPU and OS. Essentially a standard server with an advanced network attachment, or in the limit, an array of programmable Smart NICs, each hosting a shard of aggregator slots. The switch component of SwitchML would run on said network attachment. Then, racks could be equipped with such a parameter aggregator, attached for example to the legacy ToR using several 100 Gbps or 400 Gbps ports, or via a dedicated secondary network within the rack directly linking worker servers with it. We expect this would provide similar performance improvements while giving more options for deployment configurations; concurrent work has been exploring a similar approach atop an FPGA board [62].

Multi-job (tenancy). In multi-job or multi-tenant scenarios, the question arises as to how to support concurrent reductions with SwitchML. The solution is conceptually simple. Every job requires a separate pool of aggregators to ensure correctness. As discussed, the resources used for one reduction are much less than 10% of switch capabilities. Moreover, modern switch chips comprise multiple independent pipelines, each with its own resources. Thus, an admission mechanism would be needed to control the assignment of jobs to pools. Alternatively, ATP [54] – a follow up work to ours – explores the idea of partitioning aggregation functionality between a switch (for performance) and a server (for capacity) so as to seamlessly support multi-job scenarios.

Encrypted traffic. Given the cluster setting and workloads we consider, we do not consider it necessary to accommodate for encrypted traffic. Appendix F expands on this issue.

8 Related work

In-network computation trends. The trend towards programmable data planes has sparked a surge of proposals [20, 21, 46, 47, 55, 61] to offload, when appropriate [80], application-specific primitives into network devices.

In-network aggregation. We are not the first to propose aggregating data in the network. Targeting partition-aggregate and big data (MapReduce) applications, NetAgg [65] and CamDoop [18] demonstrated significant performance advantages, by performing application-specific data aggregation at switch-attached high-performance middleboxes or at servers in a direct-connect network topology, respectively. Parameter Hub [64] does the same with a rack-scale parameter server. Historically, some specialized supercomputer networks [2, 26] offloaded MPI collective operators (e.g., all-reduce) to the network. SwitchML differs from all of these approaches in that it performs in-network data reduction using a streaming aggregation protocol.

The closest work to ours is DAIET [86], which was our initial but incomplete proposal of in-network aggregation for minimizing the overhead of exchanging ML model updates.

Mellanox's Scalable Hierarchical Aggregation Protocol (SHARP) is a proprietary in-network aggregation scheme available in certain InfiniBand switches [33]. SHARP uses dedicated on-chip FPUs for collective offloading. The most recent version, SHARPv2 [68] uses streaming aggregation analogous to ours. A key difference is that SHARP builds on InfiniBand where it can leverage link-layer flow control and lossless guarantees, whereas SwitchML runs on standard Ethernet[8] with an unmodified network architecture, necessitating a new packet recovery protocol. More fundamentally, SwitchML builds on programmable network hardware rather than SHARP's fixed-function FPUs, which offers two benefits. First, operators can deploy a single switch model either for SwitchML or traditional networking without waste: the ALUs used for aggregation can be repurposed for other tasks. Second, it allows the system design to evolve to support new ML training approaches. For example, we are currently experimenting with new floating-point representations and protocols for sparse vector aggregations [27]. With a fixed-function approach, these would require new hardware, just as moving from single HPC reductions (SHARPv1) to streaming ML reductions (SHARPv2) required a new ASIC generation.

[8]Although SwitchML uses RDMA, it uses only unreliable connections, and so does *not* require any of the "lossless Ethernet" features of RoCE.

Concurrently, Li et al. [57] explored the idea of in-switch acceleration for Reinforcement Learning (RL). Their design (iSwitch) differs from ours in two fundamental ways. First, while their FPGA-based implementation supports more complex computation (e.g., native floating point), it operates at much lower bandwidth (4×10 Gbps). Second, it stores an entire gradient vector during aggregation; for RL workloads with small models, this works, but it does not scale for large DNN models. Our work targets both large models and high throughput – a challenging combination given the limited on-chip memory in high-speed networking ASICs. SwitchML's software/hardware co-design approach, using a self-clocking streaming protocol, provides $40\times$ higher throughput than iSwitch, while supporting arbitrarily large models.

Finally, targeting NVIDIA's intra-GPU NVLink network, Klenk et al. [52] proposed in-network aggregation in the context of a distributed shared-memory fabric supporting multi-GPU systems where accelerators are directly attached to the fabric. While this work is orthogonal to ours, their push reduction design resembles the SwitchML protocol, suggesting that streaming in-network aggregation has broader applicability than discussed in this paper.

Accelerating DNN training. A large body of work has proposed improvements to hardware and software systems, as well as algorithmic advances for faster DNN training. We only discuss a few relevant prior approaches. Improving training performance via data or model parallelism has been explored by numerous deep learning systems [1, 13, 15, 22, 56, 64, 94]. While data parallelism is most common, it can be advantageous to combine the two approaches. Recent work even shows how to automatically find a fast parallelization strategy for a specific parallel machine [44]. Underpinning any distributed training strategy, lies parameter synchronization. Gibiansky was among the first to research [30] using fast collective algorithms in lieu of the traditional parameter server approach. Many platforms have now adopted this approach [30, 37, 41, 87, 89]. We view SwitchML as a further advancement on this line of work – one that pushes the boundary by co-designing networking functions with ML applications.

9 Conclusion

SwitchML speeds up DNN training by minimizing communication overheads at single-rack scale. SwitchML uses in-network aggregation to efficiently synchronize model updates at each training iteration among distributed workers executing in parallel. We evaluated SwitchML with eight real-world DNN benchmarks on a GPU cluster with 10 Gbps and 100 Gbps networks; we showed that SwitchML achieves training throughput speedups up to $5.5\times$ and is generally better than state-of-the-art collective communications libraries. We are in the process of integrating SwitchML-RDMA in various ML frameworks.

Acknowledgments

We are grateful to Ibrahim Abdelaziz for his work on an initial proof-of-concept of SwitchML and to Omar Alama for contributing with performance improvements and with the code release. We thank Tom Barbette and Georgios Katsikas for their help with DPDK issues. This work benefited from feedback and comments by Steven Hand, Petros Maniatis, KyoungSoo Park, and Amin Vahdat. We thank our shepherd, Sujata Banerjee, and the anonymous reviewers for their helpful feedback. For computer time, this research used the resources of the Supercomputing Laboratory at KAUST.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*, 2016.

[2] N. Adiga, G. Almasi, G. Almasi, Y. Aridor, R. Barik, D. Beece, R. Bellofatto, G. Bhanot, R. Bickford, M. Blumrich, A. Bright, J. Brunheroto, C. Caşcaval, J. Castaños, W. Chan, L. Ceze, P. Coteus, S. Chatterjee, D. Chen, G. Chiu, T. Cipolla, P. Crumley, K. Desai, A. Deutsch, T. Domany, M. Dombrowa, W. Donath, M. Eleftheriou, C. Erway, J. Esch, B. Fitch, J. Gagliano, A. Gara, R. Garg, R. Germain, M. Giampapa, B. Gopalsamy, J. Gunnels, M. Gupta, F. Gustavson, S. Hall, R. Haring, D. Heidel, P. Heidelberger, L. Herger, D. Hoenicke, R. Jackson, T. Jamal-Eddine, G. Kopcsay, E. Krevat, M. Kurhekar, A. Lanzetta, D. Lieber, L. Liu, M. Lu, M. Mendell, A. Misra, Y. Moatti, L. Mok, J. Moreira, B. Nathanson, M. Newton, M. Ohmacht, A. Oliner, V. Pandit, R. Pudota, R. Rand, R. Regan, B. Rubin, A. Ruehli, S. Rus, R. Sahoo, A. Sanomiya, E. Schenfeld, M. Sharma, E. Shmueli, S. Singh, P. Song, V. Srinivasan, B. Steinmacher-Burow, K. Strauss, C. Surovic, R. Swetz, T. Takken, R. Tremaine, M. Tsao, A. Umamaheshwaran, P. Verma, P. Vranas, T. Ward, M. Wazlowski, W. Barrett, C. Engel, B. Drehmel, B. Hilgart, D. Hill, F. Kasemkhani, D. Krolak, C. Li, T. Liebsch, J. Marcella, A. Muff, A. Okomo, M. Rouse, A. Schram, M. Tubbs, G. Ulsh, C. Wait, J. Wittrup, M. Bae, K. Dockser, L. Kissel, M. Seager, J. Vetter, and K. Yates. An Overview of the BlueGene/L Supercomputer. In *SC*, 2002.

[3] A. F. Aji and K. Heafield. Sparse Communication for Distributed Gradient Descent. In *EMNLP*, 2017.

[4] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic. QSGD: Communication-Efficient SGD via Randomized Quantization. In *NIPS*, 2017.

[5] Barefoot Networks. Tofino. `https://barefootnetworks.com/products/brief-tofino/`.

[6] M. Barnett, L. Shuler, R. van de Geijn, S. Gupta, D. G. Payne, and J. Watts. Interprocessor collective communication library (InterCom). In *SHPCC*, 1994.

[7] J. Bernstein, Y.-X. Wang, K. Azizzadenesheli, and A. Anandkumar. signSGD: Compressed Optimisation for Non-Convex Problems. In *ICML*, 2018.

[8] J. Bernstein, J. Zhao, K. Azizzadenesheli, and A. Anandkumar. signSGD with Majority Vote is Communication Efficient And Byzantine Fault Tolerant. *arXiv 1810.05291*, 2018.

[9] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.*, 44(3), July 2014.

[10] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *SIGCOMM*, 2013.

[11] Cerebras. `https://www.cerebras.net`.

[12] C. Chelba, T. Mikolov, M. Schuster, Q. Ge, T. Brants, P. Koehn, and T. Robinson. One Billion Word Benchmark for Measuring Progress in Statistical Language Modeling. *arXiv 1312.3005*, 2013.

[13] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. In *Workshop on Machine Learning Systems*, 2016.

[14] J. H. Cheon, A. Kim, M. Kim, and Y. Song. Homomorphic Encryption for Arithmetic of Approximate Numbers. In *ASIACRYPT*, 2017.

[15] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project Adam: Building an Efficient and Scalable Deep Learning Training System. In *OSDI*, 2014.

[16] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Vargaftik, A. Berger, G. Mendelson, M. Alizadeh, S.-T. Chuang, I. Keslassy, A. Orda, and T. Edsall. dRMT: Disaggregated Programmable Switching. In *SIGCOMM*, 2017.

[17] J. Choquette, O. Giroux, and D. Foley. Volta: Performance and Programmability. *IEEE Micro*, 38(2), 2018.

[18] P. Costa, A. Donnelly, A. Rowstron, and G. O'Shea. Camdoop: Exploiting In-network Aggregation for Big Data Applications. In *NSDI*, 2012.

[19] Criteo's 1TB click prediction dataset. https://docs.microsoft.com/en-us/archive/blogs/machinelearning/now-available-on-azure-ml-criteos-1tb-click-prediction-dataset.

[20] H. T. Dang, M. Canini, F. Pedone, and R. Soulé. Paxos Made Switch-y. *SIGCOMM Comput. Commun. Rev.*, 46(2), 2016.

[21] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé. NetPaxos: Consensus at Network Speed. In *SOSR*, 2015.

[22] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng. Large Scale Distributed Deep Networks. In *NIPS*, 2012.

[23] W. Deng, J. Pan, T. Zhou, D. Kong, A. Flores, and G. Lin. DeepLight: Deep Lightweight Feature Interactions for Accelerating CTR Predictions in Ad Serving. *arXiv 2002.06987*, 2020.

[24] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv 1810.04805*, 2018.

[25] M. Drumond, T. Lin, M. Jaggi, and B. Falsafi. Training DNNs with Hybrid Block Floating Point. In *NIPS*, 2018.

[26] A. Faraj, S. Kumar, B. Smith, A. Mamidala, and J. Gunnels. MPI Collective Communications on The Blue Gene/P Supercomputer: Algorithms and Optimizations. In *HOTI*, 2009.

[27] J. Fei, C.-Y. Ho, A. N. Sahu, M. Canini, and A. Sapio. Efficient Sparse Collective Communication and its application to Accelerate Distributed Deep Learning. Technical report, KAUST, 2020. http://hdl.handle.net/10754/665369.

[28] O. Fercoq, Z. Qu, P. Richtárik, and M. Takáč. Fast distributed coordinate descent for minimizing non-strongly convex losses. In *MLSP*, 2014.

[29] N. Gebara, P. Costa, and M. Ghobadi. In-network Aggregation for Shared Machine Learning Clusters. In *MLSys*, 2021.

[30] A. Gibiansky. Effectively Scaling Deep Learning Frameworks. http://on-demand.gputechconf.com/gtc/2017/presentation/s7543-andrew-gibiansky-effectively-scakukbg-deep-learning-frameworks.pdf.

[31] Gloo. https://github.com/facebookincubator/gloo.

[32] P. Goyal, P. Dollár, R. B. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *arXiv 1706.02677*, 2017.

[33] R. L. Graham, D. Bureddy, P. Lui, H. Rosenstock, G. Shainer, G. Bloch, D. Goldenerg, M. Dubman, S. Kotchubievsky, V. Koushnir, L. Levi, A. Margolin, T. Ronen, A. Shpiner, O. Wertheim, and E. Zahavi. Scalable Hierarchical Aggregation Protocol (SHArP): A Hardware Architecture for Efficient Data Reduction. In *COM-HPC*, 2016.

[34] Graphcore. https://www.graphcore.ai.

[35] Habana Gaudi. https://www.habana.ai/training.

[36] F. M. Harper and J. A. Konstan. The MovieLens Datasets: History and Context. *ACM Trans. Interact. Intell. Syst.*, 5(4), Dec. 2015.

[37] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *HPCA*, 2018.

[38] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. In *CVPR*, 2016.

[39] X. He, L. Liao, H. Zhang, L. Nie, X. Hu, and T.-S. Chua. Neural Collaborative Filtering. In *WWW*, 2017.

[40] S. Horváth, C.-Y. Ho, L. Horváth, A. N. Sahu, M. Canini, and P. Richtárik. Natural Compression for Distributed Deep Learning. *arXiv 1905.10988*, 2019. http://arxiv.org/abs/1905.10988.

[41] F. N. Iandola, M. W. Moskewicz, K. Ashraf, and K. Keutzer. FireCaffe: Near-Linear Acceleration of Deep Neural Network Training on Compute Clusters. In *CVPR*, 2016.

[42] InfiniBand Trade Association. RoCE v2 Specification. https://cw.infinibandta.org/document/dl/7781.

[43] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *USENIX ATC*, 2019.

[44] Z. Jia, M. Zaharia, and A. Aiken. Beyond Data and Model Parallelism for Deep Neural Networks. In *MLSys*, 2019.

[45] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo. A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters. In *OSDI*, 2020.

[46] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica. NetChain: Scale-Free Sub-RTT Coordination. In *NSDI*, 2018.

[47] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *SOSP*, 2017.

[48] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *ISCA*, 2017.

[49] R. Jozefowicz, O. Vinyals, M. Schuster, N. Shazeer, and Y. Wu. Exploring the Limits of Language Modeling. *arXiv 1602.02410*, 2016.

[50] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang. On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. In *ICLR*, 2017.

[51] J. Kim, M. Kim, H. Kang, and K. H. Lee. U-GAT-IT: Unsupervised Generative Attentional Networks with Adaptive Layer-Instance Normalization for Image-to-Image Translation. In *ICLR*, 2020.

[52] B. Klenk, N. Jiang, G. Thorson, and L. Dennison. An In-Network Architecture for Accelerating Shared-Memory Multiprocessor Collectives. In *ISCA*, 2020.

[53] U. Köster, T. J. Webb, X. Wang, M. Nassar, A. K. Bansal, W. H. Constable, O. H. Elibol, S. Gray, S. Hall, L. Hornof, A. Khosrowshahi, C. Kloss, R. J. Pai, and N. Rao. Flexpoint: An Adaptive Numerical Format for Efficient Training of Deep Neural Networks. In *NIPS*, 2017.

[54] C. Lao, Y. Le, K. Mahajan, Y. Chen, W. Wu, A. Akella, and M. Swift. ATP: In-network Aggregation for Multi-tenant Learning. In *NSDI*, 2021.

[55] J. Li, E. Michael, and D. R. K. Ports. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control. In *SOSP*, 2017.

[56] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling Distributed Machine Learning with the Parameter Server. In *OSDI*, 2014.

[57] Y. Li, I.-J. Liu, Y. Yuan, D. Chen, A. Schwing, and J. Huang. Accelerating Distributed Reinforcement Learning with In-Switch Computing. In *ISCA*, 2019.

[58] Y. Li, J. Park, M. Alian, Y. Yuan, Z. Qu, P. Pan, R. Wang, A. Gerhard Schwing, H. Esmaeilzadeh, and N. Sung Kim. A Network-Centric Hardware/Algorithm Co-Design to Accelerate Distributed Training of Deep Neural Networks. In *MICRO*, 2018.

[59] T. Lin, M. Maire, S. J. Belongie, L. D. Bourdev, R. B. Girshick, J. Hays, P. Perona, D. Ramanan, P. Doll'a r, and C. L. Zitnick. Microsoft COCO: Common Objects in Context. In *ECCV*, 2014.

[60] Y. Lin, S. Han, H. Mao, Y. Wang, and B. Dally. Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training. In *ICLR*, 2018.

[61] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya. IncBricks: Toward In-Network Computation with an In-Network Cache. In *ASPLOS*, 2017.

[62] S. Liu, Q. Wang, J. Zhang, Q. Lin, Y. Liu, M. Xu, R. C. Chueng, and J. He. NetReduce: RDMA-Compatible In-Network Reduction for Distributed DNN Training Acceleration. *arXiv 2009.09736*, 2020.

[63] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg. SSD: Single Shot MultiBox Detector. In *ECCV*, 2016.

[64] L. Luo, J. Nelson, L. Ceze, A. Phanishayee, and A. Krishnamurthy. PHub: Rack-Scale Parameter Server for Distributed Deep Neural Network Training. In *SoCC*, 2018.

[65] L. Mai, L. Rupprecht, A. Alim, P. Costa, M. Migliavacca, P. Pietzuch, and A. L. Wolf. NetAgg: Using Middleboxes for Application-Specific On-path Aggregation in Data Centres. In *CoNEXT*, 2014.

[66] D. Masters and C. Luschi. Revisiting small batch training for deep neural networks. *arXiv 1804.07612*, 2018.

[67] Mellanox RDMA Aware Networks Programming User Manual. https://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf.

[68] Mellanox Scalable Hierarchical Aggregation and Reduction Protocol (SHARP). https://www.mellanox.com/products/sharp.

[69] K. Mishchenko, E. Gorbunov, M. Takáč, and P. Richtárik. Distributed Learning with Compressed Gradient Differences. *arXiv 1901.09269*, 2019. http://arxiv.org/abs/1901.09269.

[70] K. Mishchenko, B. Wang, D. Kovalev, and P. Richtárik. IntSGD: Floatless Compression of Stochastic Gradients. *arXiv 2102.08374*, 2021. https://arxiv.org/abs/2102.08374.

[71] D. Narayanan, K. Santhanam, F. Kazhamiaka, A. Phanishayee, and M. Zaharia. Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads. In *OSDI*, 2020.

[72] A. Nemirovski, A. Juditsky, G. Lan, and A. Shapiro. Robust Stochastic Approximation Approach to Stochastic Programming. *SIAM J. Optim.*, 19(4), 2009.

[73] A. Nemirovski and D. B. Yudin. *Problem complexity and method efficiency in optimization*. Wiley Interscience, 1983.

[74] NVIDIA Ampere Architecture In-Depth. https://devblogs.nvidia.com/nvidia-ampere-architecture-in-depth/.

[75] NVIDIA's Ampere A100 GPU Is Unstoppable, Breaks 16 AI Performance Records, Up To 4.2x Faster Than Volta V100. https://wccftech.com/nvidia-ampere-a100-fastest-ai-gpu-up-to-4-times-faster-than-volta-v100/.

[76] NVIDIA Data Loading Library (DALI). https://developer.nvidia.com/DALI.

[77] NVIDIA Collective Communication Library (NCCL). https://developer.nvidia.com/nccl.

[78] NVIDIA Data Center Deep Learning Product Performance. https://developer.nvidia.com/deep-learning-performance-training-inference.

[79] P. Patarasuk and X. Yuan. Bandwidth Optimal All-reduce Algorithms for Clusters of Workstations. *Journal of Parallel and Distributed Computing*, 69(2), 2009.

[80] D. R. K. Ports and J. Nelson. When Should The Network Be The Computer? In *HotOS*, 2019.

[81] P. Rajpurkar, R. Jia, and P. Liang. Know What You Don't Know: Unanswerable Questions for SQuAD. In *ACL*, 2018.

[82] P. Richtárik and M. Takáč. Distributed Coordinate Descent Method for Learning with Big Data. *Journal of Machine Learning Research*, 17(1), 2016.

[83] H. Robbins and S. Monro. A Stochastic Approximation Method. *The Annals of Mathematical Statistics*, 22(3), 1951.

[84] C. Rosset. Turing-NLG: A 17-billion-parameter language model by Microsoft. https://www.microsoft.com/en-us/research/blog/turing-nlg-a-17-billion-parameter-language-model-by-microsoft/.

[85] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, 115(3), 2015.

[86] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis. In-Network Computation is a Dumb Idea Whose Time Has Come. In *HotNets*, 2017.

[87] F. Seide and A. Agarwal. CNTK: Microsoft's Open-Source Deep-Learning Toolkit. In *KDD*, 2016.

[88] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu. 1-Bit Stochastic Gradient Descent and Application to Data-Parallel Distributed Training of Speech DNNs. In *INTERSPEECH*, 2014.

[89] A. Sergeev and M. D. Balso. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv 1802.05799*, 2018.

[90] S. Shalev-Shwartz and S. Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014.

[91] K. Simonyan and A. Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *ICLR*, 2015.

[92] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking. Packet Transactions: High-Level Programming for Line-Rate Switches. In *SIGCOMM*, 2016.

[93] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of Collective Communication Operations in MPICH. *Int. J. High Perform. Comput. Appl.*, 19(1), Feb. 2005.

[94] J. Wei, W. Dai, A. Qiao, Q. Ho, H. Cui, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Managed Communication and Consistency for Fast Data-Parallel Iterative Analytics. In *SoCC*, 2015.

[95] W. Wen, C. Xu, F. Yan, C. Wu, Y. Wang, Y. Chen, and H. Li. TernGrad: Ternary Gradients to Reduce Communication in Distributed Deep Learning. In *NIPS*, 2017.

[96] H. Xu, C.-Y. Ho, A. M. Abdelmoniem, A. Dutta, E. H. Bergou, K. Karatsenidis, M. Canini, and P. Kalnis. Compressed Communication for Distributed Deep Learning: Survey and Quantitative Evaluation. Technical report, KAUST, Apr 2020. http://hdl.handle.net/10754/662495.

[97] R. Xu, F. Han, and N. Dandapanthula. Deep Learning on V100. Technical report, DELL HPC Innovation Lab, 2017. https://downloads.dell.com/manuals/all-products/esuprt_software/esuprt_it_ops_datcentr_mgmt/high-computing-solution-resources_white-papers16_en-us.pdf.

[98] Y. You, Z. Zhang, C.-J. Hsieh, J. Demmel, and K. Keutzer. Speeding up ImageNet Training on Supercomputers. In *MLSys*, 2018.

[99] S. Zhou, Z. Ni, X. Zhou, H. Wen, Y. Wu, and Y. Zou. DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients. *arXiv 1606.06160*, 2016.

A Example execution

We illustrate through an example how Algorithms 3 and 4 behave in a system with three workers: w1, w2, and w3. We focus on the events that occur for a particular slot (x) starting from a particular offset (*off*).

- **t0:** Worker w1 sends its model update for slot x with offset = *off*.
- **t1:** Worker w2 sends its model update for slot x with offset = *off*.
- **t2:** Worker w3 sends its model update for slot x with offset = *off*. This update packet is lost on the upstream path at time **t3**, and hence the switch does not receive it.
- **t4:** w1's timeout kicks in for the model update sent at **t0**, leading to a retransmission of the same model update for slot x with offset = *off*. The switch receives the packet, but it ignores the update because it already received and aggregated an update from w1 for the given slot and offset.
- **t5:** w2's timeout kicks in for the model update sent at **t0**, leading to a retransmission of the same model update for slot x with offset = *off*. The switch receives the packet, but it ignores the update because it already received and aggregated an update from w2 for the given slot and offset.
- **t6:** w3's timeout kicks in for the model update sent at **t0**, leading to a retransmission of the same model update for slot x with offset = *off*. The switch receives the packet and aggregates the update properly. Since this update is the last one for slot x and offset *off*, the switch completes aggregation for the slot and offset, turns the slot into a shadow copy, and produces three response packets (shown as blue arrows).
- **t7:** The first response packet for w1 is lost on the downstream path, and w1 does not receive it.
- **t8:** Not having received the result packet for the update packets sent out earlier (at **t0** and **t4**), w1 retransmits its model update the second time. This retransmission reaches the switch correctly, and the switch responds by sending a unicast response packet for w1.
- **t9** and **t10:** w2 and w3 respectively receives the response packet. Hence, w2 and w3 respectively decides to reuse slot x for the next offset ($\mathit{off} + k \cdot s$) and sends their new updates at **t12** and **t13**.
- **t11:** The unicast response packet triggered by the second model-update retransmission (sent at **t8**) arrives at w1.
- **t14:** Now that w1 has received its response, it decides to reuse slot x for the next offset ($\mathit{off} + k \cdot s$) and sends its new updates. This update arrives at the switch at **t15**, upon which the switch realizes that the slot for offset ($\mathit{off} + k \cdot s$) is complete. This confirms that the result in the shadow-copy slot (the slot in pool 0) is safely received by every worker. Thus, the switch flips the roles of the slots again.

B Implementation details

Switch component. The main challenge we faced was to find a design that best utilizes the available resources (SRAM, TCAMs, hashing machinery, etc.) to perform as much computation per packet as possible. Data plane programs are typically constrained by either available execution resources or available storage; for SwitchML, execution resources are the tighter constraint. For example, a data plane program is constrained by the number of stages per pipeline [92], which limits the dependencies within the code. In fact, every action whose execution is contingent on the result of a previous operation has to be performed on a subsequent stage. A program with too many dependencies cannot find a suitable allocation

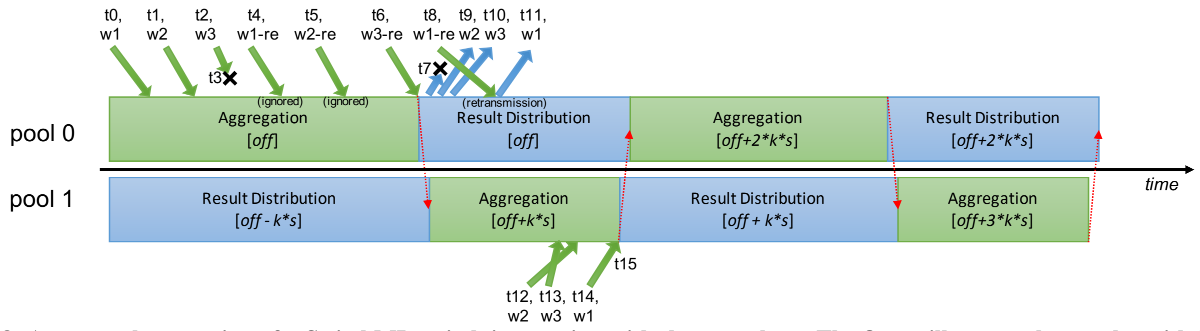


Figure 8: An example execution of a SwitchML switch interacting with three workers. The figure illustrates how a slot with index *x* is used during the different phases (shown in different colors) that alternate between the two pools.

on the hardware pipeline and will be rejected by the compiler. Moreover, the number of memory accesses per-stage is inherently limited by the maximum per-packet latency; a switch may be able to parse more data from a packet than it is able to store into the switch memory during that packet's time in the switch.

We make a number of design trade-offs to fit within the switch constraints. First, our P4 program makes the most use of the limited memory operations by performing the widest register accesses possible (64 bits). We then use the upper and lower part of each register for alternate pools. These parts can execute different operations simultaneously; for example, when used for the received work bitmap, we can set a bit for one pool and clear a bit for the alternate pool in one operation. Second, we minimize dependencies (e.g., branches) in our Algorithm 3 in order to process 64 elements per packet within a single ingress pipeline. We confine all processing to the ingress pipeline; when the aggregation is complete, the traffic manager duplicates the packet containing the aggregated result and performs a multicast. In a first version of our program, we used both ingress and egress pipelines for the aggregation, but that required packet recirculation to duplicate the packets. This caused additional dependencies that required more stages, preventing the processing of more than 64 elements per packets. Moreover, this design experienced unaccounted packet losses between the two pipelines and during recirculation, which led us to search for a better, single pipeline, program.

Worker component. Our goal for implementing the worker component is to achieve high I/O performance for aggregating model updates. At the same time, we want to support existing ML frameworks without modifications.

In existing ML frameworks, a DNN model update U comprises of a set of tensors T, each carrying a subset of the gradients. This is because the model consists of many layers; most existing frameworks emit a gradient tensor per layer and reduce each layer's tensors independently. Back-propagation produces the gradients starting from the output layer and moving towards the input layer. Thus, communication can start on the output layer's gradients while the other gradients are still being computed, partially overlapping communication with computation. This implies that for each iteration, there are as many aggregation tasks as the number of tensors (e.g., 152 for ResNet50).

Our implementation exposes the same synchronous all-reduce interface as Gloo. However, rather than treating each tensor as an independent reduction and resetting switch state for each one, our implementation is efficient in that it treats the set of tensors virtually as a single, continuous stream of data across iterations. Upon invocation, our API passes the input tensor to a virtual stream buffer manager which streams the tensor to the switch, breaking it into the small chunks the switch expects. Multiple threads may call SwitchML's all-reduce, with the requirement that each worker machine's tensor reduction calls must occur in the same order; the stream buffer manager then performs the reductions and steers results to the correct requesting thread.

One CPU core is sufficient to do reduction at line rate on a 10 Gbps network. However, to be able to scale beyond 10 Gbps, we use multiple CPU cores at each worker and use the Flow Director technology (implemented in hardware on modern NICs) to uniformly distribute incoming traffic across the NIC RX queues, one for each core. Every CPU core runs an I/O loop that processes every batch of packets in a run-to-completion fashion and uses a disjoint set of aggregation slots. Packets are batched in groups of 32 to reduce per-packet transmission overhead. We use x86 SSE/AVX instructions to scale the model updates and convert between types. We are careful to ensure all processing is NUMA aware.

RDMA implementation details. We found that the cost of processing individual SwitchML packets, even using DPDK with 256-element packets and multiple cores, was too high to achieve line rate. Other aggregation libraries use RDMA to offload packet processing to the NIC. In RDMA-based systems NICs implement packetization, flow control, congestion control, and reliable delivery. In normal usage, clients use RDMA to send and receive messages of up to a gigabyte; the NIC turns them into packets and ensures they are delivered reliably. Furthermore, clients may register memory regions with the NIC, allowing other clients to remotely read and write them without CPU involvement. This reduces or eliminates work done on the clients' CPUs to complete the transfer.

Turning a Tofino switch into a fully-featured RDMA endpoint is not the solution. Implementing timeouts and retransmission in a way that is compatible with existing poorly-documented existing RDMA NICs would be complex. Furthermore, such an implementation would not be an good fit for SwitchML: the RDMA protocols are largely designed for point-to-point communication, whereas SwitchML's protocol is designed for collective communication.

Fortunately, RDMA NICs implement multiple protocols with different properties. The standard Reliable Connected (RC) mode ensures reliable delivery and supports CPU-bypassing remote reads and writes (as well as sends and receives) of up to 1 GB. The UDP-like Unreliable Datagram (UD) mode supports just sends and receives of up the network MTU. Finally, the Unreliable Connected (UC) mode fits somewhere in between. It supports packetization, allowing for sends, receives, and writes of up to 1 GB. It also generates and checks sequence numbers, allowing it to detect packet drops, but it does not retransmit: instead, if a gap in sequence numbers is detected, incoming packets are silently dropped until the first packet of a new message arrives. Then, the sequence number counter is reset to the sequence number of that packet, and normal reception continues.

We use RDMA UC to implement a RDMA-capable variant of SwitchML using a subset of the RoCE v2 protocol [42]. Its operation is very similar to what is described in Section 4, with three main differences.

First, where base SwitchML sends and receives slot-sized packets, SwitchML RDMA sends multi-slot messages. Each packet of a message is treated largely as it is in the base protocol by the switch, but the pool index for each packet is computed as an offset from the base index provided with the first packet of the message. Timeouts are tracked just as they are in the base protocol, but when a packet drop is detected, the client retransmits the entire message rather than just the dropped packet. This makes retransmissions more expensive, but it also drastically lowers the cost incurred sending packets in the common case of no packet drops; since packet drops are rare within a datacenter, the benefit is large.

Second, SwitchML consumes and generates sequence numbers on the switch. In order to allow messages with multiple packets to aggregate concurrently, each in-flight message is allocated its own queue pair, with its own sequence number register. This allows clients to to be notified when a write message from the switch has arrived with no drops; it also allows the switch to ignore packets in messages received out of sequence. However, the same per-slot bitmap used in the base protocol is still used to ensure that duplicate packets from a retransmission of a partially-received messages are not re-applied. Packets are transmitted as individual slots addressed by a message complete. This means that the packets from multiple messages may interleave on the wire, but since each is on a separate queue pair with its own sequence number

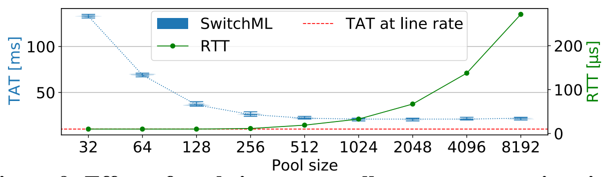


Figure 9: Effect of pool size on overall tensor aggregation time (TAT) and per-packet RTT (right y-axis) at 100 Gbps.

space, the NIC will reassemble them successfully.

Third, SwitchML RDMA uses RDMA Write Immediate messages for all communication. This allows clients to send data directly from GPU memory, and the switch to write directly into GPU memory (if the host is GPU Direct-capable). Byte order conversion and scaling are done on the GPU; the CPU is responsible only for issuing writes when data from the GPU is ready, detecting completions and timeouts, and issuing retransmissions when necessary. Necessary metadata for the SwitchML protocol is encoded in fields of the RDMA header; the RDMA RKey and Address fields are used to encode the destination slot and the address to write the response to. The Immediate field is used to carry up to four scaling factors. At job setup time, the clients communicate with the switch and give it their queue pair numbers, initial sequence numbers, and an RKey for its switch-writable memory region. The switch uses these to form RDMA Write Immediate messages with appropriate sequence numbers, destination addresses, and immediate values, of the same size as the messages sent from the clients to the switch.

Finally, it is important to note that SwitchML RDMA does not require lossless Ethernet to be configured, as is common in RoCE deployments. Enabling lossless Ethernet would reduce the probability of packet drops, but would add complexity to the network deployment. SwitchML's reliability protocol makes this unnecessary.

DNN workloads. Table 3 details the models, datasets and ML toolkits used in the experiments.

C Tuning the pool size

As mentioned, the pool size s affects performance and reliability. We now analyze how to tune this parameter.

Two factors affect s. First, because s defines the number of in-flight packets in the system that originate from a worker, to avoid wasting each worker's network bandwidth, s should be no less than the bandwidth-delay product (BDP) of each worker. Note that the delay here refers to the end-to-end delay, including the end-host processing time, which can be easily measured in a given deployment. Let b be the packet size, which is constant in our setting. To sustain line rate transmission, the stream of response packets must arrive at line rate, and this is possible when $s \cdot b$ matches the BDP. A significantly higher value of s, when used as the initial window size, will unnecessarily increase queuing time within the workers.

Model	Task	Dataset	Sys
DeepLight [23]	Click-through Rate Prediction	Criteo 1TB [19]	PyT
LSTM [49]	Language Modeling	GBW [12]	PyT
NCF [39]	Recommendation	ML-20m [36]	PyT
BERT [24]	Question Answering	SQuAD [81]	TF
VGG19 [91]	Image Classification	ImageNet-1K [85]	PyT
UGATIT [51]	Image-to-Image Translation	Selfie2Anime [51]	TF
ResNet50 [38]	Image Classification	ImageNet-1K [85]	TF
SSD [63]	Object Detection	COCO 2017 [59]	PyT

Table 3: DDL benchmarks. Models, task, dataset used for training and ML toolkit (PyT=PyTorch; TF=TensorFlow).

Model	Metric	No compression	Top-10%	Top-1%
DeepLight	AUC	0.9539	0.9451	0.9427
LSTM	Perplexity	32.74	86.26	82.55
NCF	Hit rate	0.9586	0.9369	-
BERT	F1 score	91.60	91.47	91.52
VGG19	Top-1 accuracy	68.12	64.85	57.70
UGATIT	-	-	-	-
ResNet50	Top-1 accuracy	74.34	74.59	72.63
SSD	Accuracy	0.2549	0.2487	0.2309

Table 4: Test metrics comparison. NCF at Top-1% did not converge. BERT result is the median of 6 runs of fine-tuning from a pre-trained model. BERT pre-training results are shown in Figure 13. UGATIT fails to execute with the compressor implementation in [96]. See Figure 10 for the convergence behavior during training.

Second, a correctness requirement for our communication scheme is that no two in-flight packets from the same worker use the same slot (as no worker node can ever lag behind by more than one phase). To sustain line rate and preserve correctness, the lower bound on s is such that $s \cdot b$ matches the BDP. Therefore, the optimal s is for $\lceil BDP/b \rceil$.

In practice, we select s as the next power of two of the above formula because the DPDK library – which we use to implement SwitchML – performs batched send and receive operations to amortize system overheads. Based on our measurements (Figure 9), we use 128 and 512 as the pool size for 10 and 100 Gbps, respectively. This occupies 256 KB and 1 MB of register space in the switch, respectively. We note that the switch can support one order of magnitude more slots, and SwitchML uses much less than 10% of that available.

D Compression affects convergence

Table 4 reports the model quality obtained without gradient compression and with Top-k compression using $k = 1\%, 10\%$. Model quality is assessed using per-model accuracy metrics.

Results are shown in Figure 10. We observe that loss and accuracy do not necessary correlate well. For example, in the case of SSD all methods have similar loss trace, but obvious accuracy gap. For NCF, Top-k at 1% does not converge, but the accuracy can still go up.

E Model quantization

To the best of our knowledge, no Ethernet switching chip offers floating-point operations in the dataplane for packet processing. Some InfiniBand switching chips have limited support for floating-point operations for scientific computing [33]. We also confirmed that the state-of-the-art programmable Ethernet switching chips do not support native floating-point operations either. These observations lead us to two main questions.

Where should the type conversion occur? In theory either workers or the switch can perform type conversion. In the former case, packets carry a vector of integer types while in the latter case the switch internally performs the type conversions. It turns out to be possible to implement a restricted form of 16-bit floating point on a Tofino chip by using lookup tables and ALUs to do the conversion. This means there is no type conversion overhead at end hosts. However, this approach still requires to scale the gradient values due to the limited range of floating point conversion the switch can perform. Besides, unless half-precision training is used, the worker must still convert from 32-bit to 16-bit floating points. At the same time, an efficient implementation that uses modern x86 vector instructions (SSE/AVX) to implement type conversion sees only a negligible overhead (see Figure 11).

What are the implications in terms of accuracy? Recently, several update compression (e.g., quantization, dithering or sparsification) strategies were proposed (see [96] for a survey) to be used with standard training methods, such as SGD, and bounds were obtained on how these strategies influence the number of iterations until a sufficient convergence criterion is satisfied (e.g., being sufficiently close to minimizing the empirical loss over the data set). These include aggressive 1-bit compression of SGD for DNNs [88], signSGD [7, 8], QSGD [4], which uses just the sign of the stochastic gradients to inform the update, Terngrad [95], which uses ternary quantization, and the DIANA framework [69], which allows for a wide array of compression strategies applied to gradient differences. All the approaches above use lossy randomized compression strategies that preserve unbiasedness of the stochastic gradients at the cost of increasing the variance of gradient estimators, which leads to worse iteration complexity bounds. Thus, there is a trade-off between savings in communication and the need to perform more training rounds. In contrast, our mechanism is not randomized, and for a suitable selection of a scaling parameter f, is essentially lossless or suffers negligible loss only.

We shall now briefly describe our quantization mechanism. Model updates are divided into packet-sized blocks. With a small abuse of notation, in the following all equations refer to per-block operations. Each worker multiplies its block model update $\Delta_i^t = \Delta(x^t, D_i^t)$ by a vector of scaling factors $f > 0$, obtaining $f\Delta_i^t$. The elements of f are chosen per block, and are chosen such that that all k entries of the scaled update can be rounded to a number representable as an integer without overflow. We then perform this rounding, obtaining vectors $Q_i^t = \rho(f\Delta_i^t) \in \mathbb{Z}^k$ for $i = 1, 2, \ldots, n$, where ρ is the rounding operator, which are sent to the switch and aggregated,

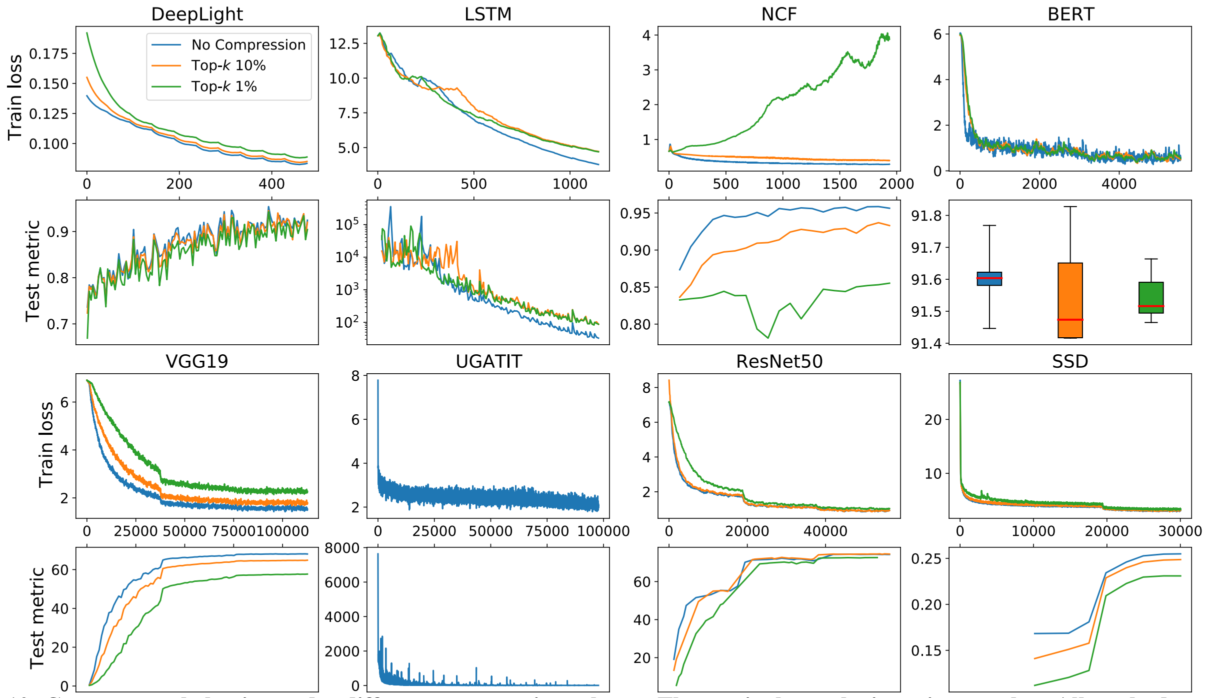


Figure 10: Convergence behavior under different compression scheme. The x axis shows the iteration number. All methods execute a fixed number of steps and hyperparameters are kept the same. Refer to Table 4 for test metrics of each task. We plot generator loss as the metric for UGATIT because there is no objective metric available for GAN workloads. Note that for the perplexity metric of LSTM, lower is better.

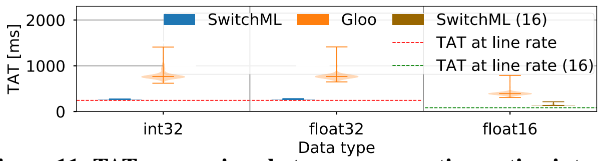


Figure 11: TAT comparison between aggregating native-integer tensors, scaling and converting from single-precision (float32) tensors, and scaling and converting from half-precision (float16) tensors.

resulting in

$$A^t = \sum_{i=1}^{n} Q_i^t.$$

Again, we need to make sure f is not too large so that A^t can be represented as an integer without overflow. The aggregated update A^t is then scaled back on the workers, obtaining A^t/f, and the model gets updated as follows:

$$x^{t+1} = x^t + \frac{1}{f} A^t.$$

Let us illustrate this on a simple example with $n = 2$ and $d = 1$. Say $\Delta_1^t = 1.56$ and $\Delta_2^t = 4.23$. We set $f = 100$ and get

$$Q_1^t = \rho(f\Delta_1^t) = \rho(156) = 156$$

and

$$Q_2^t = \rho(f\Delta_2^t) = \rho(423) = 423.$$

The switch will add these two integers, which results in $A^t = 579$. The model then gets updated as:

$$x^{t+1} = x^t + \frac{1}{100} 579 = x^t + 5.79.$$

Notice that while the aggregation was done using integers only (which is necessitated by the limitations of the switch), the resulting model update is identical to the one that would be applied without any conversion in place. Let us consider the same example, but with $f = 10$ instead. This leads to

$$Q_1^t = \rho(f\Delta_1^t) = \rho(15.6) = 16$$

and

$$Q_2^t = \rho(f\Delta_2^t) = \rho(42.3) = 42.$$

The switch will add these two integers, which results in $A^t = 58$. The model then gets updated as:

$$x^{t+1} = x^t + \frac{1}{10} 58 = x^t + 5.8.$$

Note that this second approach leads to a small error. Indeed, while the true update is 5.79, we have applied the update 5.8 instead, incurring the error 0.01.

Our strategy is to apply the above trick, but take special care about how we choose the scaling factor f so that the trick works throughout the entire iterative process with as little information loss as possible.

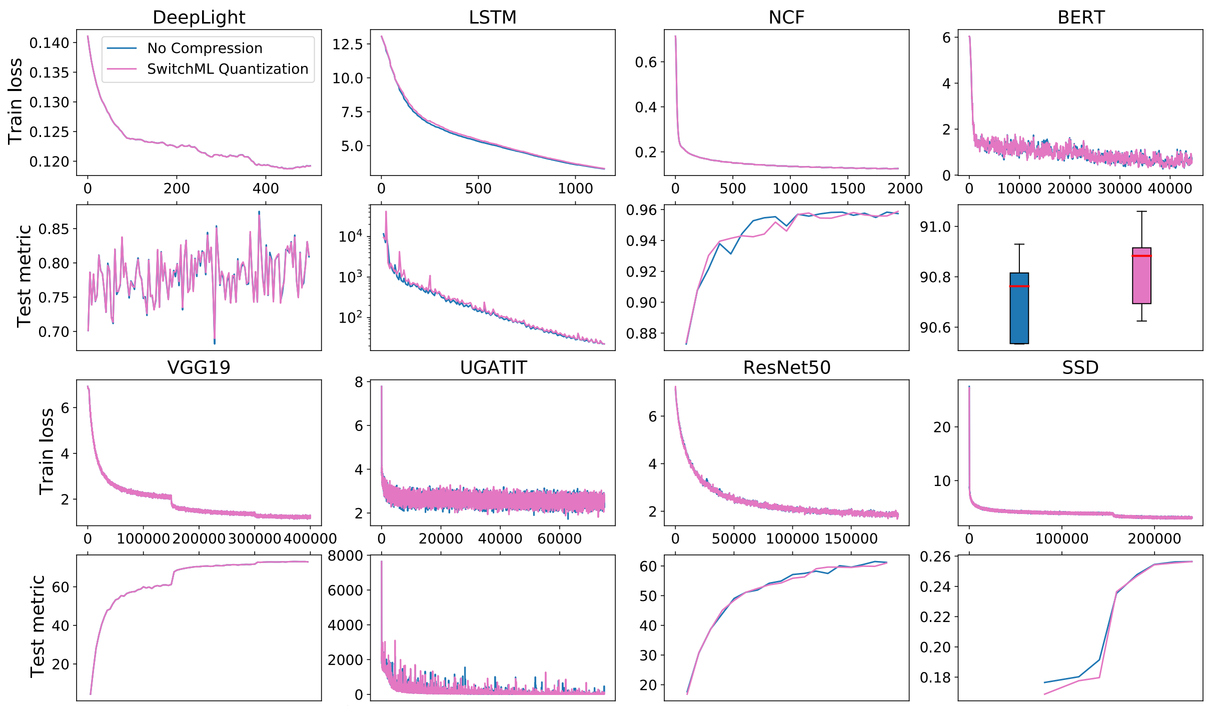


Figure 12: Convergence analysis of SwitchML quantization method on a single GPU. The x axis shows the iteration number.

A formal model. Let us now formalize the above process. We first assume that we have a scalar $f > 0$ for which the following holds:

Assumption 1. $|\rho(f\Delta_i^t)| \leq 2^{31}$ *for all* $i = 1, 2, \ldots, n$ *and all iterations* t.

Assumption 2. $|\sum_{i=1}^n \rho(f\Delta_i^t)| \leq 2^{31}$ *for all iterations* t.

The above assumptions postulate that all numbers which we obtain by scaling and rounding on the nodes (Assumption 1), and by aggregation on the switch (Assumption 2), can be represented as integers without overflow.

We will now establish a formal statement which characterizes the error incurred by our aggregation procedure.

Theorem 1 (Bounded aggregation error). The difference between the exact aggregation value $\sum_{i=1}^n \Delta_i^t$ *(obtained in case of perfect arithmetic without any scaling and rounding, and with a switch that can aggregate floats) and the value* $\frac{1}{f}A^t = \frac{1}{f}\sum_{i=1}^n \rho(f\Delta_i^t)$ *obtained by our procedure is bounded by* $\frac{n}{f}$.

Proof. To prove the above result, notice that

$$\begin{aligned}\frac{1}{f}\sum_{i=1}^n \rho(f\Delta_i^t) &\leq \frac{1}{f}\sum_{i=1}^n \lceil f\Delta_i^t \rceil \\ &\leq \frac{1}{f}\sum_{i=1}^n (f\Delta_i^t + 1) \\ &= \left(\sum_{i=1}^n \Delta_i^t\right) + \frac{n}{f}.\end{aligned}$$

Using the same argument, we get a similar lower bound

$$\begin{aligned}\frac{1}{f}\sum_{i=1}^n \rho(f\Delta_i^t) &\geq \frac{1}{f}\sum_{i=1}^n \lfloor f\Delta_i^t \rfloor \\ &\geq \frac{1}{f}\sum_{i=1}^n (f\Delta_i^t - 1) \\ &= \left(\sum_{i=1}^n \Delta_i^t\right) - \frac{n}{f}.\end{aligned}$$

□

Note that the error bound postulated in Theorem 1 improves as f increases, and n decreases. In practice, the number of nodes is constant $n = O(1)$. Hence, it makes sense to choose f as large as possible while making sure Assumptions 1 and 2 are satisfied. Let us give one example for when these assumptions are satisfied. In many practical situations it is known that the model parameters remain bounded:[9]

Assumption 3. There exists $B > 0$ *such that* $|\Delta_i^t| \leq B$ *for all* i *and* t.

As we shall show next, if Assumption 3 is satisfied, then so is Assumption 1 and 2.

Theorem 2 (No overflow). Let Assumption 3 be satisfied. Then Assumptions 1 and 2 are satisfied (i.e., there is no overflow) as long as $0 < f \leq \frac{2^{31}-n}{nB}$.

Proof. We have $\rho(f\Delta_i^t) \leq f\Delta_i^t + 1 \leq f|\Delta_i^t| + 1 \leq fB + 1$. Likewise, $\rho(f\Delta_i^t) \geq f\Delta_i^t - 1 \geq -f|\Delta_i^t| - 1 = -(fB + 1)$. So,

[9]If desirable, this can be enforced explicitly by the inclusion of a suitable hard regularizer, and by using projected SGD instead of plain SGD.

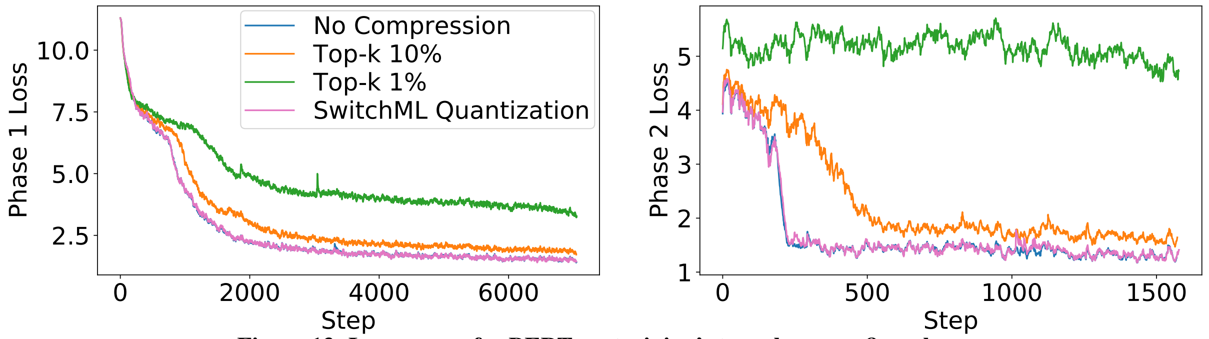


Figure 13: Loss curves for BERT pretraining's two phases on 8 workers.

$|\rho(f\Delta_i^t)| \leq fB+1$. Hence, as soon as $0 < f \leq \frac{2^{31}-1}{B}$, Assumption 1 is satisfied. This inequality is less restrictive as the one we assume. Similarly, $|\sum_i \rho(f\delta_i^t)| \leq \sum_i |\rho(f\Delta_i^t)| \leq \sum_i (fB+1) = n(fB+1)$. So, Assumption 2 is satisfied as long as $n(fB+1) \leq 2^{31}$, i.e., as long as $0 < f \leq \frac{2^{31}-n}{nB}$. $\square$

We now put all of the above together. By combining Theorem 1 (bounded aggregation error) and Theorem 2 (no overflow), and if we choose $f = \frac{2^{31}-n}{nB}$, then the difference between the exact update $\sum_i \Delta_i^t$ and our update $\frac{1}{f}\sum_i \rho(f\Delta_i^t)$ is bounded by $\frac{n^2B}{2^{31}-n}$. In typical applications, $n^2B \ll 2^{31}$, which means that the error we introduce is negligible.

Empirical study. We name the above method "SwitchML quantization" and run end-to-end experiments to study its convergence behavior. Figure 12 shows that SwitchML quantization achieves similar training loss and does not incur test metric decrease in all models detailed in Table 4. We further show in Figure 13 that SwitchML quantization has little to no difference in the training loss curve of BERT [24] pretraining compared to the no compression baseline, while Top-k compression with $k = 1\%, 10\%$ has a big impact on model convergence.

F Encrypted traffic

A recent trend, especially at cloud providers, is to encrypt all datacenter traffic. In fact, data encryption is generally performed at the NIC level itself. While addressing this setting is out of scope, we wish to comment on this aspect. We believe that given our substantial performance improvements, one might simply forego encryption for ML training traffic.

We envision a few alternatives for when that is not possible. One could imagine using HW accelerators to enable in-line decryption/re-encryption at switches. However, that is likely costly. Thus, one may wonder if computing over encrypted data at switches is possible. While arbitrary computations over encrypted data are beyond current switches' capabilities, we note that the operation performed at switches to aggregate updates is simple integer summation. The appealing property of several partially homomorphic cryptosystems (e.g., Paillier) is that the relation $E(x) \cdot E(y) = E(x+y)$ holds for any two values x, y (where E denotes encryption). For instance, recent work by Cheon et al. [14] developed a homomorphic encryption scheme for approximate arithmetic. By customizing the end-host encryption process, the worker could encrypt all the vector elements using such a cryptosystem, knowing that the aggregated model update can be obtained by decrypting the data aggregated at the switches. We leave it to future work to validate this concept.

Efficient Wideband Spectrum Sensing Using MEMS Acoustic Resonators

Junfeng Guan, Jitian Zhang, Ruochen Lu, Hyungjoo Seo,
Jin Zhou, Songbin Gong, Haitham Hassanieh
Univeristy of Illinois at Urbana-Champaign

Abstract – This paper presents S^3, an efficient wideband spectrum sensing system that can detect the real-time occupancy of bands in large spectrum. S^3 samples the wireless spectrum below the Nyquist rate using cheap, commodity, low power analog-to-digital converters (ADCs). In contrast to existing sub-Nyquist sampling techniques, which can only work for sparsely occupied spectrum, S^3 can operate correctly even in dense spectrum. This makes it ideal for practical environments with dense spectrum occupancy, which is where spectrum sensing is most useful. To do so, S^3 leverages MEMS acoustic resonators that enable spike-train like filters in the RF frequency domain. These filters sparsify the spectrum while at the same time allow S^3 to monitor a small fraction of bandwidth in every band.

We introduce a new structured sparse recovery algorithm that enables S^3 to accurately detect the occupancy of multiple bands across a wide spectrum. We use our fabricated chip-scale MEMS spike-train filter to build a prototype of an S^3 spectrum sensor using low power off-the-shelf components. Results from a testbed of 19 radios show that S^3 can accurately detect the channel occupancies over a 418 MHz spectrum while sampling 8.5× below the Nyquist rate even if the spectrum is densely occupied.

1 Introduction

The past decade has witnessed significant changes in the wireless spectrum as the FCC (Federal Communications Committee) has repurposed many frequency bands for dynamic spectrum sharing. This includes the 6 GHz band, released in April 2020, to be shared between Wi-Fi 6E and the incumbent users in this band like microwave backhaul [14]. Another example is the 3.5 GHz Citizens Broadband Radio Service (CBRS) band, which was recently approved for commercial deployments in September 2019. To leverage the CBRS band, unlicensed devices must sense a 200 MHz spectrum and avoid causing interference to primary and licensed users like military radars [13]. Of course, an earlier and more well-known example of spectrum sharing is the TV White Spaces which were released in 2010 [15]. Moreover, there are lots of opportunities for spectrum sharing in the millimeter-wave frequencies. In particular, the FCC released 14 GHz of unlicensed spectrum in the 60 GHz band that can be shared among Wi-Fi and IoT technologies [16]. These changes have been driven by the ever-increasing demand for wireless connectivity and aim to exploit previously underutilized frequency bands to accommodate new unlicensed applications and achieve highly efficient usage of the spectrum.

Efficient and truly dynamic spectrum sharing, however, requires unlicensed devices to sense wideband spectrum (hundreds of MHz to GHz) in real-time to spot and access momentarily idle channels. Unfortunately, real-time wideband spectrum sensing is challenging since it requires high-speed analog-to-digital converters (ADCs) that can sample the signal at the Nyquist sampling rate. Such high-speed ADCs are expensive, have low bit resolution, and can consume several watts of power [4, 12, 19, 34, 58].[1] To avoid using high-speed ADCs, today's systems sequentially scan the spectrum, monitoring each narrow band for a short period of time [11,47]. As a result, they cannot continuously sense all bands in real-time and can easily miss highly dynamic and fleeting signals such as radar waveforms in the CBRS band [53].

Past work has proposed using compressive sensing or sparse Fourier transforms to sense wideband spectrum without sampling at the Nyquist rate [21, 27, 31, 41]. However, these approaches inherently rely on the assumption that the frequency spectrum is sparsely occupied. Hence, they only work in the case of underutilized spectrum where at most 5% to 10% of the frequency bands are occupied [21,58]. The goal of dynamic spectrum sharing, however, is to efficiently utilize the spectrum. Hence, wideband spectrum sensing must work even in a densely occupied spectrum in order to scale usage to many users and achieve high utilization.

In this paper, we introduce S^3 (Spectrum Sensing Spike-train), an efficient low power spectrum sensing system that can monitor the real-time occupancy of multiple frequency bands in a wide spectrum. S^3 samples the wireless spectrum below the Nyquist sampling rate using cheap, commodity, low power ADCs but does not assume that the spectrum is sparsely occupied. A key enabler of S^3 is the use of MEMS (mirco-electro-mechanical-system) acoustic resonators that can create a spike-train like filter in frequency as shown in Fig 1. The MEMS filter processes the signal in the acoustic domain using carefully designed piezoelectric resonators with an assortment of equally spaced resonance frequencies. The resonators will pass the signals in these resonance frequencies and filter out the rest before converting the signal back to the RF domain. This creates an RF filter with very narrow, sharp, and periodic passbands across a wideband spectrum.

The spike-train filter enables S^3 to sample the spectrum in

[1] In fact, the power consumption of spectrum sensors is dictated by the ADC sampling rate as shown in [58]. Hence, we can significantly improve the energy efficiency by reducing the sampling rate.

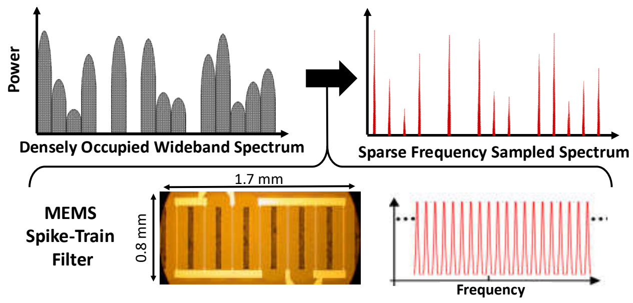

Power
Densely Occupied Wideband Spectrum
Sparse Frequency Sampled Spectrum
1.7 mm
MEMS Spike-Train Filter
0.8 mm
Frequency

Figure 1: Filtering using MEMS acoustic resonators.

the frequency domain and monitor a small fraction of bandwidth in every band. S^3 can then tell if a band is occupied or idle by examining the sampled bandwidth in it, without the need to recover the entire band. This is like finding an available spot in a parking lot; We can tell if a spot is taken by peeking at some part of the car in it and we don't need to get close to every spot and see the entire car. Moreover, even if the wideband spectrum is densely occupied, the filter makes the spectrum significantly sparser as shown in Fig. 1. This enables S^3 to sample the signal below the Nyquist sampling rate and still recover the channel occupancies.

Translating S^3 into a practical system, however, requires addressing two key challenges. First, we need an algorithm that can accurately and efficiently reconstruct the spectrum occupancy. To address this, S^3 builds on past work in sparse recovery theory but differs from it in key aspects. In particular, compressive sensing algorithms require randomly sampling the time signal and cannot simply be implemented using low-speed ADCs [27,59]. Sparse Fourier transform algorithms, on the other hand, can be implemented using low-speed ADCs, but they assume that the sparsely occupied bands are randomly distributed in the frequency spectrum [17,21]. The MEMS filter creates a sparse spectrum that is highly periodic and far from random. For such sparsity patterns, sparse Fourier transform algorithms are highly sub-optimal.

S^3 aims to achieve the best of both worlds, i.e. no random sampling in time and no assumption of random distribution of occupied frequencies. To this end, S^3 leverages the uniquely structured sparsity pattern created by the filter to overcome the above challenges. The filter restricts the occupied frequencies to known locations in the spectrum, which significantly reduces the search space. It also allows us to optimize the sub-Nyquist sampling rate. In particular, optimal recovery can be achieved by choosing a sub-sampling factor that is co-prime to the number of spikes in the filter, as we show in section 5.

The second challenge is that in practice the MEMS resonators do not create an ideal spike-train. The spikes are not extremely narrow and have a small passband bandwidth which reduces the sparsity. Moreover, the separation between the spikes is not perfectly equal, and the spikes themselves are not identical. To address this, S^3 leverages the fact that different filters that are manufactured using the same process exhibit a very similar non-ideal spike-train, as we show in Sec. 6. Hence, the filter frequency response can be measured once and incorporated into the design of S^3. Specifically, we co-design the hardware and recovery algorithm of S^3 to account for the filter non-idealities and optimize its performance.

Evaluation: We had fabricated a chip-scale MEMS filter, shown in Fig. 1, which we leveraged to build a working prototype of S^3. The prototype can sense channel occupancies over a 418 MHz spectrum in real-time while sampling $8.5\times$ below the Nyquist rate. The prototype uses two cheap, low power, off-the-shelf ADCs that sample around 50 MS/s ($\approx 1/17$ of the Nyquist rate). We extensively evaluate the performance of S^3 using a wireless testbed with 20 software defined radios that can occupy the entire 418 MHz spectrum at various power levels. Our results show that S^3 can accurately detect occupied channels. Even when the spectrum is as crowded as 90% occupied, S^3 achieves a false positive rate of 0.02 and a false negative rate of 0.0047. We also compare S^3 to state-of-the-art prior work like BigBand [21] and SweepSense [20] and demonstrate $5-10\times$ lower error rate for non-sparse spectrum. Furthermore, we show that S^3 can recover the wireless spectrum by performing outdoor and indoor measurements at various frequencies using a spectrum analyzer as the ground truth. Finally, we extend S^3 to not only detect the occupancy of the bands but also capture the power spectral density of the spectrum by quickly sweeping the center frequency for 22 MHz to cover the separation between the spikes.

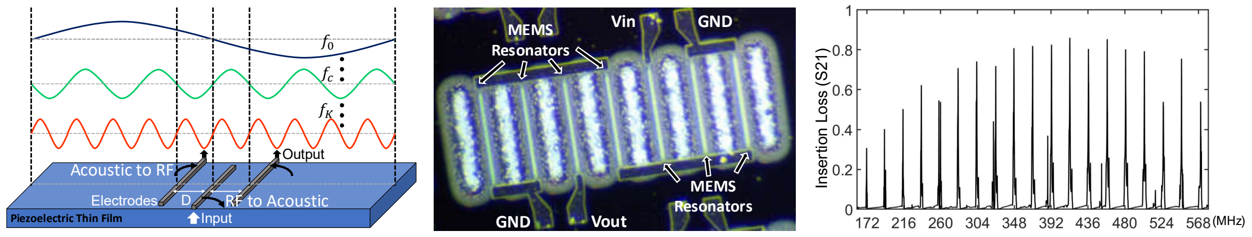
Contributions: This paper has the following contributions:

- The paper bridges the latest advances in overtone MEMS acoustic resonators to RF spectrum sensing by leveraging spike-train filters to enable cheap and low power real-time wideband sensing of a densely occupied spectrum.
- The paper presents a novel sparse recovery algorithm that leverages the uniquely structured spectrum sparsity to efficiently recover a spectrum sampled significantly below the Nyquist sampling rate.
- The paper builds a prototype using commodity low-power components and evaluates its performance in a real testbed.

2 Background

In this section, we provide a brief background on wideband spectrum sensing using sub-Nyquist sampling. Further related work and background on spectrum sensing can be found in section 9.

This paper builds on past work that senses wideband spectrum without sampling at the Nyquist rate using compressive sensing [27, 31, 32, 39, 51, 58–60] or sparse Fourier transform algorithms [21, 43]. However, these approaches only work when the spectrum is underutilized and sparsely occupied which defeats the purpose of efficiently utilizing the spectrum. Furthermore, compressive sensing needs random



(a) Acoustic Resonator Diagram (b) Microscope Image of MEMS Filter (c) Spike-Train Filter Frequency Response

Figure 2: Spike-Train filter using MEMS acoustic resonators.

sampling [27, 58, 59], and as a result, requires custom hardware designs that can consume as much power as an ADC that samples at the Nyquist rate [1, 2]. Sparse Fourier transform algorithms do not necessarily require random sampling but must assume that the sparsely occupied bands are randomly distributed in the frequency spectrum to accurately recover the frequencies [17, 21].

BigBand [21] leverages sparse Fourier transform and uses co-prime sampling to acquire a sparse bandwidth while sampling 6× below the Nyquist rate. However, it only works up to 10% spectrum occupancy at which point it cannot recover the status of more than 14% of the spectrum. An extension, D-BigBand [43] can sense dense spectrum by considering the differential changes in occupancy. However, it assumes that the spectrum occupancy is mostly static with very few changes over time. Hence, it would not work for dynamic spectrum sharing where users sense and opportunistically transmit whenever they find an idle channel. [30] also attempts to extend BigBand to dense spectrum but requires sampling the signal first at the Nyquist rate in order to permute the samples and filter the signal before further sub-sampling it below Nyquist. S^3, on the other hand, can sense dense spectrum without the need for Nyquist sampling or random sampling. It also makes no assumptions on the changes in occupancy or the distribution of occupied bands across the spectrum.

3 Spike-Train MEMS Filter

Our work builds on recent advances in overtone MEMS RF filters [18, 40]. The MEMS filters convert RF signals into acoustic vibrations through the piezoelectric effect, then filter and process the signal in the acoustic domain before converting it back to the RF domain. Such filters can be further integrated with ICs to form an RF front-end solution, operating between a few MHz and 30 GHz for mobile and IoT devices. To this end, past work on MEMS RF filters optimize for a filter with a single passband [45, 64]. In contrast, the MEMS filter used by S^3 leverages overtone resonators that have an assortment of equally spaced resonance frequencies resulting in a spike train in the frequency domain. S^3 uses some of the very first spike-train MEMS filters which we had designed and fabricated [28, 29] to enable low power real-time wideband spectrum sensing of densely occupied spectrum.

To better understand how the MEMS filter works, consider the diagram of a MEMS acoustic resonator shown in Fig. 2(a). This resonator is commonly referred to as a LOBAR (Lateral Overtone Bulk Acoustic Resonator). The device consists of three electrodes on the top of a thin film made of the piezoelectric material $LiNbO_3$. RF signals come through the middle electrode and can be efficiently converted into acoustic waves through the piezoelectric effect, as long as their frequencies match the resonances of the film and are supported by the electrode design. Otherwise, the signals are reflected back and the frequencies are filtered out.

The resonance frequencies are determined by:[2]

(1) The width of the film: the film supports resonance frequencies for which acoustic wave vanishes at the edges of the film [8] i.e., the sine wave crosses zero at the edges as shown in Fig. 2(a). This condition is satisfied when the width of the film W is an integer (k) multiple of half a wavelength ($W = k\lambda/2$). Since $f = v/\lambda$, where v is the acoustic velocity in the piezoelectric material, the MEMS resonator will resonate at frequencies: $f_k = kv/2W$.

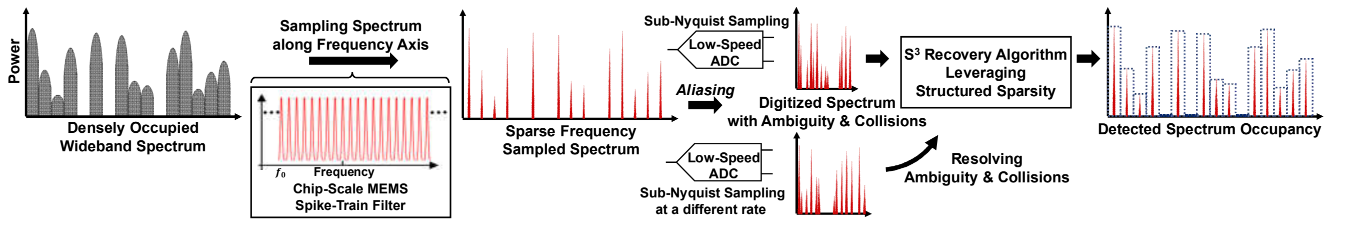
(2) The placement of electrodes: the filter will operate at center frequency f_c determined by the distance D between the electrodes: $f_c = v/2D$. Furthermore, for an odd number of electrodes, only acoustic waves that cross zero at the middle electrode, as shown in Fig. 2(a), will resonate.

Thus, the resonance frequencies will be the f_ks around f_c where k is even. This leads to a filter with center frequency f_c and a spike train where the spacing between the spikes is $\Delta f = v/W$. By modifying the width of the film and the position of the electrodes, we can modify Δf and f_c to control the frequency of the spikes in the filter.

The bandwidth or frequency span of the filter around f_c is determined by the electrodes where their RF-to-acoustic conversion efficiency degrades for resonance frequencies far from f_c, resulting in higher loss in spikes far from f_c. Adding more electrodes reduces the loss in spikes near f_c but narrows down the frequency span. We found that a three electrodes give the widest span with minimal loss of at most 2 dB.

Finally, to further enable a filter with very sharp and narrow

[2]The filter design actually involves a 4-way trade-off between (1) the frequency span, (2) the spacing between adjacent spikes, (3) the insertion loss in the spikes, and (4) the out-of-band rejection. For simplicity, we focus on the resonance frequencies design, and the trade-off between the frequency span and insertion loss. More details on the MEMS filter design trade-offs are discussed in [29].



Power
Densely Occupied Wideband Spectrum
Sampling Spectrum along Frequency Axis
f_0 Frequency
Chip-Scale MEMS Spike-Train Filter
Sparse Frequency Sampled Spectrum
Sub-Nyquist Sampling
Low-Speed ADC
Aliasing
Digitized Spectrum with Ambiguity & Collisions
Low-Speed ADC
Sub-Nyquist Sampling at a different rate
S³ Recovery Algorithm Leveraging Structured Sparsity
Resolving Ambiguity & Collisions
Detected Spectrum Occupancy

Figure 3: S^3 **System Pipeline:** S^3 samples the wideband spectrum along the frequency axis using the spike-train filter. The output spectrum is sub-sampled with low-speed ADC. S^3 leverages the structured sparsity in the filtered spectrum to resolve ambiguity and collisions due to aliasing and recovers spectrum occupancy.

spikes, we combine 7 of these MEMS resonators in a ladder filter topology [24]. Fig 2(b) shows an image of the filter under the Microscope. The frequency response of the filter is shown in Fig. 2(c). It has 19 periodic spikes with lowest loss between 161 and 579 MHz spanning a wide bandwidth of 418 MHz. The spacing between spikes is $\Delta f = 22$ MHz and the width of each spike is around $1 \sim 1.5$ MHz. Such an RF spike-train filter is the first of its kind. It presents a unique opportunity for processing wideband spectrum, which S^3 leverages towards efficient low power wideband spectrum sensing. It is also worth noting that the filter is passive and do not consume any power.

4 S^3 Overview

S^3 leverages the MEMS spike-train filters to sense wideband spectrum while sampling below the Nyquist rate. Fig. 3 illustrates an overview of the system pipeline. The received wideband spectrum is passed through the MEMS spike-train filter which samples the bands in the spectrum along the frequency axis. Specifically, the filter passes signals in frequencies aligned with the spikes and suppresses all the rest of the frequency components in the spectrum as shown in Fig. 3. The output of the filter is a sparse spectrum that preserves a small fraction of each band which we can use to monitor the occupancy of the band. Since the output spectrum is sparse, we can sample it below the Nyquist rate and still recover the occupancy information efficiently.

S^3 uses low-speed ADCs to sub-sample the signal. However, sampling below the Nyquist rate results in *"aliasing"* in the frequency domain i.e., multiple frequencies across the wide spectrum will alias (map) to the same frequency. Aliasing can lead to ambiguity and collisions, which prevent us from distinguishing frequencies that are occupied from those that are not. S^3 leverages the uniquely structured sparsity at the output of the spike-train filter to resolve such ambiguity and collisions and recover the spectrum occupancy. Ideally, one ADC is sufficient as we prove in section 5. However, due to practical limitations and imperfections in the spike-train filter, S^3 must use two ADCs sampling at different rates to accurately resolve ambiguity and collisions. We co-design the hardware and recovery algorithm to optimize the ADC sampling rates while accounting for the non-idealities of the spike-train filter as we describe in detail in section 6.

5 S^3 Recovery Algorithm

In this section, we describe S^3 recovery algorithm assuming an ideal spike-train filter. In later sections, we extend S^3 to deal with practical limitations.

Ideally, the spike-train filter will have equally spaced, very narrow and sharp spikes that can be approximated as an impulse train.[3] The frequency response of such a filter can be modeled as:

$$G(f) = \sum_{k}^{K} \delta(f - k\Delta f - f_0) \tag{1}$$

where K is the number of spikes, Δf is the spacing between spikes, and f_0 is the frequency of the first spike as shown in Fig. 3. Hence, the filter covers a spectrum bandwidth of $BW = \Delta f \times K$.

Let $x(t)$ be the input wideband signal in time domain and $X(f)$ be its non-sparse frequency representation whose bandwidth is also BW. After passing $x(t)$ through the spike-train filter, we get the signal $\tilde{x}(t)$ whose frequency spectrum is:

$$\tilde{X}(f) = X(f)G(f) = \sum_{k}^{K} A_k \delta(f - k\Delta f - f_0) \tag{2}$$

where $A_k = X(k\Delta f + f_0)$. $\tilde{X}(f)$ is at most K sparse i.e., it has at most K large frequency coefficients. Our goal is to recover these K coefficients A_k and estimate their power to detect the occupancy of the band around the frequency $f_0 + k\Delta f$.

S^3 samples the signal $\tilde{x}(t)$ using a low-speed ADC that samples at a rate $R = BW/P$ where P is an integer corresponding to the subsampling factor.[4] The sampling rate R is chosen such that $K \leq R \ll BW$. Let $y(t)$ be the sampled signal i.e., $y(t) = \tilde{x}(P \times t)$, and let $Y(f)$ be the Fourier transform of $y(t)$. Then, $Y(f)$ is an aliased version of $\tilde{X}(f)$:

$$Y(f) = \sum_{i=0}^{P-1} \tilde{X}(f + iR) \tag{3}$$

$Y(f)$ will cover a narrow bandwidth equal to R where frequencies in $\tilde{X}(f)$ that are equally spaced by R alias and sum together in the same frequency bin in $Y(f)$. Hence, once S^3 detects power in a frequency bin $Y(f)$, it knows that this power

[3] We can approximate the spikes as impulses if the width of the spike $\ll 1/T$ where T is the time window over which we sample the signal.

[4] Note that for simplicity, we have assumed that the ADC takes complex samples of the signal i.e., there are two ADCs sampling the I and Q of the wireless signal. We will relax this assumption in the following section.

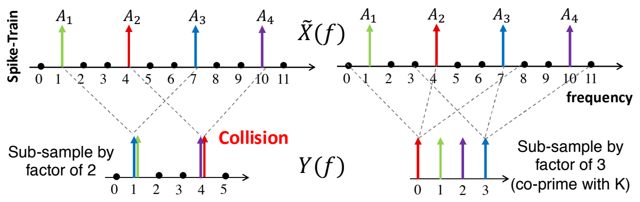


Figure 4: Co-prime sub-sampling factor avoids frequency collisions: Sub-sampling the spike train by 2 results in collisions between the spikes, but sub-sampling by 3 can avoid collisions because it is co-prime with K=4.

could have come from P different candidate frequencies in $\tilde{X}(f)$. Fig. 4 shows an example where if we sub-sample the signal by a factor $P = 2$, then every two equally spaced frequencies in $\tilde{X}(f)$ map to one value in $Y(f)$. Since $\tilde{X}(f)$ only has power in K coefficients A_k corresponding to the spikes of the filter, S^3 can easily eliminate a lot of candidates. Ideally, we want these coefficients to map to different bins. In this case, the bin value will be the same as the coefficient A_k which we can immediately estimate. However, if two coefficients A_{k_1} and A_{k_2} collide in the same bin as shown in Fig. 4, it will not be possible for S^3 to distinguish and estimate them.

S^3 can choose the sampling rate R in a manner that guarantees that no two coefficients collide. In particular, if the sub-sampling factor P and the number of spikes K are co-prime, then we can guarantee that none of K coefficients collide in the same bin and become indistinguishable. To see this, consider the example shown in Fig 4 where we have $K = 4$ spikes with coefficients A_1 to A_4 in the filtered spectrum $\tilde{X}(f)$. When we sub-sample by a factor of 2 below Nyquist, there will be collisions between A_1 and A_3, as well as A_2 and A_4. However, when we sub-sample by a factor of 3 below Nyquist, none of the coefficients collide, because the sub-sampling factor $P = 3$ and the number of spikes $K = 4$ are co-prime.

It is worth noting here that even though $P = 3$ uses a lower sampling rate than $P = 2$, increasing the sampling rate in this case results in more collisions. This is in contrast to past work on sub-Nyquist sampling [17,21] where higher sampling rates reduce collisions as the coefficients are assumed to be randomly distributed in the spectrum. Unlike past work, the structured sparsity of our spectrum requires carefully selecting the sampling rate to ensure that all coefficients can easily and immediately be recovered.

The below lemma, theoretically proves that if P and K are co-prime, then none of the coefficients will collide.

Lemma 5.1. *Given K, P are co-prime integers, let f_i and f_j be the frequencies of any two spikes in the spike train filter i.e. $f_i = k_i\Delta f + f_0$ and $f_j = k_j\Delta f + f_0$ such that $0 \le k_i, k_j < K$. Then, for all $f_i \neq f_j$, we have $f_i \neq f_j$ mod R.*

Proof. Assume there exist an $f_i \neq f_j$ such that the coefficients collide i.e., $f_i = f_j \bmod R$. Note that by definition of the spike train, we also have $f_i = f_j \bmod \Delta f$. Consequently, f_i and f_j are equal modulo the least common multiple: $\text{LCM}(R, \Delta f) = \text{LCM}(BW/P, BW/K) = BW$, since K and P are co-prime.

Algorithm 1 S^3 Sensing with an Ideal Spike-Train Filter

```
Input: $x(t)$
$B_k \leftarrow$ Band around frequency $f = k\Delta f + f_0$
$\tilde{x}(t) = g(t) \circledast x(t)$          ▷ Filter $\tilde{X}(f) = X(f)G(f)$
$y(t) = \tilde{x}(P \times t)$                  ▷ Sub-Nyquist Sample
$Y(f) = \text{FFT}(y(t))$
$A_k = Y\left((k\Delta f + f_0) \bmod R\right)$
if $E\left[|A_k|^2\right] > \sigma^2$ then
    $B_k$ is occupied
else
    $B_k$ is empty
```

Thus, $f_i = f_j \bmod BW$ which is a contradiction since BW is the entire bandwidth and we are given that $f_i \neq f_j$. Hence, by contradiction, for all $f_i \neq f_j$, we have $f_i \neq f_j \bmod R$ and none of the K coefficients collide. □

Given that we can choose a sampling rate that results in no collisions, we can easily recover the coefficients A_k as follows. We can compute $Y(f)$ by taking an FFT of $y(t)$ and for $0 \le k < K$, we directly set $A_k = Y((k\Delta f + f_0) \bmod R)$. We then apply an energy detector on A_k to obtain the occupancy of the band around the frequency $k\Delta f + f_0$. If $|A_k|^2$ is above the noise floor, then the band is occupied, otherwise, it is empty. A pseudocode for the overall sensing of S^3 with an ideal spike-train filter is shown in Alg. 1.

Next, we prove the below theorem about the correctness and the computational complexity of the algorithm.

Theorem 5.2. *Assuming a signal SNR > 0 dB for each occupied band, the system correctly recovers the occupancy of the bands using $O(K)$ samples and $O(K \log K)$ computations which is optimal.*

Proof. We will prove the above statement for the case where the entire spectrum is occupied. We can compute the the signal power of the filtered and sub-sampled signal as:

$$E\left[\|\tilde{Y}(f)\|_2^2\right] = E\left[\sum_{f=0}^{R-1} |Y(f)|^2\right] = E\left[\sum_{f=0}^{R-1}\sum_{i=0}^{P-1} |\tilde{X}(f+iR)|^2\right]$$
$$= E\left[\sum_{k=1}^{K} |\tilde{X}(k\Delta f + f_0)|^2\right] = E\left[\sum_{k=1}^{K} |A_k|^2\right] \ge KE\left[\min_k |A_k|^2\right] \tag{4}$$

Let σ^2 be the noise power per frequency. Since the spike-train filter suppresses the noise outside the spikes, the remaining noise in the signal is $K\sigma^2$. Hence, the SNR of the filtered and sub-sampled signal is:

$$SNR = \frac{E\left[\|\tilde{Y}(f)\|_2^2\right]}{K\sigma^2} \ge \frac{E\left[\min_k |A_k|^2\right]}{\sigma^2} > 1 \tag{5}$$

Thus, as long as the received signal is above the noise floor i.e. $SNR > 1$ (0 dB), filtering and sub-sampling will not increase the noise floor and the occupancy of the band can be detected

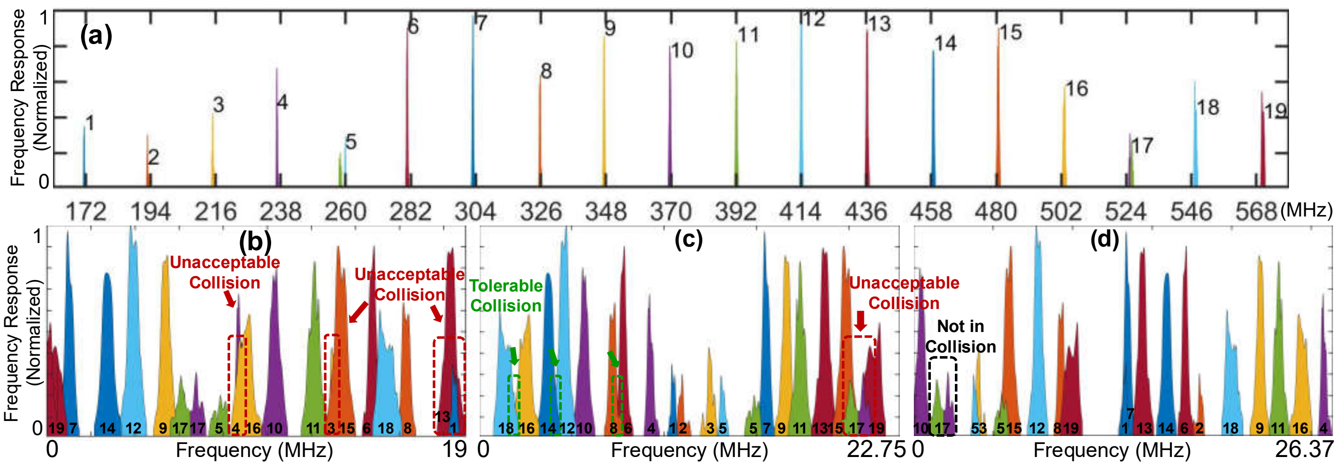


Figure 5: Aliasing of the spike-train filter at different sub-Nyquist sampling rates: (a) Locations of the 19 spikes on the frequency axis (b)Aliasing of the spikes at 38 MS/s (c) Aliasing of the spikes at 45.5 MS/s (b) Aliasing of the spikes at 52.74 MS/s.

correctly. Now, the algorithm samples at rate $R = O(K)$, takes an FFT of size $O(R)$ and then performs $O(R)$ computations. Hence, it requires $O(K \log K)$ computations and $O(K)$ sample, which is optimal. The algorithm is also deterministic, unlike compressive sensing and sparse Fourier transform algorithms which are randomized. □

6 S^3 with Practical Limitations

As mentioned earlier, the MEMS spike-train filter is non-ideal i.e., the spikes have some width as can be seen from Fig. 6. Although the ~ 1.5 MHz bandwidth is narrow compared to the channel bandwidth, it is still significant. Moreover, the spikes are neither identical nor perfectly equally spaced. In fact, they differ in magnitudes, bandwidths, and shapes. As a result, if we simply pick a sub-sampling factor P that is co-prime to the number of spikes K, there could be many collisions among the wide spikes. Figure 5 shows how the 19 spikes of our spike-train filter alias after sub-sampling. Figure 5(a) shows the spikes in the original wideband spectrum, while Fig. 5(b-d) show the aliasing of the spikes when sub-sampled at three different sampling rates. First, we choose the sampling rate to be 38 MS/s, because the resulting sub-sampling factor $P = 11$ is co-prime to $K = 19$. However, the aliased spectrum ends up with many collisions, as shown in Fig. 5(b). This suggests that the derived optimum no longer holds due to the practical limitations of the filter.

Fortunately, different filters that are manufactured through the same process exhibit a very similar spike train. Figure 6 compares the measured frequency responses of three spike-train filters we fabricated. We zoom into two spikes, otherwise the differences are very hard to spot. As one can see, the filters are almost identical. Hence, we can measure the frequency response of one spike-train filter and use it for the others.

Knowing the filter frequency response, we run an optimiza-

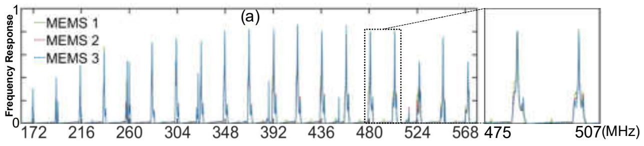


Figure 6: Measured frequency responses of three fabricated MEMS spike-train filters using the same process.

tion problem to find a sampling rate that has as little collisions as possible. Ideally, this sampling rate should separate all the wide spikes after aliasing and prevent them from overlapping with one another. If a collision is unavoidable, we want it to only occur at the boundaries of the spikes, rather than having two wide spikes fully overlap. For example, as shown in Fig. 5(b), the collisions marked in red are unacceptable, because most of a spike's frequencies experience collision. In contrast, the collisions marked in green in Fig. 5(c) are tolerable, because only the boundaries of two spikes collide. Because we can simulate and compare the aliasing at different sampling rates offline, the optimization problem, in fact, can exhaustively search for all possible sampling rates.

Another practical aspect is that in the real system, we only sample real signals and not complex in order to reduce complexity. Since the signal is real, the frequency representation is symmetric around the y-axis. Hence, with a sampling rate of $R = 38$ MS/s, the wideband spectrum actually aliases to a bandwidth of $R/2 = 19\ MHz$. Formally, if the original frequency of a spike is $f_{spike} = k\frac{R}{2} + b$, where $b < R/2$, then the aliasing frequency f_{alias} of the spike can be found through the following equation:

$$f_{alias} = \begin{cases} b & \text{if k is even} \\ \frac{R}{2} - b & \text{if k is odd} \end{cases} \tag{6}$$

In our specific case, we find 45.5 MS/s to be a really good sampling rate. As can be seen from Fig. 5(c), it spreads out the aliased frequencies of the wide spikes to 1.5, 2.6, 3.8, 4.6,

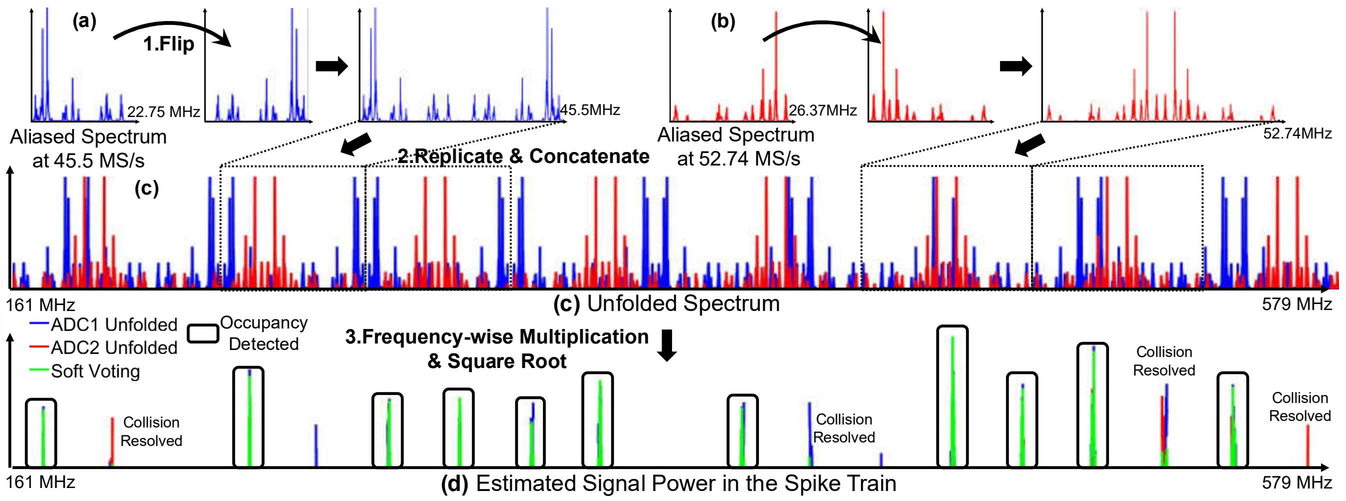


Figure 7: Practical S^3 recovery algorithm through voting: S^3 unfolds two aliased versions of the filtered spectrum to get a vote for the frequency components in the spike train. It then combines votes from the two different sampling rates to estimate the signal power in the spikes and the spectrum occupancy.

5.7, 7.3 MHz, etc. Therefore, most collisions only occur at the boundaries of the wide spikes. However, it still cannot avoid all unacceptable collisions. In fact, it's likely that no sampling rate can. For example, our spike-train filter has a unique 17th spike that is composed of two very close spikes. When sampling at 45.5 MS/s, these two small spikes completely overlap with the 15th and 19th spikes respectively. Therefore, when spike 15 and 19 are both occupied, we might falsely classify the spike 17 as occupied.

To resolve such unavoidable collisions, we leverage another sampling rate that provides us with a different set of aliasing frequencies for the spikes. We pick the second sampling rate in a way that any two spikes colliding at 45.5 MS/s do not collide again. To this end, we find a good sampling rate of 52.74 MS/s, and the resulting aliased frequencies of the spike train is shown in Fig. 5(d). One can see that the two parts of spike 17 do not collide with any other spikes at 52.74 MS/s. Thus, as long as we observe no power on frequencies corresponding to spike 17 at 52.74 MS/s, we will classify spike 17 as empty. Hence, by leveraging such incoherence between the two sampling rates, we can further resolve unavoidable frequency collisions and correctly identify the empty bands.

Using the two sub-Nyquist sampled spectra, S^3 recovers signal power in each spike, and then identifies the occupancy of the corresponding band. We leverage the two sampling rates through a soft voting scheme. The idea is that given an aliased spectrum and the sampling rate, we know all the possible original frequencies that correspond to the aliased frequencies. Hence, each aliased spectrum provides a vote for the source frequencies of the non-empty spectral components. Moreover, the non-empty frequencies on the original spectrum are also constrained to the spike-train frequencies. Therefore, when the two sampling rates vote for the same frequency that also falls in a spike, the frequency is very likely to be the true source frequency on the wideband spectrum.

Consider the two aliased versions shown in Fig. 7(a,b), where 11 out of the 19 bands are occupied and the other 8 bands are empty. Now we use them to vote where the non-empty frequency components come from. According to Eqn. 6, aliasing folds the wideband spectrum on to the bandwidth of $\frac{R}{2}$. Therefore, we can vote on all the possible source frequencies by unfolding the aliased spectrum. We accomplish this goal in the following three steps:

- **1. Unfold - Flip**: First, we flip the aliased spectrum $Y(f)$ with a bandwidth of $\frac{R}{2}$ to get the bandwidth between $\frac{R}{2}$ and R, as it equals to $Y(\frac{R}{2}-f)$ according to Eqn. 6.
- **2. Unfold - Replicate**: Then we replicate and concatenate the resulting spectrum from 0 Hz to R, and we get a vote for all frequencies in the frequency range of the spike train as shown in Fig. 7(c).
- **3. Soft Voting**: Finally, we combine the votes of the two sampling rates, where we only consider the frequencies within the spikes. This is done by multiplying the two votes on every frequency and taking a square-root. As a result, the non-empty frequencies that are voted by both sampling rates are amplified. In contrast, the frequencies where the two sampling rates vote differently will be attenuated as shown in Fig. 7(d).

After unfolding the aliased spectra and recovering the filtered spectrum through voting, we calculate the average signal power in each spike by summing up the voting results and divide it by the spike width. Additionally, we also estimate the average signal power in the spikes using the unfolded spectrum at each sampling rate separately. We classify a band as occupied if all three power estimations in the corresponding spike exceed a pre-selected power threshold. This power threshold is selected based on the noise floor, which is measured when all bands are empty. By using all three estimates, we add hard voting on top of the soft voting which adds more robustness to the occupancy detection.

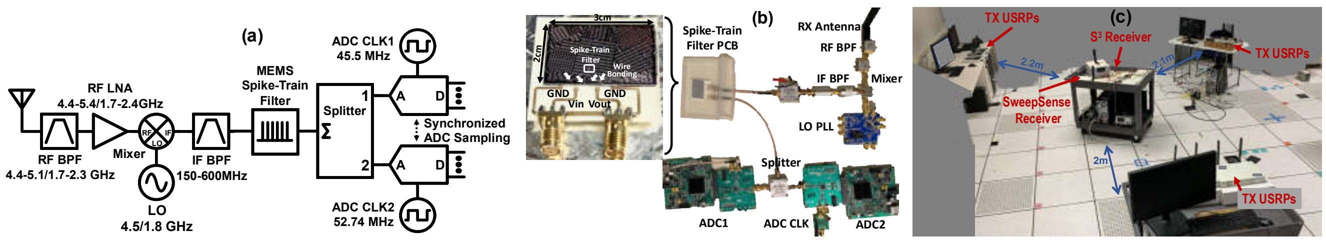


Figure 8: S^3 **Prototype System and Evaluation Testbed**: (a) S^3 Basic prototype circuit diagram (b) S^3 basic prototype circuit (c) Evaluation testbed.

7 Implementation

A. Basic Prototype: We have built a basic prototype of an S^3 spectrum sensor by combining our MEMS spike-train filter with commodity, off-the-shelf, low-power components. Figure 8(a) shows the circuit diagram of this basic prototype, and the actual prototype is demonstrated in Fig. 8(b). The signal is received through a broadband receiver. It is bandpass filtered and amplified before down-conversion to an intermediate frequency (IF) between 150 and 600 MHz. The IF signal is bandpass filtered and passed through the spike-train filter. It is then split and sampled by the two synchronized ADCs.

We wire-bond the MEMS spike-train filter onto a gold-plated PCB (printed circuit board) as shown in Fig. 8(b). We use K&S 4523A Wedge Bonder and 25 μm Aluminum wire. We use two Anolog Devices LTC2261-14 14-bit ADCs to sample the output of the spike-train filter. This ADC features an 800 MHz wideband input analog bandwidth and low power consumption of 89 mW. The ADC sampling is timed through an external square-wave clock signal. We use the DC1370A ADC evaluation board and the DC890 data acquisition controller to control the ADC sampling through the open-sourced LinearLab Tools Python API. We bypass the input low pass anti-aliasing filter on the ADC evaluation board to maintain the wide analog bandwidth.

B. Extending the Prototype: The above basic prototype using only one spike-train filter can be extended to sense spectra with different center frequency, bandwidth, and channel allocation. Moreover, system level parallelism introduces another degree of freedom and allow us to break the fixed design trade-offs at the filter level.

- **Different Spectrum:** By changing the LO frequency as well as the RF bandpass filter and LNA, we can sense different frequency ranges. In our evaluation, we test at center frequencies of 2.1, 2.4, 4.9, and 5.7 GHz.
- **Larger Bandwidth:** The current spike-train filter supports a bandwidth of 418 MHz. We can extend S^3 to larger bandwidth by either using two sensors and configuring them to sense adjacent spectra or by using two MEMS filters in parallel channels before combining the signals and sampling it as we describe in more detail in appendix A.
- **Narrower Bands:** The spikes in the spike-train filter are separated by 22 MHz. Hence, narrowband signals (< 20 MHz) that are not aligned with the spikes might be filtered out. To address this, we can combine frequency domain sampling with LO frequency sweeping over 22 MHz to capture and sense all the frequencies in the spectrum as shown in our results in section 8. Alternatively, we can design a MEMS filter with narrower spacing as explained in section 3 or use two MEMS filters and set the center frequency to be slightly different as we describe in more detail in appendix A.

C. Testbed: We evaluate the performance on S^3 both through controlled experiments in an indoor wireless testbed as well as through measurements of ambient transmissions outdoors and indoors. The wireless testbed allows us to control the spectrum sparsity, how fast the occupancy changes for different bands, the type of signals transmitted, and the power of various transmissions. It also allows us to know the groundtruth band occupancy in order to evaluate the performance of S^3.

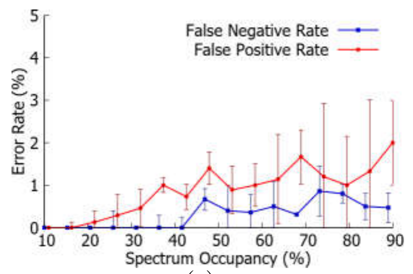
The testbed, shown in Fig. 8(c), can create a 418 MHz spectrum with various occupancy status at different frequencies. It consists of 19 N210 USRP software-defined radios, each transmitting on a 25 MHz bandwidth. While the USRPs are not very far from each other, we vary their transmission power randomly by up to 10 dB and observe received signal SNR that varies by up to 20 dB between different USRP transmitters. To avoid interference from ambient 2.4 and 5 GHz ISM band signals, we conducted experiments in two 418 MHz-wide spectra: 4.73 to 5.15 GHz and 1.93 to 2.35 GHz, each divided into nineteen 22 MHz bands. We vary the spectrum occupancy from 10% to 90%. We also vary the type of modulation being used. We test with single carrier BPSK and QAM as well as OFDM signals. Note that single carrier modulation has a non-flat power spectral density and significantly more leakage, so it results in higher false positive rates as we show in section 8. We also leverage the testbed to compare S^3 with state-of-the-art sensing systems as our baselines. We ran over 5000 experiments with different configuration of occupancy, power, modulation, etc.

8 Results

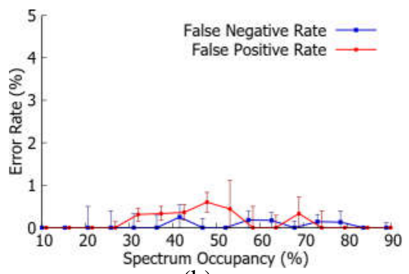
In this section, we present our main evaluation results along with a few microbenchmarks that provide insights into the performance of S^3 in various spectra.

Evaluation Metrics: We evaluate S^3 using following metrics:

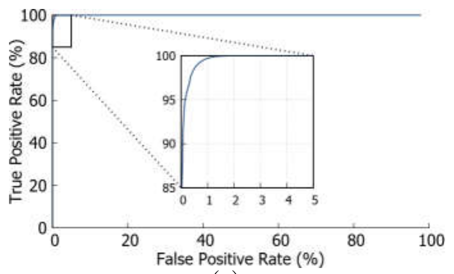
- *False Positive Rate (FPR)*: Percentage of empty bands that S^3 incorrectly reports as occupied.



(a)



(b)



(c)

Figure 9: False positives and negatives as a function of spectrum occupancy: The figure shows the false positive rate (FPR) and false negative rate (FNR) of S^3 as the spectrum occupancy increases when: (a) Modulation schemes are randomly picked by transmitters. (b) Transmitters only use OFDM modulation. (c) shows the receiver operating characteristic (ROC) curve of S^3 when the modulation schemes are randomly picked.

- *False Negative Rate (FNR)*: Percentage of occupied bands that S^3 incorrectly reports as empty.
- *True Positive Rate (TNR)*: Percentage of occupied bands that S^3 correctly reports as occupied.

A. Sensing Densely Occupied Spectrum:

Fig. 9 shows S^3's error rate in detecting occupied bands as we vary the total occupancy of the spectrum between 10% and 90%. Fig. 9(a) shows the results when the transmitters randomly pick a modulation scheme (e.g. single carrier BPSK, QAM, or OFDM). In this case, when the total occupancy of the spectrum is less than 30%, S^3 achieves a median false positive rate (FPR) less than 0.5% and a median false negative rate (FNR) of 0%. As the total occupancy increases and the spectrum becomes more crowded, the FPR and FNR gradually increase. However, even when the spectrum is extremely crowded ($\sim$ 90% occupied), S^3 can still achieve 2% median FPR and 0.47% median FNR.

Fig. 9(b) shows the same results when the transmitters only use OFDM modulation. In this case, the FPR and FNR become even smaller at all levels of occupancy with a maximum median FPR of 0.6% and a maximum median FNR of 0.25%. This result can be attributed to two factors: (1) OFDM signals have flat power spectral densities. Therefore, signal power detected in the spike train can more accurately reflect the signal presence in the corresponding channels. (2) Single carrier modulation schemes have lower spectral efficiency and leak power outside their bands, which leads to a higher FPR as can be seen from Fig. 9(a). Finally, Fig. 9(c) shows the receiver operating characteristic (ROC) curve, which demonstrates the trade-off between false positives and false negatives as we vary the threshold for detecting occupied band.

B. Comparison with State-of-the-Art:

We compare S^3 with three baselines from prior work:

- **BigBand:** [21] leverages sparse Fourier transform and uses co-prime sampling to acquire sparse spectrum. It achieves 6× sub-sampling below the Nyquist rate, but only works when the spectrum is sparse.
- **D-BigBand:** [43] extends BigBand to sense dense spectrum by considering the differential changes in occupancy. It also achieves 6× sub-sampling, but assumes the changes in the spectrum occupancy over time are sparse.
- **SweepSense:** [20] enhances USRP software-defined radio's ability to quickly scan and sense wideband spectrum. It is able to scan 5 GHz bandwidth in 5 ms with 2×25 MS/s ADC sampling rate.

Table 1: Sum of false positives and negatives for S^3 and State-of-the-Art prior work: The table compares the sum of FPR and FNR of S^3, BigBand, D-BigBand, and SweepSense at different spectrum occupancies.

	BigBand	D-BigBand	SweepSense	S^3
10%	0.38% + 14% (unresolved)	$\sim$ 0.95%	4.88%	0.00%
50%	N/A	$\sim$ 1.75%	13.09%	1.29%
90%	N/A	$\sim$ 3%	13.76%	2.47%

Table 1 shows the sum of FPR and FNR when the total spectrum occupancy is 10%, 50%, and 90%. We compare S^3 directly to the results reported in [21] and [43], because they used custom hardware but were evaluated using the same metrics as ours. One can see that in sparse spectrum (<10% occupied) where BigBand works, BigBand has a total error rate of 0.38% but still cannot recover the status of 14% of the spectrum. In contrast, S^3 accomplishes a 0% error rate at such low spectrum occupancy and samples 8.5× below the Nyquist rate, which exhibits a 1.4× gain over BigBand.

D-BigBand is able to work in densely occupied spectrum. It has a total error rate of 0.95% and 3% when the spectrum is 50% and 90% occupied respectively. However, S^3 is able to outperform D-BigBand at all occupancy levels with a 1.2× to 1.35× gain in accuracy. Moreover, S^3 also achieves 1.4× gain in sampling rate reduction and makes no assumptions on the changes in spectrum occupancy. Therefore, unlike D-BigBand, S^3 can monitor highly dynamic spectrum, which we will demonstrate later in this section.

To compare S^3 to SweepSense, we reproduce SweepSense on a N210 USRP with a CBX daughterboard using the codes and FPGA images released by the authors. We use SweepSense to sense the spectrum generated by our testbed along with S^3. In Fig. 10, we show SweepSense's error rate in detecting occupied bands as we vary the spectrum occupancy between 10% and 90%, and when the modulation scheme is randomly picked by the transmitters. This result shows that SweepSense can work in densely occupied spectrum. When

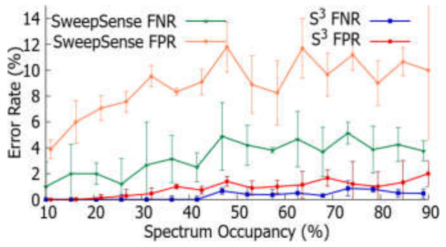


Figure 10: False positives and negatives comparison between S^3 and SweepSense.

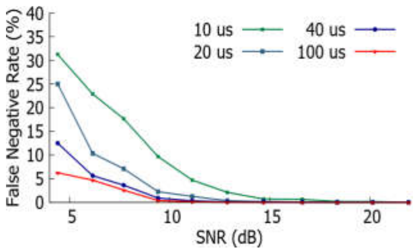


Figure 11: FNR vs SNR and sampling time.

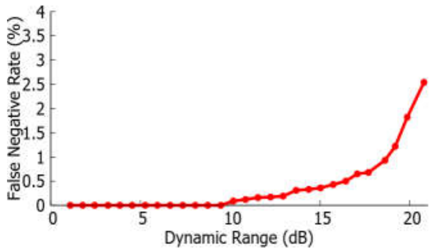


Figure 12: FNR vs dynamic range.

the spectrum is 10% occupied, SweepSense achieves an FPR of 3.88% and an FNR of 1%. As the spectrum becomes more crowded, the FPR and FNR of SweepSense increase, but they remain below 10% and 3.76% respectively even if the spectrum is 90% occupied. SweepSense's higher error rates are likely due to the fact that fast LO sweeping can smear nonempty frequency components, resulting in more leakage from the occupied bands to the adjacent bands, which increases its false positive rates. We also note that, SweepSense requires accurate phase information for the digital chirp demodulation, so it is sensitive to the IQ imbalance in the hardware, which is likely why it underperforms S^3. However, SweepSense is highly valuable as we can combine it with S^3 to capture the power spectral density as we show later in this section.

Next, we present some microbenchmarks that provide more insights into the working of S^3 and its performance.

C. Microbenchmark - Sensitivity:
To understand the ability of S^3 to detect low signal-to-noise ratio (SNR) signals, we examine the FNR of bands with different SNRs. The SNR we show is the average signal power per Hz of RX signal / noise floor. We compare four different sampling duration: 10, 20, 40, and 100 μs. The FNR is high when the SNR is low, however, this can be addressed by increasing the sampling duration. In fact, we can reduce the FNR by $5\times$ at 3dB SNR. As the SNR gets higher, the FNR goes down and down, and eventually even for shoft sampling windows, the FNR is very low ($\approx 0\%$). Note that $40 \sim 100$ μs is a short enough window to detect short transient packets and fleeting signals, as it is comparable to the DIFS duration for Wi-Fi carrier sensing (e.g. 34 or 50 μs).

D. Microbenchmark - Dynamic Range:
Here we evaluate the dynamic range of S^3, which is the ratio between the strongest and weakest signal powers S^3 can accurately detect at the same time. It reflects the ability of S^3 to detect low-power signals with the presence of much higher power signals, that would cause interference and lower the signal-to-noise-pluse-interference ratio (SINR), making low-power signals harder to be detected. In Figure 12, we compare the FNRs in experiments with different dynamic ranges. One can see that S^3 achieves very low FNR ($< 0.63\%$) when the power difference between the occupied bands is up to 15 dB. As the signal power difference becomes even larger, the FNR of S^3 increases. When the spectrum dynamic range reaches ~ 21 dB, the FNR of S^3 is 2.54%. This result shows that S^3 can accurately detect (FPN<1%) the relatively weak signals under interference from signals 19 dB stronger.[5]

E. Microbenchmark - Resolving Collisions with Voting:
We want to verify that through voting using two different sampling rates, S^3 can effectively resolve frequency collisions. To this end, we compare S^3 to baselines where we detect the spectrum occupancy using only one ADC. In Fig. 13(a), we qualitatively compare the correctness of occupancy detection on each band in 20 randomly selected experiments. It shows that when using either ADC alone, we have many false positives due to frequency collisions. However, the two ADCs exhibit false positives in different bands, because they experience frequency collisions between different spikes. As a result, through voting S^3 is able to distinguish and resolve false positives where the two sampling rates disagree with each other. Furthermore, we also quantitatively show the FPR of S^3 and baselines. As can be seen from Fig. 13(b), the FPR of S^3 is much lower than those of baselines, which suggests that our voting scheme can effectively leverage the different sampling rates to resolve frequency collisions. Note that ADC1 outperforms ADC2. This is expected because, as we discussed in section 6, 45.5 MHz is an optimized sampling rate that can spread out the spikes and minimize the frequency collisions. In contrast, the second sampling rate of 52.74 MHz is optimized to avoid having the same collisions as ADC1, so it does not work as well by itself.

F. Monitoring Dynamic Spectrum:
S^3 senses all bands in the spectrum in real-time and makes no assumptions on the changes of spectrum occupancy, so it can monitor highly dynamic spectrum with rapidly-changing occupancied bands. To evaluate this ability of S^3 we create a rapidly-changing spectrum in our testbed whose occupied bands change every 327 μs, and as a result, the total spectrum occupancy varies between 0% and 63%. We use S^3 to continuously monitor the occupancy changes in the spectrum, and output a signal power estimation and occupancy detection for every 76 μs-long frame. We show a spectrogram captured by S^3 consisting the signal power detected in every band per

[5]After wire-bonding, the spike-train filter experiences degradation in the out-of-band suppression due to the direct leakage from the input port to the output port of the PCB. Hence, the sensitivity and dynamic range of S^3 also degrade, but this issue can be resolved by better isolation in the PCB design.

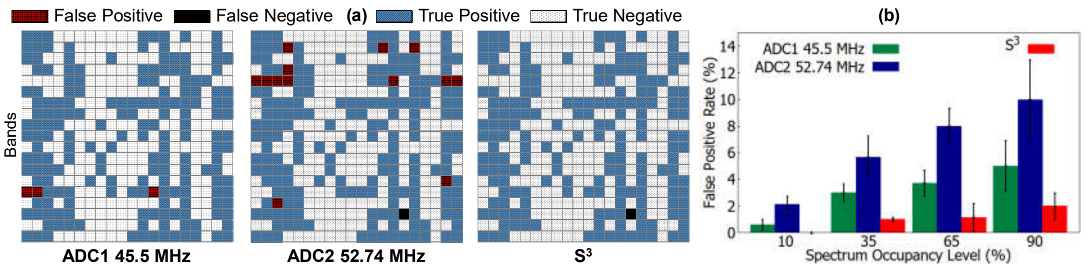


Figure 13: False positives comparison between S^3 and baselines that use single sampling rate: (a) shows the spectrum occupancy detected by S^3 and the baselines in 20 randomly selected experiments. (b) shows the false positive rate comparison as the spectrum occupancy increases.

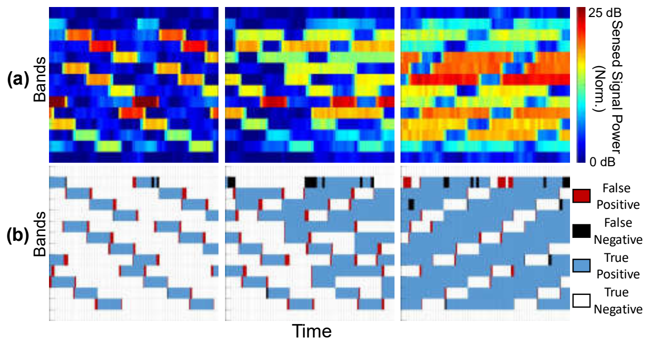


Figure 14: Monitoring rapidly-changing spectrum: The figure shows (a) spectrogram (b) spectrum occupancy captured by S^3 in real time.

frame in Fig. 14(a). Furthermore, we show the accuracy of the corresponding occupancy detection per frame in Fig. 14(b). It shows that S^3 is able to capture the occupancy of rapidly-changing spectrum with great accuracy and time precision.

G. Capturing Wideband Power Spectral Density: As we mentioned in section 7, we can sweep the LO frequency of S^3 over the 22 MHz spacing between spikes to sense all the frequencies in the spectrum. This enables S^3 to capture the power spectral density (PSD) of the entire wideband spectrum. At every LO frequency, S^3 captures signal power in the spike train and identifies the occupancy of each spike. For the occupied spikes, S^3 uses the signal power estimates in them to reconstruct the PSD at the corresponding RF frequencies. When LO sweeping finishes, all frequencies on the wideband spectrum will be reconstructed. Comparing to conventional spectrum scanners, this extended S^3 prototype only needs to sweep a much narrower frequency range. Therefore, the scanning time is much shorter.

Figure 15(a-c) shows the PSD captured by the extended S^3 prototype of spectra generated by our testbed, along with the detected spike occupancy. We use S^3 and an HP 8563E Spectrum Analyzer to monitor the 1.8 to 2.4 GHz spectrum simultaneously. As one can see, the PSDs captured by S^3 match the ground truth from the spectrum analyzer very well. Besides, we also measure PSDs of real-world spectra, both outdoors and indoors, which are shown in Figure 15(d-f). Figure 15(d) shows the spectrum between 1.8 and 2.4 GHz measured outdoor at our geographical location. It shows that S^3 is able to capture the PSD of 4G LTE signals in Band 2 and 66. In Fig. 15(e) and (f), we show the PSD of 2.4 GHz and 5 GHz Wi-Fi signals captured by S^3 respectively. One can see that Channel 1 and 11 in the 2.4 GHz band as well as four non-overlapping 20 MHz channels (Channel 116, 120, 124, and 128) from 5.57 to 5.65 GHz in the 5 GHz band are being used. Figure 15(d-f) demonstrate that the real-world PSDs captured by S^3 also closely match the spectrum analyzer ground truth. On some frequencies that S^3 classifies as empty, the spectrum analyzer shows some non-zero spectral components. However, this is expected because in our experiments, the spectrum analyzer takes the maximum over a lot more scans than S^3.

9 Related Work

In this section, we provide more related work. For further background, we refer the reader to section 2.

Spectrum sensing has been extensively studied in the past two decades [3, 5, 49]. However, most of this work focuses on narrowband sensing [6, 7, 9, 23, 25, 35, 38, 62]. In contrast, this paper focuses on wideband spectrum sensing to enable dynamic spectrum sharing of many channels. Several systems attempt to sense wideband spectrum using narrowband sensors without sequentially scanning each band [42, 46, 61]. QuickSense [61] leverages analog filters and energy detectors to hierarchically sense wide bandwidth by detecting the total signal power in groups of consecutive channels. However, when the spectrum is densely occupied, QuickSense's approach reduces to sequentially scanning the spectrum. SpecInsight [42] leverages machine learning to predict spectrum occupancy based on learned utilization patterns and optimize which channels to sense. Similarly, [46] uses time-series analysis to predict which bands are occupied. However, these systems are sensitive to training data and assume that transmissions follow predictable patterns. Spectrum sharing is based on opportunistic access and as a result is highly dynamic and unpredictable [50].

Recent work aims to enhance USRP software-defined radio's ability to sense wideband spectrum [20, 26]. SweepSense [20] enables sensing wideband spectrum by quickly sweeping the center frequency of the USRP. It is able to sweep 5 GHz bandwidth in 5 ms, which offers great potential for sensing an extremely wideband spectrum on commercial software radios. However, SweepSense requires

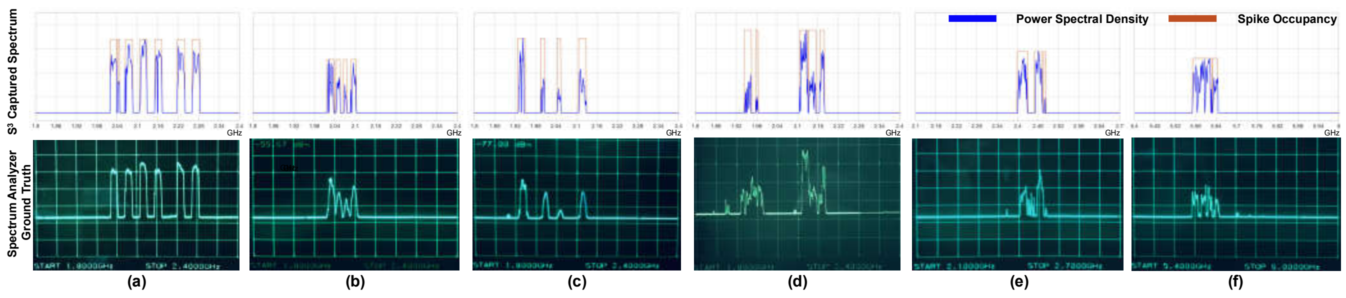


Figure 15: Wideband Power Spectral Density Capture: The figure shows the wideband power spectral density captured by the extended S^3 prototype. Spectrum (a-c) are generated by our evaluation testbed. Spectrum (d-f) are real-world spectra captured both outdoor and indoor.

accurate phase information for the digital chirp demodulation and is sensitive to the IQ imbalance. As a result, our comparison with SweepSense in section 8 shows that it can suffer from a high error rate especially when the spectrum is not sparse. SparSDR [26] reduces the backhaul and computation requirements for sensing sparse spectrum on USRPs, which offers great utility for continuously monitoring underutilized spectra but cannot scale to densely occupied spectrum.

The use of single passband MEMS filters in spectrum sensing has been studied [33, 36]. However, these techniques require an array of channel-select MEMS filters to form a reconfigurable filter bank. In contrast, S^3 only uses a single MEMS spike-train filter that consists of overtone resonators.

Our work is also related to theoretical work on co-prime sampling [54, 56, 57] and multicoset sampling [22, 55] of sparse wideband spectrum. These approaches also do not work for densely occupied spectrum. Moreover, [56, 57] require using k ADCs where k is the number of occupied frequencies. [22, 55] aim to recover the signals in each occupied band and must assume prior knowledge of which bands are occupied. In contrast, S^3 aims to recover the occupancy of each band and uses 2 ADCs irrespective of the number of occupied frequencies. S^3 is further implemented and shown to work in practice.

Sub-Nyquist sampling has been used for test equipment to reconstruct wideband periodic signals [44, 52]. However, these techniques require the signal to be periodic and repeat for a long time in order to take on samples during each period until all samples are recovered. Hence, these techniques are not applicable to real communication signals where the signal is constantly changing and carries different modulated bits.

Finally, some works aim to capture spectrum usage at large geographical and time scales through crowdsourcing [10, 37, 63]. S^3 is complementary to these works, as it enables real-time wideband occupancy detection of every single sensor with minimum data size and computational complexity.

10 Limitations

In this section, we discuss some limitations of S^3.

- The frequency-domain sampling rate and maximum sensing bandwidth is limited by the filter design trade-offs. As a result, narrowband signals (< 20 MHz) and over GHz-wide spectrum cannot be sensed using a single spike-train filter. This can be resolved by hopping the LO frequency as shown in Section 8. Alternatively, we can also use the extended architectures proposed in appendix A that combines multiple spike-train filters in parallel to break the fixed filter-level design trade-offs.
- Sub-Nyquist sampling leads to aliasing of both signals and noise, which typically lowers the signal SNR and degrades the spectrum sensor's sensitivity. To minimize the loss of SINR, we design the spike-train filters to have low insertion loss and high out-of-band suppression i.e., most of the noise is filtered out before it aliases. Moreover, instead of detecting signal power, known signals like the preambles can be leveraged to improve the sensitivity [48]. However, directly applying this technique to S^3 would require further research as the preambles might become corrupted after applying the filter.
- While S^3 can detect the occupancy of the different bands and reconstruct the power spectral density of the spectrum, it cannot recover complex I and Q samples of the signal. As results, S^3 cannot reconstruct the signal itself or decode the data in the signal.

11 Conclusion

This paper presents S^3, a new efficient real-time wideband spectrum sensing mechanism that can work in densely occupied spectrum. S^3 monitors only a small fraction of bandwidth in each band to accomplish significantly below-Nyquist sampling and hence great energy efficiency. It leverages recent advances in RF MEMS filtering solution that enables sampling the spectrum along the frequency axis. Empirical evaluation demonstrates that S^3 can accurately sense densely occupied spectrum and rapidly-changing spectrum; we also show that S^3 can be extended to capture the power spectral density of the entire spectrum. We believe S^3 can enable dynamic spectrum access and very high spectrum utilization.

Acknowledgments: We would like to thank our shepherd Dinesh Bharadia as well as the reviewers for their feedback. This work is funded in part by NSF Award 1824320.

References

[1] O. Abari, F. Chen, F. Lim, and V. Stojanović. Performance trade-offs and design limitations of analog-to-information converter front-ends. In *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5309–5312. IEEE, 2012.

[2] O. Abari, F. Lim, F. Chen, and V. Stojanović. Why analog-to-information converters suffer in high-bandwidth sparse signal applications. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 60(9):2273–2284, 2013.

[3] A. Ali and W. Hamouda. Advances on spectrum sensing for cognitive radio networks: Theory and applications. *IEEE Communications Surveys Tutorials*, 19(2):1277–1304, 2017.

[4] Analog Devices, Norwood, MA, USA. *12-Bit, 2.6 GSPS/2.5 GSPS/2.0 GSPS, 1.3 V/2.5 V Analog-to-Digital Converter AD9625 Data Sheet.*

[5] Y. Arjoune and N. Kaabouch. A comprehensive survey on spectrum sensing in cognitive radio networks: Recent advances, new challenges, and future research directions. *Sensors (Basel, Switzerland)*, 19(1):126, 01 2019.

[6] E. Axell and E. G. Larsson. Optimal and sub-optimal spectrum sensing of ofdm signals in known and unknown noise variance. *IEEE Journal on Selected Areas in Communications*, 29(2):290–304, Feb. 2011.

[7] P. Bahl, R. Chandra, T. Moscibroda, R. Murty, and M. Welsh. White space networking with wi-fi like connectivity. *ACM SIGCOMM Computer Communication Review*, 39(4):27–38, 2009.

[8] M. Bao and H. Yang. Squeeze film air damping in mems. *Sensors and Actuators A: Physical*, 136(1):3 – 27, 2007. 25th Anniversary of Sensors and Actuators A: Physical.

[9] D. Bhargavi and C. R. Murthy. Performance comparison of energy, matched-filter and cyclostationarity-based spectrum sensing. In *2010 IEEE 11th International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*, pages 1–5, June 2010.

[10] A. Chakraborty, M. S. Rahman, H. Gupta, and S. R. Das. Specsense: Crowdsensing for efficient querying of spectrum occupancy. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2017.

[11] A. A. Cheema and S. Salous. Digital fmcw for ultrawideband spectrum sensing. *Radio Science*, 51(8):1413–1420, 2016.

[12] DigiKey. Data acquisition - analog to digital converters (adc). https://www.digikey.com/products/en/integrated-circuits-ics/data-acquisition-analog-to-digital-converters-adc, Sep. 2020.

[13] Federal Communications Commission. 3.5 ghz band overview. https://www.fcc.gov/wireless/bureau-divisions/mobility-division/35-ghz-band/35-ghz-band-overview, Jan. 2020.

[14] Federal Communications Commission. Fcc adopts new rules for the 6 ghz band, unleashing 1,200 megahertz of spectrum for unlicensed use. https://www.fcc.gov/document/fcc-opens-6-ghz-band-wi-fi-and-other-unlicensed-uses, Apr. 2020.

[15] Federal Communications Commission. White space. https://www.fcc.gov/general/white-space, Jan. 2020.

[16] Y. Ghasempour, C. R. C. M. da Silva, C. Cordeiro, and E. W. Knightly. Ieee 802.11ay: Next-generation 60 ghz communication for 100 gb/s wi-fi. *IEEE Communications Magazine*, 55(12):186–192, 2017.

[17] B. Ghazi, H. Hassanieh, P. Indyk, D. Katabi, E. Price, and L. Shi. Sample-optimal average-case sparse fourier transform in two dimensions. In *2013 51st Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 1258–1265. IEEE, 2013.

[18] S. Gong, Y. Song, T. Manzaneque, R. Lu, Y. Yang, and A. Kourani. Lithium niobate mems devices and subsystems for radio frequency signal processing. In *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 45–48, 2017.

[19] Y. M. Greshishchev, J. Aguirre, M. Besson, R. Gibbins, C. Falt, P. Flemke, N. Ben-Hamida, D. Pollex, P. Schvan, and S. Wang. A 40gs/s 6b adc in 65nm cmos. In *2010 IEEE International Solid-State Circuits Conference - (ISSCC)*, pages 390–391, 2010.

[20] Y. Guddeti, R. Subbaraman, M. Khazraee, A. Schulman, and D. Bharadia. Sweepsense: Sensing 5 ghz in 5 milliseconds with low-cost radios. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, Feb. 2019.

[21] H. Hassanieh, L. Shi, O. Abari, E. Hamed, and D. Katabi. Ghz-wide sensing and decoding using the fourier transform. In *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*, Apr. 2014.

[22] C. Herley and Ping Wah Wong. Minimum rate sampling and reconstruction of signals with arbitrary frequency support. *IEEE Transactions on Information Theory*, 45(5):1555–1564, 1999.

[23] S. Hong and S. Katti. Dof: a local wireless information plane. In *Proceedings of the ACM SIGCOMM 2011 conference*, pages 230–241, 2011.

[24] M. Kadota, S. Tanaka, Y. Kuratani, and T. Kimura. Ultrawide band ladder filter using sh0 plate wave in thin linbo3 plate and its application. In *2014 IEEE International Ultrasonics Symposium*, pages 2031–2034, 2014.

[25] S. Kapoor, S. Rao, and G. Singh. Opportunistic spectrum sensing by employing matched filter in cognitive radio network. In *2011 International Conference on Communication Systems and Network Technologies*, pages 580–583, June 2011.

[26] M. Khazraee, Y. Guddeti, S. Crow, A. C. Snoeren, K. Levchenko, D. Bharadia, and A. Schulman. Sparsdr: Sparsity-proportional backhaul and compute for sdrs. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '19, page 391–403, 2019.

[27] J. N. Laska, W. F. Bradley, T. W. Rondeau, K. E. Nolan, and B. Vigoda. Compressive sensing for dynamic spectrum access networks: Techniques and tradeoffs. In *2011 IEEE International Symposium on Dynamic Spectrum Access Networks (DySPAN)*, 2011.

[28] R. Lu, T. Manzaneque, Y. Yang, A. Kourani, and S. Gong. Lithium niobate lateral overtone resonators for low power frequency-hopping applications. In *2018 IEEE Micro Electro Mechanical Systems (MEMS)*, pages 751–754, 2018.

[29] R. Lu, T. Manzaneque, Y. Yang, J. Zhou, H. Hassanieh, and S. Gong. Rf filters with periodic passbands for sparse fourier transform-based spectrum sensing. *Journal of Microelectromechanical Systems*, 27(5):931–944, 2018.

[30] Y. Ma, Y. Gao, A. Cavallaro, C. G. Parini, W. Zhang, and Y. Liang. Sparsity independent sub-nyquist rate wideband spectrum sensing on real-time tv white space. *IEEE Transactions on Vehicular Technology*, 66(10):8784–8794, 2017.

[31] M. Mishali and Y. C. Eldar. From theory to practice: Sub-nyquist sampling of sparse wideband analog signals. *IEEE Journal of Selected Topics in Signal Processing*, Apr. 2010.

[32] M. Mishali, Y. C. Eldar, O. Dounaevsky, and E. Shoshan. Xampling: Analog to digital at sub-nyquist rates. *IET Circuits, Devices Systems*, Jan. 2011.

[33] T. Mukherjee, G. K. Fedder, H. Akyol, U. Arslan, J. Brotz, F. Chen, A. Jajoo, C. Lo, A. Oz, D. P. Ramachandran, V. K. Saraf, M. Sperling, and J. Stillman. Reconfigurable mems-enabled rf circuits for spectrum sensing. In *Government Microcircuit Applications and Critical Technology Conference*, 2005.

[34] B. Murmann. A/d converter trends: Power dissipation, scaling and digitally assisted architectures. In *2008 IEEE Custom Integrated Circuits Conference*, pages 105–112, 2008.

[35] A. Nasser, A. Mansour, K. C. Yao, H. Charara, and M. Chaitou. Efficient spectrum sensing approaches based on waveform detection. In *The Third International Conference on e-Technologies and Networks for Development (ICeND2014)*, pages 13–17, April 2014.

[36] C. T. C. Nguyen. Mems-based rf channel selection for true software-defined cognitive radio and low-power sensor communications. *IEEE Communications Magazine*, 51(4):110–119, 2013.

[37] D. Pfammatter, D. Giustiniano, and V. Lenders. A software-defined sensor architecture for large-scale wideband spectrum monitoring. In *Proceedings of the 14th International Conference on Information Processing in Sensor Networks*, pages 71–82, 2015.

[38] H. Rahul, N. Kushman, D. Katabi, C. Sodini, and F. Edalat. Learning to share: narrowband-friendly wideband networks. *ACM SIGCOMM Computer Communication Review*, 38(4):147–158, 2008.

[39] M. Rashidi, K. Haghighi, A. Panahi, and M. Viberg. A nlls based sub-nyquist rate spectrum sensing for wideband cognitive radio. In *2011 IEEE International Symposium on Dynamic Spectrum Access Networks (DySPAN)*, pages 545–551. IEEE, 2011.

[40] M. Rinaldi, C. Zuniga, C. Zuo, and G. Piazza. Ultra-thin super high frequency two-port aln contour-mode resonators and filters. In *TRANSDUCERS 2009 - 2009 International Solid-State Sensors, Actuators and Microsystems Conference*, pages 577–580, 2009.

[41] S. K. Sharma, E. Lagunas, S. Chatzinotas, and B. Ottersten. Application of compressive sensing in cognitive radio communications: A survey. *IEEE Communications Surveys Tutorials*, 18(3):1838–1860, 2016.

[42] L. Shi, P. Bahl, and D. Katabi. Beyond sensing: Multighz realtime spectrum analytics. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, May 2015.

[43] L. Shi, H. Hassanieh, and D. Katabi. D-bigband: Sensing ghz-wide non-sparse spectrum on commodity radios. In *Proceedings of the 6th Annual Workshop on Wireless of the Students, by the Students, for the Students*, S3 '14, page 13–16, 2014.

[44] A. J. Silva. Reconstruction of undersampled periodic signals. *MIT Technical Report*, 1986.

[45] Y. Song and S. Gong. Wideband spurious-free lithium niobate rf-mems filters. *Journal of Microelectromechanical Systems*, 26(4):820–828, 2017.

[46] J. Su and W. Wu. Wireless spectrum prediction model based on time series analysis method. In *Proceedings of the 2009 ACM workshop on Cognitive radio networks*, pages 61–66, 2009.

[47] S. Subramaniam, H. Reyes, and N. Kaabouch. Spectrum occupancy measurement: An autocorrelation based scanning technique using usrp. In *2015 IEEE 16th Annual Wireless and Microwave Technology Conference (WAMICON)*, pages 1–5, 2015.

[48] H. Sudo, K. Kosaka, H. Kanemoto, N. Gejoh, T. Yasunaga, and M. Uesugi. Study of spectrum sensing scheme using received power within preamble signals. In *2017 20th International Symposium on Wireless Personal Multimedia Communications (WPMC)*, pages 592–597, 2017.

[49] H. Sun, A. Nallanathan, C. Wang, and Y. Chen. Wideband spectrum sensing for cognitive radio networks: a survey. *IEEE Wireless Communications*, 20(2):74–81, April 2013.

[50] The Software Defined Radio Forum Inc. Application of management technologies in dynamic spectrum sharing, Jul. 2019.

[51] J. A. Tropp, J. N. Laska, M. F. Duarte, J. K. Romberg, and R. G. Baraniuk. Beyond nyquist: Efficient sampling of sparse bandlimited signals. *IEEE transactions on information theory*, 56(1):520–544, 2009.

[52] N. Tzou, D. Bhatta, S. Hsiao, H. W. Choi, and A. Chatterjee. Low-cost wideband periodic signal reconstruction using incoherent undersampling and back-end cost optimization. In *2012 IEEE International Test Conference*, pages 1–10, 2012.

[53] U.S. Government. CFR title 47 section 96.67 environmental sensing capability, Jan. 2020.

[54] P. P. Vaidyanathan and P. Pal. Sparse sensing with coprime samplers and arrays. *IEEE Transactions on Signal Processing*, 59(2):573–586, 2011.

[55] R. Venkataramani and Y. Bresler. Perfect reconstruction formulas and bounds on aliasing error in sub-nyquist nonuniform sampling of multiband signals. *IEEE Transactions on Information Theory*, 46(6):2173–2183, 2000.

[56] X. G. Xia. On estimation of multiple frequencies in undersampled complex valued waveforms. *IEEE transactions on signal processing*, 47(12):3417–3419, 1999.

[57] X. G. Xia. An efficient frequency-determination algorithm from multiple undersampled waveforms. *IEEE Signal Processing Letters*, 7(2):34–37, 2000.

[58] R. T. Yazicigil, T. Haque, M. R. Whalen, J. Yuan, J. Wright, and P. R. Kinget. Wideband rapid interferer detector exploiting compressed sampling with a quadrature analog-to-information converter. *IEEE Journal of Solid-State Circuits*, Dec 2015.

[59] J. Yoo, S. Becker, M. Loh, M. Monge, E. Candes, and A. Emami-Neyestanak. A 100mhz–2ghz 12.5 x subnyquist rate receiver in 90nm cmos. In *2012 IEEE Radio Frequency Integrated Circuits Symposium*, pages 31–34. IEEE, 2012.

[60] J. Yoo, C. Turnes, E. B. Nakamura, C. K. Le, S. Becker, E. A. Sovero, M. B. Wakin, M. C. Grant, J. Romberg, A. Emami-Neyestanak, and E. Candes. A compressed sensing parameter extraction platform for radar pulse signal acquisition. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2(3):626–638, 2012.

[61] S. Yoon, L. E. Li, S. C. Liew, R. R. Choudhury, I. Rhee, and K. Tan. Quicksense: Fast and energy-efficient channel sensing for dynamic spectrum access networks. In *2013 Proceedings IEEE INFOCOM*, April 2013.

[62] T. Yucek and H. Arslan. A survey of spectrum sensing algorithms for cognitive radio applications. *IEEE communications surveys & tutorials*, 11(1):116–130, 2009.

[63] Y. Zeng, V. Chandrasekaran, S. Banerjee, and D. Giustiniano. A framework for analyzing spectrum characteristics in large spatio-temporal scales. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–16, 2019.

[64] C. Zuo, N. Sinha, and G. Piazza. Very high frequency channel-select mems filters based on self-coupled piezoelectric aln contour-mode resonators. *Sensors and Actuators A: Physical*, 160(1):132 – 140, 2010.

Appendix A Alternative S^3 Architectures

As we mentioned in section 7, we can extend S^3 to use two or more MEMS filters in parallel to enlarge the sensing bandwidth or enable S^3 to sense different channel allocations including narrower bands and even non-uniformly allocated

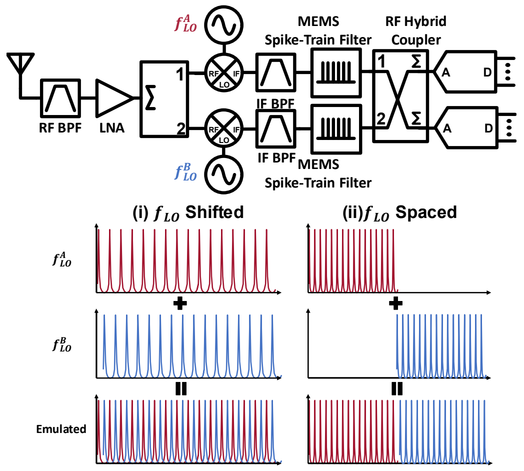


Figure 16: Circuit diagram and emulated spike-train filter for alternative S^3 architecture leveraging LO center frequency difference.

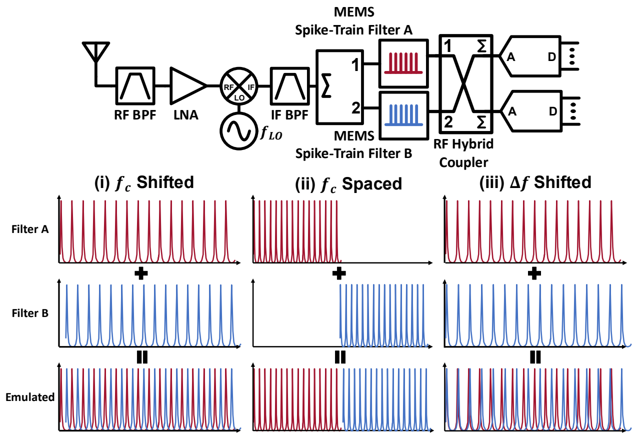


Figure 17: Circuit diagram and emulated spike-train filter for alternative S^3 architecture leveraging different MEMS spike-train filters.

bands. This system level parallelism introduces another degree of freedom and allows us to break the fixed design tradeoffs at the filter level. Here, we present some alternative architectures of S^3 that combine two spike-trains filters.

(1) Changing LO Center Frequencies: First, we can combine two identical spike-train filters that are fabricated using the same process. Hence, these two filters will have almost identical frequency responses with the same center frequency f_c and spacing between spikes Δf. In order to cover different frequencies in the RF spectrum, we use the two filters on separate receiver RF chains with different LO frequencies. We demonstrate the circuit diagram and the emulated spike trains in the RF spectrum in Fig. 16.

After bandpass filtering and amplifying the received signal, we split the RF signal into two channels, and use two LOs with center frequencies f_{LO}^A and f_{LO}^B to down-convert the signal to the IF frequencies. Then we pass each IF signal into a spike-train filter to sampling the spectrum along the frequency axis. Based on the LO frequency difference $df_{LO} = f_{LO}^B - f_{LO}^A$, we can emulate two types of spike trains, as shown in Fig. 16(i) and (ii). When $\Delta f_{LO} < \Delta f$, the two spike trains are slightly shifted on the frequency axis as shown in Fig. 16(i). As a result, we can emulate a spike-train filter with narrower spacing between the spikes. This increases the frequency domain sampling rate of the filter and enables S^3 to sense narrower channel bandwidths. On the other hand, when $df_{LO} = K\Delta f$, the two spike trains are concatenated along the frequency axis as shown in Fig. 16(ii). In this way, a longer spike train with more spikes covering wider bandwidth is emulated.

Although it is straight forward to sample the two IF signals separately, the number of ADCs required will increase linearly with the number of spike-train filters. Instead, after passing IF signals on the two channels through the spike-train filters, we combine the filtered signals and sample the combined signal using two low-speed ADCs. The analog combination and splitting can be achieved using an RF power combiner in series with an RF power splitter, but it can also be done using a RF hybrid coupler. Note that with more spikes in the emulated filter, there will be more aliasing in the subsampled spectrum. Therefore, a higher ADC sampling rate might be needed, but the sampling rate should be able to scale sublinearly with respect to the number of spikes. Besides, the ADC input cutoff frequency needs to be higher than the spike-train bandwidth.

The advantage of this architecture is that we can use the same MEMS spike-train filter on the two channels without needing to redesign a new filter. However, it requires two LOs and mixers which increases the cost and power consumption of the system. [6]

(2) Changing Spike-Train Filter Structure: Instead of introducing a second local oscillator, we can use only one LO and two different spike-train filters to emulate spike trains with wider bandwidth as well as narrower or nonuniform spike spacing. As we mentioned in section 3, we can modify the width of the piezoelectric film and the position of the electrodes to obtain different Δf and f_c. When two spike-train filters with different Δf and/or f_c are combined in parallel, we can emulate a spike train with more sophisticated sparsity structures. We show the circuit diagram for this type of alternative architectures in Fig. 17, along with three emulated spike-train filter frequency responses.

In this architecture, the down-converted IF signal is split and filtered by two different MEMS spike-train filters, whose center frequencies and spike spacing are f_c^A, f_c^B, and Δf^A, Δf^B. The output spectra of the two filters are then combined and sampled by two low-speed ADCs. Using this architecture, we can emulate the same spike trains as the first alternative architecture. For instance, when the difference between the filter center frequencies $df_c = f_c^B - f_c^A < \Delta f$, as shown in Fig. 17(i), the two spike trains are slightly shifted on the

[6]Since the power consumption of spectrum sensors is dictated by the ADC [58], the additional power consumption of the second LO is not the primary concern.

frequency axis and emulate a spike-train filter with narrower spacing between the spikes. Besides, when $df_c = K\Delta f$, the two spike trains are spaced by the bandwidth of the spike train and emulate a wider bandwidth spike train as shown in Fig. 17(ii). However, in additional to enlarging the filter bandwidth and narrowing the spike spacing, we can even emulate a non-uniformly spike train as as shown in Fig. 17(iii). This is achieved by combining two spike-train filters with different Δf^A and Δf^B. Such spike train profile provides us with all sorts of frequency resolutions across the spectrum to accommodate the different channel bandwidth required by the secondary users in TV Whitespace and CBRS bands.

WiForce: Wireless Sensing and Localization of Contact Forces on a Space Continuum

Agrim Gupta, Cédric Girerd, Manideep Dunna, Qiming Zhang, Raghav Subbaraman, Tania K. Morimoto, Dinesh Bharadia
{agg003,cgirerd,mdunna,qiz127,rsubbaraman,tkmorimoto,dineshb}@eng.ucsd.edu
University of California, San Diego

Abstract

Contact force is a natural way for humans to interact with the physical world around us. However, most of our interactions with the digital world are largely based on a simple binary sense of touch (contact or no contact). Similarly, when interacting with robots to perform complex tasks, such as surgery, richer force information that includes both magnitude and contact location is important for task performance. To address these challenges, we present the design and fabrication of WiForce which is a 'wireless' sensor, sentient to contact force magnitude and location. WiForce achieves this by transducing force magnitude and location, to phase changes of an incident RF signal of a backscattering tag. The phase changes are thus modulated into the backscattered RF signal, which enables measurement of force magnitude and contact location by inferring the phases of the reflected RF signal. WiForce's sensor is designed to support wide-band frequencies all the way up to 3 GHz. We evaluate the force sensing wirelessly in different environments, including through phantom tissue, and achieve force accuracy of 0.3 N and contact location accuracy of 0.6 mm.

1 Introduction

Our sense of touch is critical for understanding and interacting with the world around us. While interacting with the physical world, force-sensitive mechanoreceptors in the skin respond to various vibrations, motions, pressures, and stretching of the skin to provide us with critical information on the location and magnitude of the stimuli [1]. Thus, if we want the next generation of tactile sensors to emulate how our skin reacts to stimuli, we need to both sense the magnitude and location of contact forces acting on the sensing surface.

Current skin-like continuum tactile sensors enable numerous critical applications. These applications mostly involve dexterous tasks to be performed via mechanical tools or robotic manipulators, rather than via human hands. For example, in order to grasp and manipulate an object, a robot must be able to sense where and how firmly it is pressing the object [2, 3]. Another example can be seen during minimally invasive surgery, where a surgeon must operate inside the body with a surgical tool that naturally contacts numerous tissues throughout the procedure. A sensing layer which acts like a skin covering the entire surgical tool could enable safer surgeries [4–6], since the surgeon would know exactly where the tool is in contact with the tissues and with how much force. In addition to these robotics applications, tactile sensing can supplement our interactions with the digital world. Most of our current interactions with digital technologies occur with aid of a touchscreen, which binarizes human contact into simply touch/no-touch, and the richer information on contact force is typically lost. Augmenting our digital interfaces with the capability to sense the magnitude of the forces with which we interact with them could lead to more natural, intuitive, and realistic interactions, creating new possibilities for the evolving AR/VR settings [7–9].

Driven by these applications, design of such continuum sensor skins has been an active area of research over the past decade [10–20]. The common approach has been to create a sensing surface consisting of an array of discrete force sensitive resistors or electrodes, whose measurements are interpolated to reconstruct a continuum force profile. However, this approach has prohibitive wiring costs [10–16], since it requires a wired link to obtain data from each individual sensor, as well as wires for satisfying the power requirements. In scenarios where space is a premium, including surgical robotic applications, this wiring challenge is exacerbated, and force sensing for the surgical robotics has been acknowledged as a 'Grand-Challenge' [21]. One way to address the wiring requirements is to reduce the density of sensors in the surface and improve the interpolating algorithms [10–12]. A more drastic solution is to eliminate the wiring problem completely by creating new sensing modalities with modest power requirements such that both the sensor feedback and power can be delivered wirelessly [13, 14, 16].

Motivated by these challenges, we present WiForce, which makes progress in this direction by sensing force magnitude and location over a 1-D continuum by leveraging backscattering techniques. Rather than generating a wireless signal to feed back the sensor readings, which would require power-hungry electronics, WiForce's sensor transduces force magnitude and location directly onto the reflections of the incident RF signals. The design has very minimal power requirements, and consists of only one antenna, a small identification unit, and a force continuum surface. Thus, WiForce presents a new tactile sensing modality, which makes headway towards batteryless wire-free sensor skins.

The key enabler for such a low-powered design is the

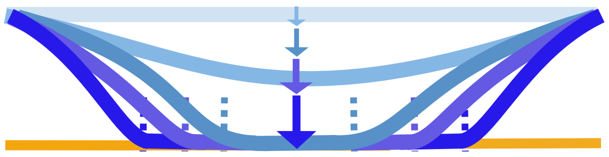


Figure 1: Beam bending in effect of contact force: As contact force increases (shown via increasing arrow lengths), the top beam bends and collapses more and more onto the bottom beam

transduction mechanism, which modulates the reflected signal with information on the contact force and its location, by altering the RF signal parameters as applied force on the sensor changes. To achieve this, WiForce links contact force, a mechanical entity, to RF signal parameters by combining classical beam bending models and RF transmission line concepts, using a novel sensor surface. This sensor surface consists of two parallel conductive traces, similar to a microstrip line, augmented with a soft specialized polymer beam. As a force is applied at a specific location on the sensor surface, the beam bends, causing the traces to connect (Fig. 1). From the RF perspective, this beam bending leads to shorting of the traces, which causes reflection of signals. From the mechanical perspective, the soft beam allows us to use beam bending models to characterize how the shorting phenomenon changes as the applied force increases.

Essentially, the shorting points shift towards the ends of the sensor, as the applied force increases and the soft layer of the beam bends and flattens on the bottom trace (Fig. 1). By estimating the shorting lengths from both ends of the sensor, we can determine the magnitude and location of the applied force. The shorting lengths are related to the signal phases measured on both the ends of the sensor. Basically, the longer the signal travels on the sensor surface, the more phase change it will accumulate. The goal at the wireless reader is to measure the accumulated phases due to signal propagation on the sensor surface from both the ends, in order to use the transduction mechanism to sense and localize the forces.

To enable sensing of these phases by the wireless reader, the phases from both ends have to be disambiguated, and thus each end has to be given an identity. To do so, a naive solution would be to have RF switches toggling on-off with different frequencies on either ends (f_{s_1}, f_{s_2}, Fig. 2), with the toggling frequency providing the unique identity to each of the ends. However, this naive solution does not work out of the box, because the two ends are electrically connected to each other via the transmission line, causing signals to leak from one end to the other, which would in turn cause intermodulation effects. To resolve this issue, WiForce comes up with a creative RF switch toggling strategy, which not only provides electrical isolation to combat intermodulation, but also provides different identities to these ends in terms of different frequency shifts. Thus, the external wireless

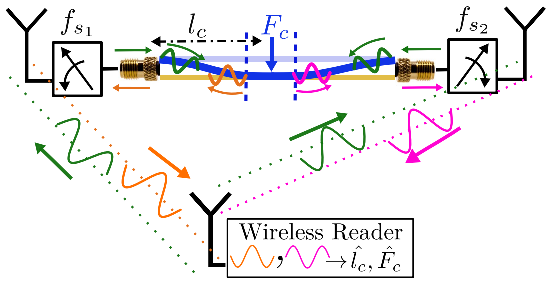


Figure 2: The key insight of WiForce is to view the parallel beams as a microstrip line. Force F_c and it's location l_c gets transduced onto changes in the reflections due to the line shorting caused by beam bending. The wireless reader uses the reflected signals to estimate l_c, F_c

reader is able to view the sensor ends as having different identities in frequency domain, as envisioned by the intuitive scheme in Fig. 2, with the intermodulation problem abstracted out via the intelligent toggling scheme.

The final piece in WiForce is designing the wireless reader, such that it can use any wireless device (like WiFi (OFDM) or LoRa (FMCW)) with wide-band transmission to read the WiForce's force and location. The task of the wireless reader is two-fold, first identify and isolate the signals coming from the sensor and second, accurately track the phase of the sensor signal. Since the wireless phase observed by the reader can also be altered by various entities in the environment, the task of reading phase changes stemming only from the sensor is non-trivial. Hence, WiForce designs a novel signal processing algorithm which utilizes periodic wide-band channel estimates to pick up the reflection signatures from the sensor to isolate the signal, as well as to read the phase changes at multiple frequencies, providing robustness to the phase sensing requirements for the proposed force transduction mechanism.

We designed and fabricated the sensor with a soft-polymer augmented microstrip line, which is 'force-sensitive'. That is, the microstrip line sensing surface has bending properties which maximize the phase changes transduced by contact forces. This sensing surface was retrofitted with RF-switches and antenna to enable backscattered feedback. The fabricated sensor works for the entire sub-3 GHz verified with the test equipment (vector network analyzer). We evaluated the WiForce sensor abilities to report force magnitude and location in multiple settings, both indoors and inside a body-like environment using gelatin. We used USRP radios as the readers, and tested the sensor at 900 MHz and 2.4 GHz, which are the two most popular ISM bands. We show that the sensor can be read up to 5 meters of range over the air, and show the algorithm working even with propagation through the gelatin-based muscle/fat/skin tissue layers composition similar to the human body to demonstrate the surgical applications. We achieved phase sensing with an accuracy as low as 0.5^o, giving us a force resolution of 0.3 N, and location accuracy of 0.3 mm. We also showcase the ability to read from multiple sensors, by sensing forces from 2 sensors simultaneously. Finally, we even evaluated our force sensor with a user pressing with his hand, and we achieved force resolution of 0.3 N, and location accuracy of 0.3 mm. In fact, recent interfaces for Human-Computer Interactions (HCI) work shows that similar resolution (0.2 N) is required to support force enabled gestures on smartphones and desktop computers [22]. We believe this is the first step towards enabling numerous force sensing applications.

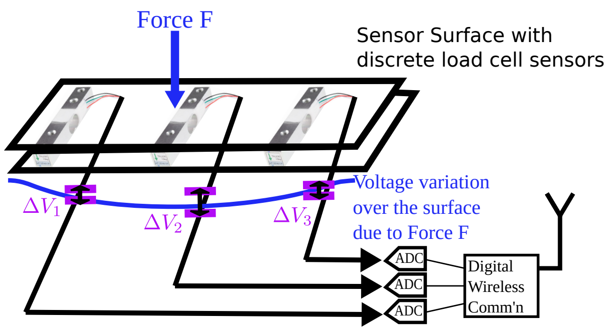
2 Background and Motivation

The problem of sensing tactile phenomenon over a surface continuum has attracted considerable research interest [10–12, 15, 18, 20]. The usual approach has been to densely populate the surface with either force sensitive resistors [15], electrodes [10–12, 20], or force sensitive yarns [18]. The continuum sensing is performed by interpolating over the sensor readings of these discrete sensors. Numerous papers in the past decade have raised the issues stemming from the wiring requirements of the developed sensor skin modalities [13, 14, 16].

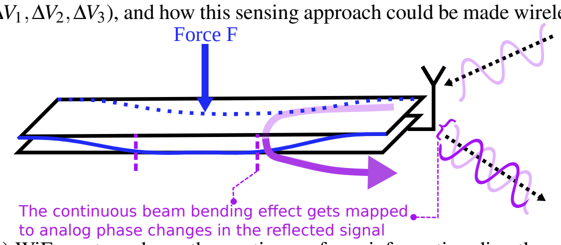
Researchers have tried addressing these issues by considering sparser deployments, such as considering sensors only on the boundaries [20], or populating the sensors in a minimal way across the surface [10, 23]. Although these efforts have reduced the wiring requirements for sensing considerably, these surfaces still lack a solution to both feedback the sensor readings wirelessly, as well as get rid of wired battery connections required for these sensing efforts. Recent review papers have advocated the need of powering up these sensing surfaces with energy harvesting methods to alleviate the battery requirements [14, 16], and a backscatter-enabled sensor is a promising approach to address the battery concerns.

Before re-designing the sensing modalities to be compatible with low-powered backscatter communications, a key question to answer is whether a hybrid solution would work. That is, can we take one of the sensing solutions requiring the least number of wired connections across the surface [10, 20] and feedback the sensor readings via currently developed backscattering RFICs[1] [24, 25]. However, this solution won't suffice since these backscatter links typically work with a RF energy harvestor, which generates small voltages capable of powering a small RFID chip, and not a large continuum sensing surface. Further we would need to sense multiple voltages from these electrodes via an array of ADCs (Fig. 3), managed by a micro-controller, which

[1]This fusion of sensor skin + backscatter RFIC has not been yet demonstrated, however we consider it as an hypothetical scenario


Force F
Sensor Surface with discrete load cell sensors
ΔV_1
ΔV_2
ΔV_3
Voltage variation over the surface due to Force F
ADC
ADC
ADC
Digital Wireless Comm'n

(a) Shows a possible continuum sensing approach with existing wired force sensors, which sense discrete voltage changes over a continuum ($\Delta V_1, \Delta V_2, \Delta V_3$), and how this sensing approach could be made wireless


Force F
The continuous beam bending effect gets mapped to analog phase changes in the reflected signal

(b) WiForce transduces the continuum force information directly onto analog backscattered phase changes without discretization

Figure 3: Force feedback design, WiForce in comparison with a possible wireless extension to existing sensing modalities

would then digitize and buffer the data for transmission through the low capacity backscatter link.

Hence, WiForce attempts a RF-only analog approach, where the sensing modality directly transduces force and its location into wireless signal phase changes, which can be read by a radio over the air. The argument here is that, if analog phase readings can be fed back accurately, it would require much less power than procuring the analog readings, digitizing/buffering them, and then sending them over the backscatter link. Thus, the novel force to phase transduction mechanism, coupled with the analog phase feedback, fulfills both the key requirements for low powered tactile sensing – the ability to sense over a continuum and low-powered battery-free operation.

3 Design of WiForce

In this section, we present the design principles of WiForce. First, we describe WiForce's force transduction mechanism, which translates the force and its application location to changes in the RF signal phase. Next, we present novel algorithms to measure changes in the RF properties to deduce the force and its application location wirelessly. Finally, we conclude with a robust channel measurement technique that uses a wireless waveform to read the sensors while rejecting multi-path.

3.1 Force Transduction Mechanism

As a first step towards a backscattered force sensor, WiForce has to formulate a transduction mechanism which relates force magnitude and application location, to parameters like RF signal amplitude and phase, which can then be used to modulate the sensor readings onto the

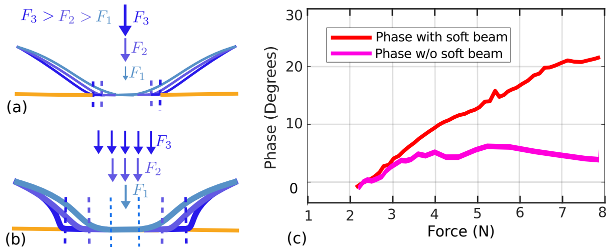


Figure 4: Bending of (a) Thin, (b) Soft beam augmented thick trace, as forces increase ($F_1 < F_2 < F_3$). The soft beam distributes forces along the length, which leads to profound phase changes (c) as compared to thin traces w/o soft beam

reflected signals. The challenge here is to take an object (like transmission lines) which supports RF signal propagation, and make it force-sensitive. That is, RF propagation in this object should give significant changes in its signal parameters as we press the object at different locations with varying force magnitudes. In this section, we will elaborate on how WiForce makes microstrip lines 'force-sensitive'.

A microstrip line traditionally consists of two parallel conducting traces– the signal trace and the ground trace. A force applied to the microstrip line would cause the traces to bend and come in contact with each other, which shorts the line and leads to signal reflections. The reflections produced by this shorting have different phase accumulation based on the location of pressing. However, this reflected phase is not sensitive to force magnitude at all. That is, irrespective of the contact force applied, the traces will short each other only in the vicinity of a single point (Fig. 4a). The contact point invariance leads to a near invariant phase response as force is changed (Fig. 4c), therefore preventing the measurement of force through phase changes.

WiForce modifies the traditional microstrip line by augmenting a new soft, flexible beam on top of the signal trace to address this problem and make the microstrip line force-sensitive. The key insight here is that the soft beam distributes the force along the length of the trace (Please refer to [26] for details on the mechanical implementation of the sensor). The distributed force leads to a finite length of the signal trace to come in contact with the ground trace, creating two distinct shorting points, as shown in Fig. 4b. Further, these shorting points shift towards the ends as the applied force magnitude increases. Varying shifts induce different phase changes since the signals travel less distance on the microstrip line before getting reflected at the shorting points. Hence, the reflections caused by higher magnitude forces accumulate less phase relative to the reflected phases when lower force was applied. Thus, the soft beam augmented microstrip line allows phase to force transduction (Fig. 4c).

This beam bending effect manifests itself in the form of a varying phase-force relationship depending on the con-

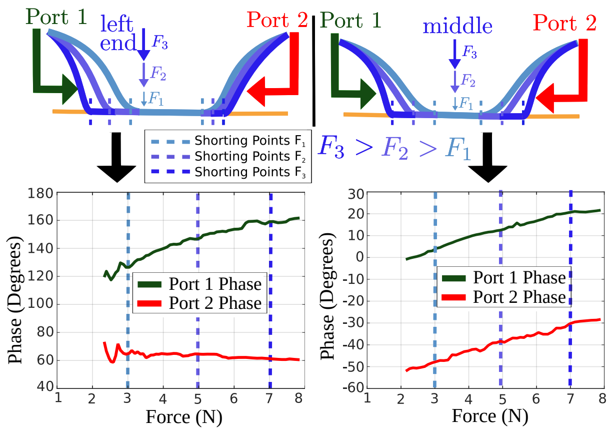


Figure 5: The shorting points shift due to increased force because of bending of the soft beam, changing the reflected phase. When the force is applied at the middle, we observe symmetric phase change across the two ports, whereas when force is applied towards the ends, we see asymmetric phase changes, due to the beam bending mechanism illustrated in the top row

tact force's point of application (Fig. 5, top images). A force applied in the middle of the sensor compresses it symmetrically, and therefore the reflected signals from both the ends show similar phase changes. In contrast, a force that acts asymmetrically will disproportionately compress the smaller length of the beam. Therefore, the end near the smaller length shows a higher phase shift than the end near the longer one. The longer length collapses onto the bottom trace, leading to an almost stationary shorting point as the force increases (Fig. 5, bottom images). These varying phase changes that depend on the location of a contact force also allow WiForce to localize the force application point. Thus, the double-ended measurement allows us to estimate the applied force's magnitude and its application location along the sensor length. However, at the same time, this asymmetric behavior of phase change with force contact location necessitates sensing phases from both ends of the sensor.

3.2 Two-ended backscatter modulation

As described in the previous section, sensing phases from both ends of the sensor forms the cornerstone of the phase to force transduction mechanism. This is because it allows to disambiguate the different force profiles observed as the sensor is pressed at different locations, which basically allows us to both locate and then measure how much force was being applied. In this section, we will go over how to attempt this double ended phase sensing via wireless backscatter sensing.

The first and foremost thing which any backscatter sensing solution needs, is an ability to give an identification to the reflections occurring at the sensor. This identification helps the wireless reader isolate the sensor reflections from the environmental clutter. A popular technique to do so has been to use RF switches toggling at

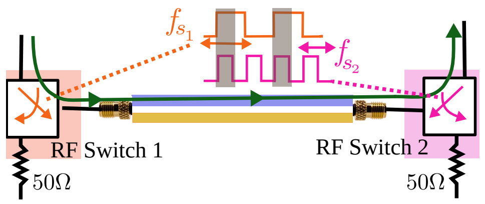


Figure 6: RF switches toggle between sensor (on) and 50 Ω (off) depending on the control input. Different freq. clocks as control inputs introduce intermodulation due to both switches being toggled on at the same time (grey shaded time instants)

different frequencies as identification unit [27–29]. This technique basically multiplies the incident signal with an on-off modulation of certain switching frequency.

In frequency domain, this operation leads to frequency shifts corresponding to the switching frequency. Putting this mathematically, say the sensor receives the excitation signal $s(t)$ and reflects $s(t)m(t)$ where $m(t)$ is a square wave, with time period T. Expanding $m(t)$'s Fourier series, we get odd harmonics, $m(t) = \sum_{k\in(2i+1),i\in\mathbb{Z}} \frac{1}{|k|} e^{(j2\pi k f_s t)}$ where $f_s = \frac{1}{T}$. Ignoring the high order harmonics, we get reflected signal as $r(t) = s(t)m(t) \approx s(t)e^{j2\pi f_s t}$. Hence the reflected signal will be shifted by the switching frequency $\pm f_s$, which allows the reflected signal, $r(t)$, to be isolated from the excitation signal, $s(t)$ in the frequency domain.

A naive extension of this idea to the double ended sensing problem at hand, would be to have switches at both ends of the sensor and make them switch at different frequencies (f_{s_1}/f_{s_2}). Theoretically, this solution should give separate identities to reflections emanating from both the ends. However, the problem at hand is inherently coupled to allow for such naive de-coupled solutions, because both the ends are physically connected to each other via a microstrip line. When both the switches are toggled-on, the signals will propagate through the sensor and leak out from the other end (Fig. 6). This causes intermodulated reflections, where the reflected signal would be partially modulated by both toggling frequencies, leading to muddled up identities.

The challenge in avoiding the intermodulation effects is that the sensor has to reflect signals from the opposite end when a switch is toggled on. Using reflective RF switches in the off state allows us to make either of the ends reflective, under the restriction that the other should be off when one switch is on. Said differently, we want to design a coupled two-ended switching scheme, which gives separation in frequency domain, under the constraint that both switches are not 'on' at the same time. The unique insight which allows WiForce to have such an switching scheme, is the use of duty cycle properties of square wave Fourier series.

In a standard square wave, with 50% duty cycle, all the even harmonics (i.e. every second harmonic) are absent. Similarly, in a wave with 25% duty cycle, ev-

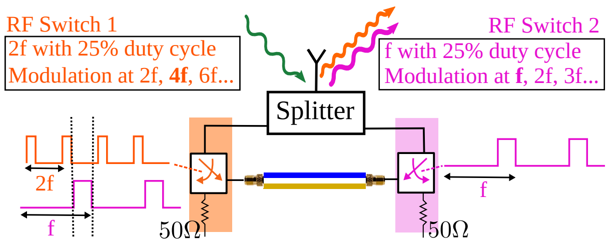


Figure 7: The duty-cycled modulation ensures that switches aren't toggled on at once, as well as providing freq. separation

ery fourth harmonic would be absent. Hence, a frequency f_s, 25% duty cycle square wave will give modulation at $\underline{\mathbf{f_s}}, 2f_s, 3f_s, \cancel{4f_s} 5f_s, \ldots$ Similarly, a frequency $2f_s$, 25% duty cycle square wave will give modulation at $2f_s, \underline{\mathbf{4f_s}}, 6f_s, \cancel{8f_s}, 10f_s \ldots$ Observe that a combination of these 2 clocks will cause interference at $2f_s$, but can be read up individually at f_s for the former clock, and $4f_s$ for the latter clock. Hence, a combination of these 2 clocks can provide separation in the frequency domain. Also, by controlling the initial phases of these two clocks, we can suppress the intermodulation problem as well. This is possible because when one clock is high, the other clock will be guaranteed to be low and vice versa (Fig. 7). Hence, at any given time, only one port will be on, and other port will be reflective open.

Further, this clocking design allows us to reduce the form factor requirements, instead of having 2 antennas, one for each end of the sensor, we can just have just a one antenna design using a splitter. Since the clocking strategy provides separation in the frequency domain, we can add the modulated signals from the either ends via a splitter. Thus, the wireless reader can identify the two ends by reading at $f_s, 4f_s$ frequency shifts.

3.3 Sensing Forces at the Wireless Reader

Till now, we have described the phase-force transduction mechanism, and delineated a method to give disambiguated identities to both ends of the sensor. Now, in this section, we move on to the description of how the wireless reader is designed. We design our wireless reader to detect the separate identities stemming from frequency shifts, and then extract the valuable phase information which allows us to sense and localize forces. The key insight of WiForce here is to view the frequency shifts from the sensor as 'artificial-doppler' and use wide-band channel estimates in order to estimate the doppler and thus isolate the signal coming from the sensor. This approach to view the backscatter tag's frequency shift as an artificial doppler has been also utilized in some of the recent past work [28]. Finally, to obtain the required analog phase estimates required to sense and localize the forces, we utilize that fact that force, a mechanical quantity, changes slowly (at about 1 kHz rate [30–32]), as compared to MHz's of RF bandwidth. This allows us to group the channel estimates and perform a 'short-time phase transform', which enables us to track the phase shifts at the two artificial doppler bins corresponding to the two ends of the sensor.

The algorithm WiForce uses to extract the backscattered phases embedded inside the wideband channel estimates is visually illustrated in Fig. 8. Say we are estimating the channel periodically after every T seconds, with frequency steps of F. If we use OFDM channel sounding strategy, T will be the time of the OFDM frame, and F will be the subcarrier spacing. We denote $H(kF, nT) = H[k,n]$ where k is subcarrier index and n is time index. If there are M multipaths in addition to the signal coming from the sensor, we can write the channel estimates from geometric channel model as

$$H[k,n] = \sum_{i=1}^{M} \alpha_i e^{-j2\pi kF\frac{d_i}{c}} + (s_1(nT)e^{-j\phi_n^1} + s_2(nT)e^{-j\phi_n^2})\alpha_s e^{-j2\pi kF\frac{d_s}{c}} \quad (1)$$

Here, α_i is the attenuation factor for the i-th path, d_i is the distance separation between the TX-reflector and reflector-RX, and ϕ_n^1, ϕ_n^2 is the phase accumulated from the RF propagation in the microstrip line sensor at time index n from sensor end 1 and sensor end 2. $s_1(t), s_2(t)$ are the duty-cycle square wave modulations to give intermodulation free frequency identities at $f_s, 4f_s$ as discussed in Section 3.2. Ignoring the higher harmonics terms in $s_1(t), s_2(t)$, we get

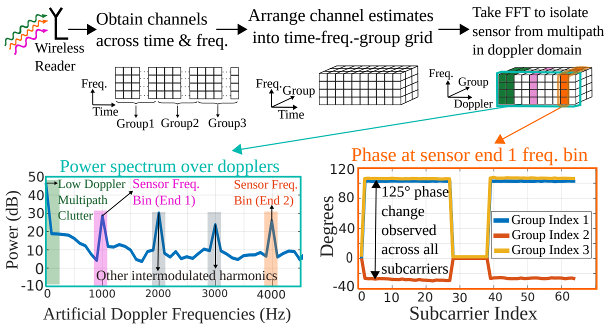
$$H[k,n] \approx \sum_{i=1}^{M} \alpha_i e^{-j2\pi kF\frac{d_i}{c}} + (e^{j2\pi(f_s)nT}e^{-j\phi_n^1} + e^{j2\pi(4f_s)nT}e^{-j\phi_n^2})\alpha_s e^{-j2\pi kF\frac{d_s}{c}} \quad (2)$$

Now, to isolate the signal from the sensor, we take FFT over the n index, to obtain $\tilde{H}[k,f]$. We observe N channel snapshots to calculate, $\tilde{H}[k,f] = \sum_{n=1}^{N} H[k,n]e^{-j2\pi fnT}$. Assuming ϕ_n^1, ϕ_n^2 stay constant over the period of N snapshots, at f_s, $4f_s$, we have,

$$\tilde{H}[k,\{f_s,4f_s\}] = \sum_{n=1}^{N} H[k,n]e^{-j2\pi\{f_s,4f_s\}nT} = \alpha_s e^{-j2\pi kF\frac{d_s}{c}} e^{j\phi_n^{\{1,2\}}} \quad (3)$$

Observe that for this transform, the nyquist frequency would be $\frac{1}{2T}$, and hence, f_s has to be chosen such that $4f_s \leq \frac{1}{2T}$. The switching frequency f_s can be related to an equivalent Doppler, $f_s = \frac{f_c v}{c}$, and thus an object in the environment moving at velocity $v = \frac{cf_s}{f_c}$ would create interference with the sensor signal. However, the chosen f_s is large enough so that this equivalent speed is so high to guarantee that, the signal observed in the frequency bins corresponding to $f_s, 4f_s$ are free from multipath clutter.

However, recall that while writing Eqn. (3), we assumed ϕ_n^1, ϕ_n^2 stay constant as n goes from 1 to N. That is, the transform is only valid when the phases from the sensor ends do not change much over the period of taking



Wireless Reader
Obtain channels across time & freq.
Arrange channel estimates into time-freq.-group grid
Take FFT to isolate sensor from multipath in doppler domain
Freq.
Time
Group1
Group2
Group3
Freq.
Group
Time
Freq.
Group
Doppler
Power spectrum over dopplers
Low Doppler Multipath Clutter
Sensor Freq. Bin (End 1)
Sensor Freq. Bin (End 2)
Other intermodulated harmonics
Power (dB)
Artificial Doppler Frequencies (Hz)
Phase at sensor end 1 freq. bin
125° phase change observed across all subcarriers
Group Index 1
Group Index 2
Group Index 3
Degrees
Subcarrier Index

Figure 8: WiForce's reader utilizes wideband channel estimates to isolate sensor signal from multipath in doppler domain. Arranging the channels into 'groups' allows to read phase changes across subcarriers to give robust measurements

the transform. This is a reasonable assumption to make, since the sampling is occurring in MHz rate, whereas force will at max change in rate of kHz, since it is a mechanical quantity. However, we can not obviously assume the phase to stay constant forever, and, the phase will change as we apply force on the sensor which would move the shorting points. More importantly, we need to not only tweak the standard doppler transform to respect phase change, we also need to estimate the phase changes in order to estimate the forces. Thus, WiForce designs an algorithm similar to the familiar short-time transforms. The algorithm divides the channel estimates into groups of N_g, referred to as 'phase-groups'. For each phase-group we first take the harmonics FFT as described earlier and obtain two $K \times 1$ vectors from Eqn.3 for FFT frequency f_s, $4f_s$. Assume that there are G such phase groups, i.e. $N = GN_g$. For all the N_g samples of g-th group, $\phi_n^1, \phi_n^2 \approx \phi_g^1 \phi_g^2 \forall n \in \{1,2,\ldots N_g\}$ from the choice of N_g to respect the time it takes for the force to become effective. The output at g-th phase group, k-th subcarrier, after harmonics FFT at $f_s, 4f_s$, is denoted as $P_1[k,g], P_2[k,g]$.

$$P_{\{1,2\}}[k,g] = \tilde{H}[k,\{f_s,4f_s\}] = \alpha_s e^{-j2\pi kF\frac{d_s}{c}} e^{j\phi_g^{\{1,2\}}} \quad (4)$$

To get rid of the air phases, we can obtain the phase change between 2 groups by conjugate multiplication:

$$\tilde{P}_{\{1,2\}}[k,g] = P_{\{1,2\}}[k,g+1] * \text{conj}(P_{\{1,2\}}[k,g]) = \alpha_s^2 e^{j(\phi_{g+1}^{\{1,2\}} - \phi_g^{\{1,2\}})} \quad (5)$$

Hence, we have

$$\angle\tilde{P}_1[k,g] = \phi_{g+1}^1 - \phi_g^1, \angle\tilde{P}_2[k,g] = \phi_{g+1}^2 - \phi_g^2 \quad (6)$$

for each subcarrier k. Observe that the right side of the equation is the phase change independent of k, which entails that we have K independent estimates of the phase change from the K subcarriers. Thus, we can estimate very precise phase changes by averaging over these K independent estimates, allowing WiForce to calculate very precisely the analog phase changes.

The last piece in the puzzle to conclude the design section, is to internalize how can we use differential phase

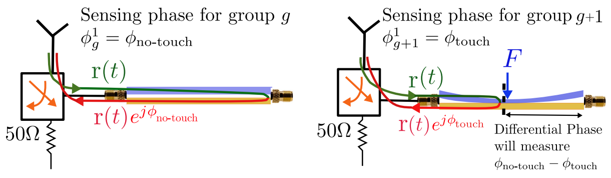

Sensing phase for group g
$\phi^1_g = \phi_{\text{no-touch}}$
$\mathrm{r}(t)$
$\mathrm{r}(t)e^{j\phi_{\text{no-touch}}}$
50Ω
Sensing phase for group $g{+}1$
$\phi^1_{g+1} = \phi_{\text{touch}}$
F
$\mathrm{r}(t)$
$\mathrm{r}(t)e^{j\phi_{\text{touch}}}$
50Ω
Differential Phase will measure $\phi_{\text{no-touch}} - \phi_{\text{touch}}$

Figure 9: Differential Phase measurements can be compensated with fixed quantity $\phi_{\text{no-touch}}$ procured via calibration, to obtain the quantity of interest ϕ_{touch}, which varies with force magnitude and location

to sense and localize the contact forces. In fact, since force is an event based quantity, that is, unlike quantities like temperature, moisture, which sense the ambient quantity, tactile sensors have to sense the force from an 'touch-event' which exerts certain force on the sensor at a given location. When we measure the differential phase between the 'no-touch' and 'touch' events, we can measure differential phase and obtain the absolute phase by simply subtracting the phase which the waves accumulate when the sensor is at rest (Fig. 9). Since this no-touch phase is a fixed quantity and depends only on the length of the trace, we measure it beforehand via a VNA setup and compensate. Hence, compensating the differential phase with the VNA calibrated no-touch phase allows us to recover phases from both the ends, which can then used by the transduction mechanism in order for estimation of force magnitude and location at the reader.

4 Implementation

4.1 Microstrip line RF Interfacing

To support the two broad applications targeted by WiForce, the sensor must give good RF propagation performances at 900 MHz (for in-body sensing applications) and at 2.4 GHz (to be compatible with Wi-Fi/Bluetooth standards). From our simulations (Section 10.2), we expect that by having the ratio between trace width and sensor height to be around 4 : 1, we should get good impedance matching. Hence, we design an air substrate microstrip line with trace width of 2.5 mm, ground trace width of 6 mm and height of 0.63 mm, for a sensor length of 80 mm. To verify the impedance matching, we assess the RF design performance of the sensor in terms of insertion/thru losses. For this, we perform a 2 port amplitude/phase analysis using VNA. As visible in Fig. 10, this leads to a S11/S22 ratio below -10 dB over the entire frequency range from 0 to 3 GHz, along with linear S12 phase, which justifies the broadband nature of the sensor.

4.2 Forming the sensor model with soft beam microstrip line

After having verified the RF properties of the microstrip line, we fabricate the soft layer using Ecoflex 00-30 (a commonly used elastic material [33–35]). The Ecoflex layer is placed onto the top trace to create the WiForce

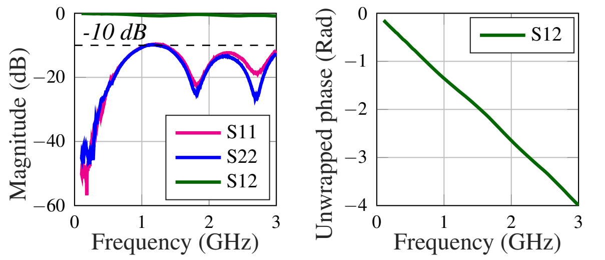


Figure 10: 2 port RF profiling of the sensor. S11 stays below -10 dB across 0-3 GHz, with S12 around 0 dB with linear phase.

sensing surface with thick traces, endowed with the novel phase to force transduction mechanism.

To verify if the sensor is following the transduction mechanism, we exert forces on the sensor at 5 locations, namely 20, 30, 40, 50 and 60 mm (20, 40, 60 marked in Fig. 11). We expect the beam to show a symmetric phase changes on both the ends when tested at the center point, and asymmetric phase changes for the end points, as described in Section 3.1. When pressed on the end points, the port near the pressing location would show more phase change, whereas the other end essentially shows a constant phase as force increases. For this testing, we use the setup visible Fig. 11, where an indenter allows us to apply a force at a given location on the sensor, and a load cell on which the sensor is attached, allows us to collect the values of force magnitudes applied. As seen clearly in the 20/40/60 mm figures in Table 1, the phases do follow the beam bending model as discussed above, since 40 mm testing shows symmetric behaviour, whereas for 20/40 mm, one of the ends show a constant phase as force magnitude is increased.

We now use the data obtained by applying forces at all 5 locations, and compute a cubic-fit to make a sensor model that allows us to compute the force magnitude and force location based on the measured phase changes. To confirm the validity of the model, we asses it at an intermediate point (55 mm), and plot the phase-force profile

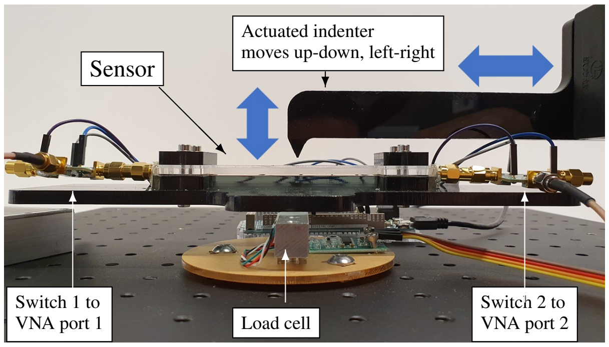


Figure 11: Sensor on the load cell platform, The actuated indenter which can move up-down to exert force on the sensor, as well as left-right, to do so at a particular location, shown via blue arrows . A load cell below the platform and VNA (not shown here) provide force, phase ground truth measurements.

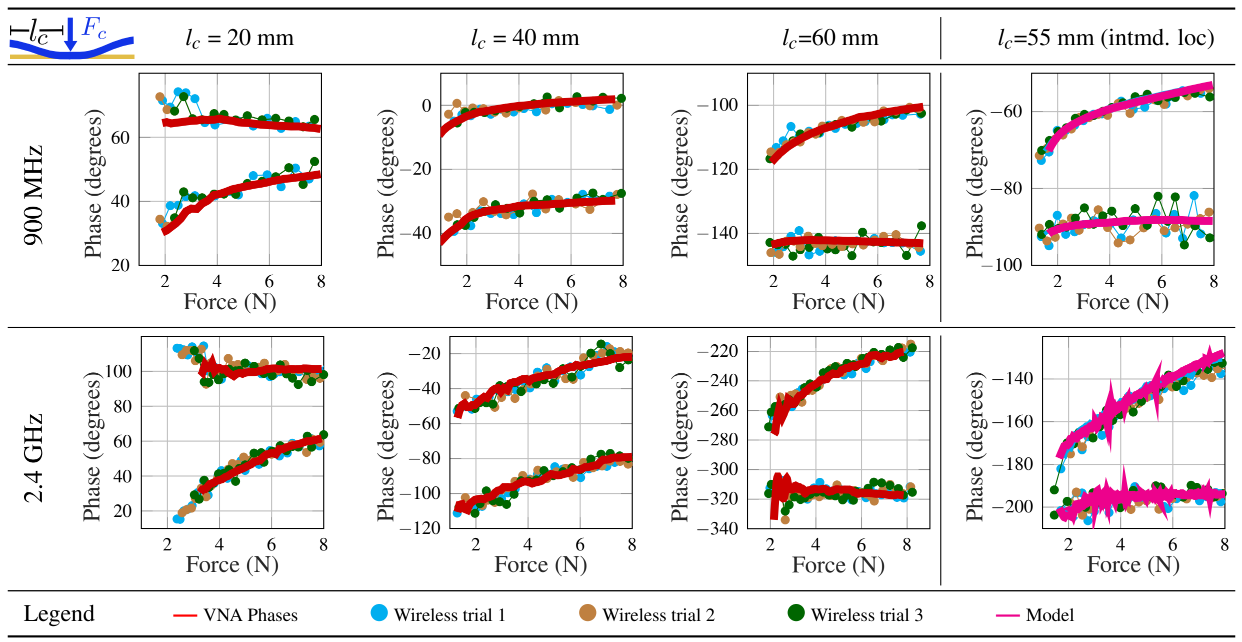


Table 1: Ground truth phase-force profiles (red) measured via the VNA and load cell setup show symmetric phase changes when pressed at center ($l_c = 40$mm, total length is 80mm), and asymmetric phases when pressed at $l_c = 20, 60$mm, as discussed in Section 3.1. We collect VNA data at $l_c = 20, 30, 40, 50, 60$mm, and perform a cubic fit to get the sensor model, which is evaluated by testing at intmd. $l_c = 55$mm. The wireless phase measurements, as well as the model predictions at 55mm consistently overlap with ground truth, warranting the performance of WiForce's design .

as predicted by the model alongside the ground-truth profile we collect from the VNA. As visible in Table 1, all graphs for force applied at 55 mm overlay on each other, which confirms the reliability of the sensor model.

4.3 Clock Design and RF Switches

To encode the phase changes caused by different shorting positions on the microstrip line due to application of a force, we utilize 2 RF-switches with the duty cycled clocking strategy described in Section 3.2. We use the HMC544AE from Analog Devices in our prototype, which is a reflective-open switch consistent with our duty cycling requirements discussed in Section 3.2.

The final component in our prototype design is the clock source. We use the timer channels in Arduino Due with an Atmel SAM3X8E ARM Cortex-M3 processor [36] to generate the duty cycled clock source as described in Section 2. We generate a 25% duty cycled 1 kHz square wave, and a 75% duty cycled 2 kHz square wave to modulate the two RF switches. This gives us interference-free modulation at $1, 4$ kHz.

Hence our sensor prototype consists of five components, shown in Fig. 12b, the microstrip line sensor, 2 RF switches, 2 clock sources, a splitter to combine outputs of the 2 switches and one antenna to communicate the backscaterred phases to the wireless reader. The elements in our design which require power thus are the 2 RF switches and the 2 clock sources. For switching at kHz frequencies, we observed that the power consumed by the 2 HMC544AE switches was almost similar to the static power consumption of 3.3 μW (the static current for switch at 3.3V voltage level is 0.5 μA [37]). Although in our design we use a microcontroller to provide the clocks, by using low-powered oscillators, we can meet the clocking requirements with about 2 μW power budget [38]. Overall, the requirements are lesser than 10μW which are modest enough to be supported by a RF energy harvesting circuit. In recent works, papers have even shown more than 50 μW power being harvested via RF signals, across 1cm of tissue [39].

4.4 Wireless Reader Implementation

The main task of the wireless reader is to transmit the OFDM waveform and periodically estimate the channel, so that phase changes at the shifted frequencies from the sensor can be read wirelessly. To perform the channel estimation, we utilize a 64 subcarrier, 12.5 MHz OFDM waveform. We test this for both center frequency of 900 MHz and 2.4 GHz. We use separate antennas for transmission and reception, and use the same USRP N210 SDR [40][2] for both functions. Since the transmit and receive chains are on the same device, they are synchronized and will not show frequency/phase offsets. We emphasize here that the arduino clock is not synchronized with the other elements of the system since the force sensor is deployed as a separate entity.

We use a 320 sample long OFDM preamble padded with 400 zeros for the channel estimate. At the sam-

[2]We can potentially use a COTS device as well as the wireless reader. Refer to Section 10.1 for a brief discussion on the same

pling rate used, this translates to fresh channel estimates every $T = \frac{720}{12.5\times10^6} = 60\ \mu$s. Recall that to sense harmonics, the maximum harmonic frequency which can be sensed would be $|f_{\max}| = \frac{1}{2T} \approx 8.7$ kHz due to the Nyquist Limit. We therefore chose our sensor clock frequencies to be 1,2 kHz, which would give modulation at 1,4 kHz, and falls comfortably within measurable limits.

5 Experimental Evaluation

Armed with a sensor model to get from phases to force magnitude/location, as well as wireless reader and sensor implementation to enable backscatter sensing capabilities, we now evaluate the wireless performance of our sensor in different indoor environments. The developed sensor model is first used to estimate the force magnitude and location exerted on the sensor, and this predicted force magnitude/location is compared to ground truth readings from the load cell and actuator position, as shown in Fig. 12a. In addition, we plot empirical CDFs, which allow us to understand the accuracy of our sensing solution. Not restricting to over the air evaluations, we even evaluate our sensor when the wireless propagation occurs through tissue phantoms made to emulate human tissues. We also show the capability of reading forces from two sensors simultaneously. Finally, we show that the force sensing works not only with the precision touches of the actuator, but can also detect the force and contact location when a human interacts with the sensor via finger touches.

5.1 Wireless Performance Evaluation

The first step in the wireless performance evaluation is to verify if the estimated force magnitudes and locations agree with the ground truth force-phase curves obtained via the load cell and the VNA setup. For this purpose, the setup illustrated in Fig. 12a is used[3], with the sensor on top of a platform having load cell to give ground truth readings for the experiment, similar to the VNA experiment in Section 4.2. Forces are applied between 0 and 8 N at 20, 40, 55 and 60 mm positions on the sensor. From Table 1, we can clearly see that wireless sensing is able to follow the VNA force-phase curves. Hence, this allows to validate the wireless implementation.

Using the estimated values of force magnitudes and force location to the ground truth, i.e. load cell readings and indenter location, we plot empirical CDFs to evaluate the wireless performance metrics. In Fig. 13a, Fig. 13b, we see that median error of force magnitude estimation of WiForce is 0.56 N when being read at 900 MHz, and 0.34 N when being read at 2.4 GHz. These results are satisfactory, since the errors are a fraction of the operating range of the sensor, which is approximately 8 N. One can observe that the error is lower at high frequency. Since higher frequency signals accumulate more phases for the same travelled distance, the required granularity for phase sensing is more relaxed, leading to lower error than sensing at low frequencies. Another observation from Fig. 13a, Fig. 13b, is that the sensor works uniformly across its length, i.e. error CDFs are similar when plotted for touching at individual locations with increasing magnitude of forces.

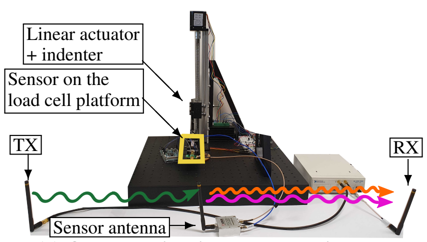
Proceeding similarly, the median errors on the estimated force location is 0.86 mm at 900 MHz, and 0.59 mm at 2.4 GHz, as visible in Fig. 13c. Similar to force magnitude CDFs, performance is better at a higher frequency, since more phase change is accumulated per unit length at higher frequencies, enabling finer location estimation. These location results are satisfactory, with about 5 times higher accuracy than reported in recent work [41,42], where errors are in the order of magnitude of centimeters. The reasons for this improved performance are two-fold. To localize the contact location, WiForce correlates the extra separation between the shorting points caused by sensor bending in the action of a certain contact force. This correlation is enabled by the novel two-ended sensing strategy of WiForce. This is fundamentally very different from the past contact location sensing approaches [41,42]. Furthermore, this is supported by a wideband phase sensing algorithm (Section 3.3), which is capable of sensing these phases very accurately and robustly, unlike the previous works which used a narrowband RFID reader for the evaluations [41].

5.2 Testing with Tissue Phantoms

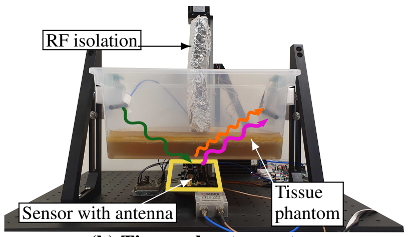
We now assess the performance of our backscatter sensing strategy through human tissue. Propagation through human tissues necessitates using 900 MHz over 2.4GHz, as frequencies higher than 1 GHz are severely attenuated in such environments [43,44]. Wireless signals undergo huge losses when they propagate through human tissue, since these tissues are typically materials with high dielectric constants (with $\varepsilon_r > 10$) [45]. Further, the propagation is hampered via refraction and total internal propagation effects, which exacerbate the losses. Thus, to sense the robustness of our strategy with these impairments, we use the setup visible in Fig. 13d. It consists of a tissue phantom composed of three layers (muscle, fat and skin, and thickness of 25, 10 and 2 mm, respectively), with dielectric properties selected to mimic human tissue properties, as in [46].

During these experiments, we observe that there was around 110 dB two-way backscatter loss from the TX-sensor and sensor-RX, for center frequency 900 MHz, when communicating through the tissue phantom. However, the direct path TX to RX signal had about 10-15 dB loss. The dynamic range of the USRP SDR we use was around 60 dB, because of which we can't decode

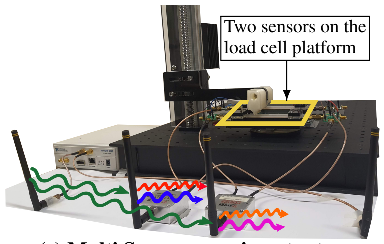
[3]We also evaluate the performance of the sensing algorithm over a range of distances till 2m. The results are presented in Section 10.3



(a) Over the air wireless evaluation setup

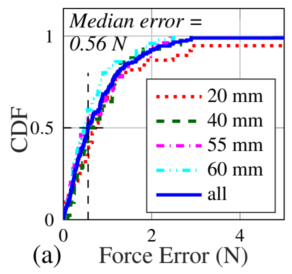


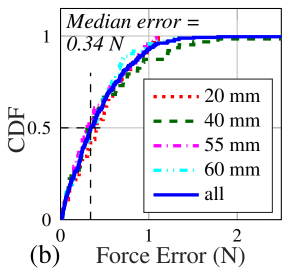
(b) Tissue phantom setup

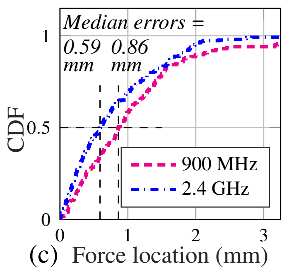


(c) Multi Sensor experiment setup

Figure 12: Different evaluation settings for WiForce's sensor prototype. In (a), the distance between TX-sensor antenna is 50cm, sensor antenna-RX is 50cm. In (b), the antenna and sensor placement ensures that propagation happens through the tissue phantom. In both (a), (b), green: excitation signal, orange,pink: freq. shifted backscattered signal. We also show capability to read from multiple sensors via the setup in (c), here, the red/blue waves represent a different set of frequency shifts for the second sensor







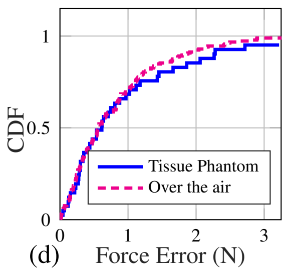


Figure 13: Force CDF Plots, 900 MHz (a), 2.4 GHz (b). For testing at individual points, the error is comparable to the combined CDF for which readings across the length are collected. Hence, the performance of the sensing strategy remains good throughout the sensor length. (c) shows the localization CDF for both the frequencies and (d) compares over the air performance with the tissue phantom setup. Similar CDF plots for both the setups show the sensing robustness when tested with the tissue phantoms

the weak backscattered signal under the presence of the much stronger direct path signal. Hence, for these experiments, we isolated the TX and RX with a metal plate for this experiment. Because of the metal plate, the direct path loss increased to about 60 dB, which allowed us to decode the 50 dB lower backscattered signal at the receiver using the 60 dB dynamic range ADC of the USRP. For this experiment, we apply contact force at 60 mm on the sensor. We obtain similar performance as with the over-the-air tests, with the median force error increasing slightly from 0.56 N to 0.62 N (Fig. 13d). These results demonstrate the robustness of WiForce's wireless capabilities, since the sensing algorithm was able to decode force readings from a weaker signal trough the tissue phantom. In future works, the metal blockage can be replaced by self-interference canceling strategies, however, this is beyond the scope of this paper.

5.3 Multi-sensor experiments

We also evaluate the capability of WiForce to sense from multiple sensors simultaneously. The setup here consists of two sensors placed on a platform, and we use a custom designed indenture with the actuator in order to press on the two sensors simultaneously (Fig. 12c). A load cell is attached below the platform to measure the combined forces acting on the platform (Fig. 14), whereas via wireless sensing from the two sensors we can estimate F_1, F_2 individually. In order to have separate identities for the two ends of the other sensor, we modulate via 1400, 2800 Hz duty cycled waves (visually illustrated via red, blue waves Fig. 12c, Fig 14).

By reading at these frequencies, we can wirelessly obtain estimates $\hat{F_1}, \hat{F_2}$ of F_1, F_2. Because the load cell measures $F_1 + F_2$, we expect that adding these two estimates should allow us to compare against the ground truth load cell readings. The added estimates are expected to give a median error of $1.12N$, since one estimate from the sensor comes with a median error of $0.56N$ at 900 MHz, as profiled by the CDF plots (Fig. 13). Thus, we plot $F_1 + F_2 \pm 1.12N$ as the blue shaded region as the expected median performance of the sensor to sense the added force. Indeed, we see the added up estimates respecting the median error by being confined inside the median error region (Fig. 14).

5.4 Getting More Than Finger Touch: Measuring Fingertip Forces

We now motivate a UI use case, which has the potential to improve and change the way users interact with digital devices. For this purpose, we select a center frequency of 2.4 GHz for our sensor, which is well-adapted to Wi-Fi and Bluetooth devices. To assess the relevance of our sensor for such applications, we use the fingertip, instead of the actuated indenture, to press the sensor with varying force levels. An operator presses the sensor at the 60 mm location. We plot the force readings from the load cell in real time, and use this real-time plot to give the user visual cues to settle in to some force level. Then, we

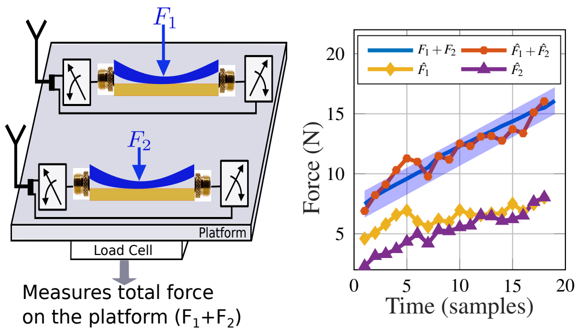
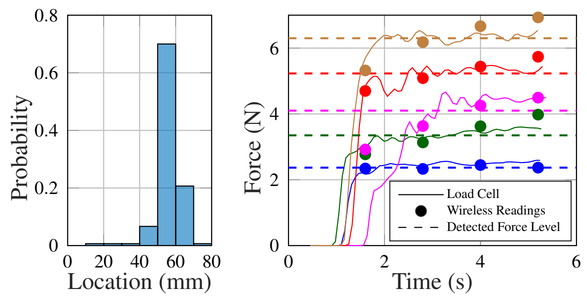


Figure 14: Sensing forces simultaneously from two sensors



(a) Finger touch location histogram.

(b) Finger force level readings vs time.

Figure 15: Wireless Sensing results for pressing at 60 mm with increasing force levels via a fingertip. From (a) we see that all the touch interactions at 60 mm ±20mm were classified correctly, as the sensor was pressed with a finite-width fingertip (about 15-20 mm [47, 48]). From (b), it can be seen that WiForce was able to estimate increasing force levels accurately

estimate these force levels using WiForce to evaluate if it can support these sensing capabilities.

Fig. 15 shows the evaluation results of WiForce's experiments. From Fig. 15a one can see that the sensor could accurately detect the pressing location, which was 60 mm, with sufficient accuracy, considering the fact that a typical human fingertip has a width and thickness of approximately 15-20 mm [47, 48]. That is, even though WiForce's location sensing had sub-mm location sensing accuracy, now the error source will most likely be coming from the uncertainty of how operators press the sensor with their finite width fingertips, instead of the precise point-pressing feature of the actuator before. Since most of the readings are clustered ±20mm near the visual cue of 60mm given to the operators, WiForce does operate under the practical limit of the experimental setting due to finite width of the operators.

Further, in Fig. 15b, we see how WiForce is able to get more than just binary touch sensing results. Not only can WiForce detect the point where a finger touched the sensor, going one step ahead, WiForce is able to detect the force profiles of the touch interactions as well, which motivates much improved UI use cases by getting more than just touch/no touch information.

6 Discussions and Applications

The most natural usecase for such wireless haptic feedback lies in surgical robots and tools. Human hands are extremely dexterous, and provide unparalleled sensory feedback which enable very precise operations required for surgery. However, we need tools and robots to emulate the human hands when direct operation is not possible, such as during minimally invasive surgical operations. Ideally surgeons should receive haptic feedback from the tools/robots they are operating, which would require information of both force magnitude and contact location. However, such haptic feedback is generally not available in practice, and one reason has been that force sensing modalities are still not evolved enough to support these applications [5, 49–52]. Loss of haptic feedback increases the training time for surgeons, increases risk of surgical errors, and hinders the closed-loop operation for robot assisted surgeries [53–56].

The current form factor and sensor interface hinders direct use of WiForce's sensor in more complex surgical tasks requiring force feedback, such as cardiac ablation [55] or pre-retinal membrane peeling [57]. However, the sensor can help solve a major problem in laparoscopy known as the fulcrum effect [58]. The fulcrum effect is caused due to lever effect caused by contact forces between the body and surgical tool, at the entry point of the surgical incision. Due to lack of feedback on both the magnitude of force and location, the tool tend to slip, which causes risks of tissue damage. A laparoscopic surgical tool augmented with a WiForce sensor to determine and localize the contact force can prevent this fulcrum effect since the surgeon can do a closed loop correction based on this haptic feedback.

Apart from surgical applications, sensing contact force and location can be extremely useful for robotic tasks which require a manipulator/gripper. Robotic manipulators need this haptic feedback to determine how firmly they have grasped a particular object [2, 3]. People have attempted doing this via vision induced haptics [59, 60], however, these methods typically require computationally intensive algorithms and fail to meet the required temporal rate of feedback required to determine if the grasp of the object is loosening and slipping [61]. However, since wireless sensing is not bound to such issues, and can be made near real-time. Thus, such sensors can be used for direct and low-latency haptic feedback to improve robotic manipulation operations.

Alongside the robotics centered applications, force sensing can have many latent applications in the next generation interfaces for HCI/AR-VR. Smart surfaces have been an active area of research, with touch sensing touted to a game changer for ubiquitous computing [62–64]. Force sensing will add more depth to these touch sensing solutions, and can lead to some unforeseen applications.

7 Future Work

Extending to 2-D continuum: The current sensor prototype of WiForce consists of sensing on a 1-D continuum. To extend this sensing to a 2-D continuum, we can deploy multiple WiForce sensors placed next to each other. Hence, by reading phase changes from multiple WiForce sensors, we can infer the location and contact force magnitude on the 2-D continuum spanned by these multiple sensors. A hindering factor to this 2-D extension is how to address multiple touch points simultaneously, which will be explored in future works.

Reducing the form factor: WiForce is the first work which presents such a low-powered sensor, and thus naturally leads the way to realize a battery-free haptic feedback. The current sensor prototype of WiForce is 80 mm long, and about 10 mm thick. With the current form factor, the sensor is not directly applicable for some of the medical applications which need smaller sensors. The sensors can get to the correct form-factor requirements by designing integrated circuits, antenna and the sensor fabrication. To make the sensor prototype more flexible, we will explore new fabrication strategies like flexible PCB printing and creating custom RF connectors.

8 Related Work

Force sensors have been developed using a variety of transduction mechanisms, such as capacitive, piezoresistive, piezoelectric, optical, magnetic, and inductive [65]. There are a number of tradeoffs among the various mechanisms, including, for example, sensitivity, spatial resolution, accuracy, power consumption, and size. To meet the requirements of many emerging systems, particularly those where it may be difficult to have a physical wired connection to the sensors, many researchers have been investigating the creation of wireless sensors.

Wireless force sensors: A number of wireless capacitive force sensors that leverage a change in capacitance due to deformation have been recently developed. For example, a flexible capacitive sensor was created for wirelessly measuring strain in tires [66], and a capacitive textile sensor was developed for wireless respiratory monitoring [67]. While the capacitive sensing paradigm can work well for a number of force sensing applications, it is not naturally compatible with wireless sensing. In order to wirelessly transmit force information obtained through capacitive sensing, additional hardware and circuits are needed, complicating the design.

Inductor-capacitor (LC) wireless sensors are passive devices that can remotely sense a number of parameters, including pressure. The working principle of these sensors is based on changes in the capacitance that causes a shift in the LC resonant frequency, which can be wirelessly measured [68, 69]. A number of these LC sensors have been developed for applications like monitoring of pressures during arterial blood flow [70], and the measure of finger tip forces during athletic activities [71]. However, the resonance frequency of these sensors is in the range of a few hundred kHz to a few MHz [69], which makes wireless sensing difficult. As a consequence, these sensors suffer from short interrogation distances in the range of a few centimeters [69, 72].

There has also been a large body of research on strain sensors [35, 73–76]. In strain sensing, instead of sensing the normal transversal force, the longitudinal force is sensed. Longitudinal force has a tendency to stretch and elongate the object it is acting upon, hence these sensors estimate the change in the length to infer strain. Thus, most of the wireless strain sensors exploit the shifts in resonant frequencies to sense strain. Thhat is, to infer strain, a wireless reader evaluates signal strength at multiple frequencies, to estimate the resonant frequency, where a notch will form in the signal strength measurements. It is well known that signal strength is a fickle quantity easily corrupted by multipath. Indeed, most of these works show evaluations in a controlled, anechoic environment, and the technology has not been found to be robust to static multipath [74].

Backscatter sensing systems: Recent advancements in 'backscatter sensing' has enabled the creation of passive, battery-free touch interfaces. Touch sensing has been a well explored use case of RFID-based sensing [41, 42, 77–82]. IDSense [82] utilized the fact that reflected RSS and phase change in a unique way when the RFID chip is touched, and following up on this PaperId [81] even gave a simple manufacturing method by which one could simply use an inkjet printer to manufacture these RFID tags and augment everyday objects with touch interactions. RIO [41] further explored the touch to reflected signal phase mapping to extend touch sensing to multiple RFID tags by utilizing mutual coupling effects, and extended the design further to use custom designed, application specific RFID tags. Livetag [42] presented a similar touch sensing system showing how to sense these touch interactions using Wi-Fi based readers, instead of relying on expensive, dedicated RFID readers used by earlier works. However, none of these systems could sense force magnitude and were limited to sensing just the position where the tag was being touched in order to sense simple gestures/sliding movements etc.

9 Acknowledgements

We thank anonymous reviewers and our shepherd, Prof. Haitham Hassanieh, for their insightful feedback. We also thank members of WCSNG, UCSD for their feedback throughout the process, as well as analog devices for providing us HMC544AE samples. This research was supported in part by the National Science Foundation under Grant 1935329.

References

[1] Victoria E Abraira and David D Ginty. The sensory neurons of touch. *Neuron*, 79(4):618–639, 2013.

[2] Aude Billard and Danica Kragic. Trends and challenges in robot manipulation. *Science*, 364(6446), 2019.

[3] Zhen Deng, Yannick Jonetzko, Liwei Zhang, and Jianwei Zhang. Grasping force control of multi-fingered robotic hands through tactile sensing for object stabilization. *Sensors*, 20(4):1050, 2020.

[4] Yousef Al-Handarish, Olatunji Mumini Omisore, Tobore Igbe, Shipeng Han, Hui Li, Wenjing Du, Jinjie Zhang, and Lei Wang. A survey of tactile-sensing systems and their applications in biomedical engineering. *Advances in Materials Science and Engineering*, 2020, 2020.

[5] Javad Dargahi, Saeed Sokhanvar, Siamak Najarian, and Siamak Arbatani. *Tactile Sensing and Displays: Haptic Feedback for Minimally Invasive Surgery and Robotics*. John Wiley & Sons, 2012.

[6] Jacqueline K Koehn and Katherine J Kuchenbecker. Surgeons and non-surgeons prefer haptic feedback of instrument vibrations during robotic surgery. *Surgical endoscopy*, 29(10):2970–2983, 2015.

[7] Hedan Bai, Shuo Li, Jose Barreiros, Yaqi Tu, Clifford R Pollock, and Robert F Shepherd. Stretchable distributed fiber-optic sensors. *Science*, 370(6518):848–852, 2020.

[8] Minglu Zhu, Zhongda Sun, Zixuan Zhang, Qiongfeng Shi, Tianyiyi He, Huicong Liu, Tao Chen, and Chengkuo Lee. Haptic-feedback smart glove as a creative human-machine interface (hmi) for virtual/augmented reality applications. *Science Advances*, 6(19):eaaz8693, 2020.

[9] Wang Dangxiao, Guo Yuan, Liu Shiyi, Yuru Zhang, Xu Weiliang, and Xiao Jing. Haptic display for virtual reality: progress and challenges. *Virtual Reality & Intelligent Hardware*, 1(2):136–162, 2019.

[10] Hyosang Lee, Kyungseo Park, Jung Kim, and Katherine J Kuchenbecker. Internal array electrodes improve the spatial resolution of soft tactile sensors based on electrical resistance tomography. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 5411–5417. IEEE, 2019.

[11] Kyungseo Park, Hyunkyu Park, Hyosang Lee, Sungbin Park, and Jung Kim. An ert-based robotic skin with sparsely distributed electrodes: Structure, fabrication, and dnn-based signal processing. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1617–1624. IEEE, 2020.

[12] Hyosang Lee, Hyunkyu Park, Gokhan Serhat, Huanbo Sun, and Katherine J Kuchenbecker. Calibrating a soft ert-based tactile sensor with a multi-physics model and sim-to-real transfer learning. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1632–1638. IEEE, 2020.

[13] Ravinder S Dahiya, Philipp Mittendorfer, Maurizio Valle, Gordon Cheng, and Vladimir J Lumelsky. Directions toward effective utilization of tactile skin: A review. *IEEE Sensors Journal*, 13(11):4121–4138, 2013.

[14] Mahesh Soni and Ravinder Dahiya. Soft eskin: distributed touch sensing with harmonized energy and computing. *Philosophical Transactions of the Royal Society A*, 378(2164):20190156, 2020.

[15] Ilya Rosenberg and Ken Perlin. The unmousepad: an interpolating multi-touch force-sensing input pad. In *ACM SIGGRAPH 2009 papers*, pages 1–9. 2009.

[16] Liang Zou, Chang Ge, Z Jane Wang, Edmond Cretu, and Xiaoou Li. Novel tactile sensor technology and smart tactile sensing systems: A review. *Sensors*, 17(11):2653, 2017.

[17] Patrick Parzer, Kathrin Probst, Teo Babic, Christian Rendl, Anita Vogl, Alex Olwal, and Michael Haller. Flextiles: a flexible, stretchable, formable, pressure-sensitive, tactile input sensor. In *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, pages 3754–3757, 2016.

[18] Patrick Parzer, Florian Perteneder, Kathrin Probst, Christian Rendl, Joanne Leong, Sarah Schuetz, Anita Vogl, Reinhard Schwoediauer, Martin Kaltenbrunner, Siegfried Bauer, et al. Resi: a highly flexible, pressure-sensitive, imperceptible textile interface based on resistive yarns. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, pages 745–756, 2018.

[19] Andreas Pointner, Thomas Preindl, Sara Mlakar, Roland Aigner, and Michael Haller. Knitted resi: A highly flexible, force-sensitive knitted textile based on resistive yarns. In *ACM SIGGRAPH 2020 Emerging Technologies*, pages 1–2. 2020.

[20] Hyosang Lee, Kyungseo Park, Yunjoo Kim, and Jung Kim. Durable and repairable soft tactile skin for physical human robot interaction. In *Proceedings of the Companion of the 2017 ACM/IEEE International Conference on Human-Robot Interaction*, pages 183–184, 2017.

[21] Jessica Burgner-Kahrs, D Caleb Rucker, and Howie Choset. Continuum robots for medical applications: A survey. *IEEE Transactions on Robotics*, 31(6):1261–1280, 2015.

[22] Axel Antoine, Sylvain Malacria, and Géry Casiez. Forceedge: controlling autoscroll on both desktop and mobile computers using the force. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 3281–3292, 2017.

[23] David Silvera-Tawil, David Rye, Manuchehr Soleimani, and Mari Velonaki. Electrical impedance tomography for artificial sensitive robotic skin: A review. *IEEE Sensors Journal*, 15(4):2001–2016, 2014.

[24] Po-Han Peter Wang, Chi Zhang, Hongsen Yang, Dinesh Bharadia, and Patrick P Mercier. 20.1 a 28μw iot tag that can communicate with commodity wifi transceivers via a single-side-band qpsk backscatter communication technique. In *2020 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 312–314. IEEE, 2020.

[25] Pengyu Zhang, Dinesh Bharadia, Kiran Joshi, and Sachin Katti. Hitchhike: Practical backscatter using commodity wifi. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*, SenSys '16, page 259–271, New York, NY, USA, 2016. Association for Computing Machinery.

[26] C. Girerd, Q. Zhang, A. Gupta, M. Dunna, D. Bharadia, and T. K. Morimoto. Towards a wireless force sensor based on wave backscattering for medical applications. *IEEE Sensors Journal*, pages 1–1, 2021.

[27] Zhihong Luo, Qiping Zhang, Yunfei Ma, Manish Singh, and Fadel Adib. 3d backscatter localization for fine-grained robotics. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 765–782, 2019.

[28] Colleen Josephson, Bradley Barnhart, Sachin Katti, Keith Winstein, and Ranveer Chandra. Rf soil moisture sensing via radar backscatter tags. *arXiv preprint arXiv:1912.12382*, 2019.

[29] Pengyu Zhang, Colleen Josephson, Dinesh Bharadia, and Sachin Katti. Freerider: Backscatter communication using commodity radios. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '17, page 389–401, New York, NY, USA, 2017. Association for Computing Machinery.

[30] Anderson Maciel, Tansel Halic, Zhonghua Lu, Luciana P Nedel, and Suvranu De. Using the physx engine for physics-based virtual surgery with force feedback. *The International Journal of Medical Robotics and Computer Assisted Surgery*, 5(3):341–353, 2009.

[31] Christopher R Wagner, Robert D Howe, and Nicholas Stylopoulos. The role of force feedback in surgery: analysis of blunt dissection. In *Haptic Interfaces for Virtual Environment and Teleoperator Systems, International Symposium on*, pages 73–73. Citeseer, 2002.

[32] Dangxiao Wang, Meng Song, Afzal Naqash, Yukai Zheng, Weiliang Xu, and Yuru Zhang. Toward whole-hand kinesthetic feedback: A survey of force feedback gloves. *IEEE transactions on haptics*, 12(2):189–204, 2018.

[33] Yong-Lae Park, Bor-Rong Chen, and Robert J Wood. Design and fabrication of soft artificial skin using embedded microchannels and liquid conductors. *IEEE Sensors journal*, 12(8):2711–2718, 2012.

[34] Yahya Elsayed, Augusto Vincensi, Constantina Lekakou, Tao Geng, CM Saaj, Tommaso Ranzani, Matteo Cianchetti, and Arianna Menciassi. Finite element analysis and design optimization of a pneumatically actuating silicone module for robotic surgery applications. *Soft Robotics*, 1(4):255–262, 2014.

[35] Lijun Teng, Kewen Pan, Markus P Nemitz, Rui Song, Zhirun Hu, and Adam A Stokes. Soft radio-frequency identification sensors: Wireless long-range strain sensors using radio-frequency identification. *Soft robotics*, 6(1):82–94, 2019.

[36] Atmel SAM3X8E ARM Cortex-M3. https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-11057-32-bit-Cortex-M3-Microcontroller-SAM3X-SAM3A_Datasheet.pdf.

[37] Analog Devices. Hmc544ae. https://www.analog.com/media/en/technical-documentation/data-sheets/hmc544ae.pdf.

[38] Aatmesh Shrivastava and Benton H Calhoun. A 150nw, 5ppm/o c, 100khz on-chip clock source for ultra low power socs. In *Proceedings of the IEEE 2012 Custom Integrated Circuits Conference*, pages 1–4. IEEE, 2012.

[39] Farah Laiwalla, Jihun Lee, Ah-Hyoung Lee, Ethan Mok, Vincent Leung, Steven Shellhammer, Yoon-Kyu Song, Lawrence Larson, and Arto Nurmikko. A distributed wireless network of implantable sub-mm cortical microstimulators for brain-computer interfaces. In *2019 41st Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, pages 6876–6879. IEEE, 2019.

[40] a National Instruments Company Ettus Research. USRP: Universal software defined radio peripheral networked series. https://www.ettus.com/product-categories/usrp-networked-series/. Accessed: 2020-09-01.

[41] Swadhin Pradhan, Eugene Chai, Karthikeyan Sundaresan, Lili Qiu, Mohammad A Khojastepour, and Sampath Rangarajan. Rio: A pervasive rfid-based touch gesture interface. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, pages 261–274, 2017.

[42] Chuhan Gao, Yilong Li, and Xinyu Zhang. Livetag: Sensing human-object interaction through passive chipless wifi tags. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 533–546, 2018.

[43] Sandeep KS Gupta, Suresh Lalwani, Yashwanth Prakash, E Elsharawy, and Loren Schwiebert. Towards a propagation model for wireless biomedical applications. In *IEEE International Conference on Communications, 2003. ICC'03.*, volume 3, pages 1993–1997. IEEE, 2003.

[44] Ilka Dove. Analysis of radio propagation inside the human body for in-body localization purposes. Master's thesis, University of Twente, 2014.

[45] Deepak Vasisht, Guo Zhang, Omid Abari, Hsiao-Ming Lu, Jacob Flanz, and Dina Katabi. In-body backscatter communication and localization. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 132–146, 2018.

[46] A. Sani, M. Rajab, R. Foster, and Y. Hao. Antennas and propagation of implanted rfids for pervasive healthcare applications. *Proceedings of the IEEE*, 98(9):1648–1655, 2010.

[47] Kiran Dandekar, Balasundar I Raju, and Mandayam A Srinivasan. 3-d finite-element models of human and monkey fingertips to investigate the mechanics of tactile sense. *J. Biomech. Eng.*, 125(5):682–691, 2003.

[48] Eric M Young, David Gueorguiev, Katherine J Kuchenbecker, and Claudio Pacchierotti. Compensating for fingertip size to render tactile cues more accurately. *IEEE transactions on haptics*, 13(1):144–151, 2020.

[49] Nazim Haouchine, Winnie Kuang, Stephane Cotin, and Michael Yip. Vision-based force feedback estimation for robot-assisted surgery using instrument-constrained biomechanical three-dimensional maps. *IEEE Robotics and Automation Letters*, 3(3):2160–2165, 2018.

[50] Angelica I Aviles, Samar Alsaleh, Pilar Sobrevilla, and Alicia Casals. Sensorless force estimation using a neuro-vision-based approach for robotic-assisted surgery. In *2015 7th International IEEE/EMBS Conference on Neural Engineering (NER)*, pages 86–89. IEEE, 2015.

[51] Allison M Okamura. Haptic feedback in robot-assisted minimally invasive surgery. *Current opinion in urology*, 19(1):102, 2009.

[52] Carol E Reiley, Takintope Akinbiyi, Darius Burschka, David C Chang, Allison M Okamura, and David D Yuh. Effects of visual force feedback on robot-assisted surgical task performance. *The Journal of thoracic and cardiovascular surgery*, 135(1):196–202, 2008.

[53] Shanu N Kothari, Brian J Kaplan, Eric J DeMaria, Timothy J Broderick, and Ronald C Merrell. Training in laparoscopic suturing skills using a new computer-based virtual reality simulator (mist-vr) provides results comparable to those with an established pelvic trainer system. *Journal of laparoendoscopic & advanced surgical techniques*, 12(3):167–173, 2002.

[54] M Zhou, S Tse, A Derevianko, DB Jones, SD Schwaitzberg, and CGL Cao. Effect of haptic feedback in laparoscopic surgery skill acquisition. *Surgical endoscopy*, 26(4):1128–1134, 2012.

[55] Michael Yip and David Camarillo. Model-less hybrid position/force control: a minimalist approach for continuum manipulators in unknown, constrained environments. *IEEE Robotics and Automation Letters*, 2016.

[56] Gregory Tholey, Jaydev P Desai, and Andres E Castellanos. Force feedback plays a significant role in minimally invasive surgery: results and analysis. *Annals of surgery*, 241(1):102, 2005.

[57] Anibal Francone, Jason Mingyi Huang, Ji Ma, Tsu-Chin Tsao, Jacob Rosen, and Jean-Pierre Hubschman. The effect of haptic feedback on efficiency and safety during preretinal membrane peeling simulation. *Translational vision science & technology*, 8(4):2–2, 2019.

[58] Ilana Nisky, Felix Huang, Amit Milstein, Carla M Pugh, Ferdinando A Mussa-Ivaldi, and Amir Karniel. Perception of stiffness in laparoscopy–the fulcrum effect. *Studies in health technology and informatics*, 173:313, 2012.

[59] Wenzhen Yuan, Siyuan Dong, and Edward H Adelson. Gelsight: High-resolution robot tactile sensors for estimating geometry and force. *Sensors*, 17(12):2762, 2017.

[60] Yu She, Sandra Q Liu, Peiyu Yu, and Edward Adelson. Exoskeleton-covered soft finger with vision-based proprioception and tactile sensing. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 10075–10081. IEEE, 2020.

[61] Wei Chen, Heba Khamis, Ingvars Birznieks, Nathan F Lepora, and Stephen J Redmond. Tactile sensors for friction estimation and incipient slip detection—toward dexterous robotic manipulation: A review. *IEEE Sensors Journal*, 18(22):9049–9064, 2018.

[62] Hiroshi Ishii. The tangible user interface and its evolution. *Communications of the ACM*, 51(6):32–36, 2008.

[63] Yang Zhang, Yasha Iravantchi, Haojian Jin, Swarun Kumar, and Chris Harrison. Sozu: Self-powered radio tags for building-scale activity sensing. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*, pages 973–985, 2019.

[64] Chuhan Gao, Yilong Li, and Xinyu Zhang. Livetag: Sensing human-object interaction through passive chipless wi-fi tags. *GetMobile: Mobile Computing and Communications*, 22(3):32–35, 2019.

[65] Cheng Chi, Xuguang Sun, Ning Xue, Tong Li, and Chang Liu. Recent progress in technologies for tactile sensors. *Sensors*, 18(4):948, 2018.

[66] Ryosuke Matsuzaki and Akira Todoroki. Wireless flexible capacitive sensor based on ultra-flexible epoxy resin for strain measurement of automobile tires. *Sensors and Actuators A: Physical*, 140(1):32–42, 2007.

[67] T Hoffmann, B Eilebrecht, and S Leonhardt. Respiratory monitoring system on the basis of capacitive textile force sensors. *IEEE sensors journal*, 11(5):1112–1119, 2010.

[68] Chen Li, Qiulin Tan, Pinggang Jia, Wendong Zhang, Jun Liu, Chenyang Xue, and Jijun Xiong. Review of research status and development trends of wireless passive lc resonant sensors for harsh environments. *Sensors*, 15(6):13097–13109, 2015.

[69] Qing-An Huang, Lei Dong, and Li-Feng Wang. Lc passive wireless sensors toward a wireless sensing platform: status, prospects, and challenges. *Journal of Microelectromechanical Systems*, 25(5):822–841, 2016.

[70] Clementine M Boutry, Levent Beker, Yukitoshi Kaizawa, Christopher Vassos, Helen Tran, Allison C Hinckley, Raphael Pfattner, Simiao Niu, Junheng Li, Jean Claverie, et al. Biodegradable and flexible arterial-pulse sensor for the wireless monitoring of blood flow. *Nature biomedical engineering*, 3(1):47–57, 2019.

[71] Baoqing Nie, Ting Yao, Yiqiu Zhang, Jian Liu, and Xinjian Chen. A droplet-based passive force sensor for remote tactile sensing applications. *Applied Physics Letters*, 112(3):031904, 2018.

[72] Marco Baù, Marco Demori, Marco Ferrari, and Vittorio Ferrari. Contactless readout of passive lc sensors with compensation circuit for distance-independent measurements. In *Multidisciplinary Digital Publishing Institute Proceedings*, volume 2, page 842, 2018.

[73] Haining Li, Jiexiong Ding, Siqi Wei, Bowen Chu, Lichao Guan, Li Du, Guangmin Liu, and Jianguo He. A miniature layered saw contact stress sensor for operation in cramped metallic slits. *Instruments and Experimental Techniques*, 61(4):610–617, 2018.

[74] Xiaohua Yi, Terence Wu, Yang Wang, Roberto T Leon, Manos M Tentzeris, and Gabriel Lantz. Passive wireless smart-skin sensor using rfid-based folded patch antennas. *International Journal of Smart and Nano Materials*, 2(1):22–38, 2011.

[75] Trang T Thai, Herve Aubert, Patrick Pons, Manos M Tentzeris, and Robert Plana. Design of a highly sensitive wireless passive rf strain transducer. In *2011 IEEE MTT-S International Microwave Symposium*, pages 1–4. IEEE, 2011.

[76] JR Humphries and DC Malocha. Passive, wireless saw ofc strain sensor. In *2012 IEEE International Frequency Control Symposium Proceedings*, pages 1–6. IEEE, 2012.

[77] Nicolai Marquardt, Alex S Taylor, Nicolas Villar, and Saul Greenberg. Rethinking rfid: awareness and control for interaction with rfid systems. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2307–2316, 2010.

[78] Alanson P Sample, Daniel J Yeager, and Joshua R Smith. A capacitive touch interface for passive rfid tags. In *2009 IEEE International Conference on RFID*, pages 103–109. IEEE, 2009.

[79] Timothy M Simon, Bruce H Thomas, Ross T Smith, and Mark Smith. Adding input controls and sensors to rfid tags to support dynamic tangible user interfaces. In *Proceedings of the 8th International Conference on Tangible, Embedded and Embodied Interaction*, pages 165–172, 2014.

[80] Albrecht Schmidt, H-W Gellersen, and Christian Merz. Enabling implicit human computer interaction: a wearable rfid-tag reader. In *Digest of Papers. Fourth International Symposium on Wearable Computers*, pages 193–194. IEEE, 2000.

[81] Hanchuan Li, Eric Brockmeyer, Elizabeth J Carter, Josh Fromm, Scott E Hudson, Shwetak N Patel, and Alanson Sample. Paperid: A technique for drawing functional battery-free wireless interfaces on paper. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pages 5885–5896, 2016.

[82] Hanchuan Li, Can Ye, and Alanson P Sample. Idsense: A human object interaction detection system based on passive uhf rfid. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pages 2555–2564, 2015.

[83] Quantenna. QSR10GU-AX. http://www.quantenna.com/wp-content/uploads/2018/04/QSR10GU-AX-V1.1.pdf.

[84] Eugenio Magistretti, Krishna Kant Chintalapudi, Bozidar Radunovic, and Ramachandran Ramjee. Wifi-nano: Reclaiming wifi efficiency through 800 ns slots. In *Proceedings of the 17th annual international conference on Mobile computing and networking*, pages 37–48, 2011.

[85] Manikanta Kotaru and Sachin Katti. Position tracking for virtual reality using commodity wifi. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 68–78, 2017.

[86] Yaxiong Xie, Jie Xiong, Mo Li, and Kyle Jamieson. md-track: Leveraging multi-dimensionality for passive indoor wi-fi tracking. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–16, 2019.

[87] Michael Steer. *Microwave and RF design*. NC State University, 2019.

10 Appendix

10.1 Replacing SDR with COTS

We can read the sensor using a COTS-WiFi implementation which can provide CSI, like Quantenna [83]. With COTS devices, periodic channel estimates can be obtained with similar time latency as compared to our implementation using SDRs. In our implementation, $T = 60\mu s$ (Section 4.4), which is reasonably higher than packet sizes of $12\mu s$ achievable by 1 Gbps WiFi systems [84]. A potential issue could be MAC overheads, as also cited in [84], however, we can alter the packet structure slightly to avoid backoffs which can mitigate against the MAC overheads. However, with COTS devices, we will have to deal with CFO effects in the measured channel response.

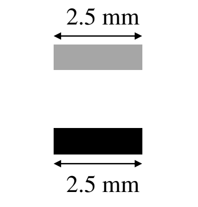
In our implementation on USRP, we had TX and RX sharing the same RF chain. With COTS devices like quantenna, we will have TX and RX as separate devices which might not be able to share a clock, thus leading to frequency and phase offsets. Although WiForce design is robust to phase offsets due to our differential phase sensing approach, we would need to counter CFO. To counter the CFO effects, we can use the fact that CFO will remain same for both the direct path between TX and RX, and the reflected signal from the sensor. To do so, we can consider a differential sensing approach by calculating phase relative to the direct path, and similar approaches have been explored to do so in past work [85, 86]

10.2 HFSS Simulations

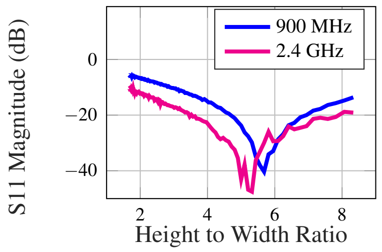
For an air substrate microstrip transmission line, we have the following equation which governs the impedance matching, $Z = 60 \ln \left[\frac{6h}{w} + \sqrt{1 + \left(\frac{2h}{w} \right)^2} \right]$, where h is the vertical separation between signal and ground trace, and w is the width of the signal trace [87].

Setting $Z = 50\,\Omega$ in the above equation, gives us the operating $\frac{h}{w}$ ratio to be approximately $5:1$. In order to interface SMA connectors to the air-substrate microstrip line designed, we have to increase the width of the ground trace so that the bottom legs of the SMA connector can be soldered directly to the ground trace.

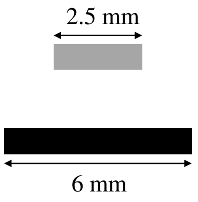
However, we notice some deviation from this ratio when the width of ground trace is increased to allow for easier interfacing with SMA connectors. We simulate the sensor in Ansys HFSS (Fig. 16) to determine this deviation, and observe that the ideal operating ratio shifts to about 4:1 instead of 5:1 when the width of ground trace is increased.



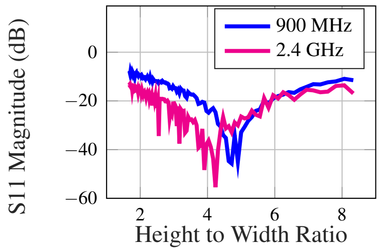
(a) Signal (gray) and Ground (black) traces



(b) Insertion Loss optimal near 5:1 ratio



(c) Signal (gray) and Ground (black) traces

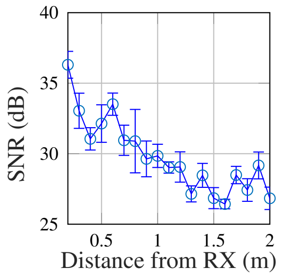


(d) Insertion Loss optimal near 4:1 ratio

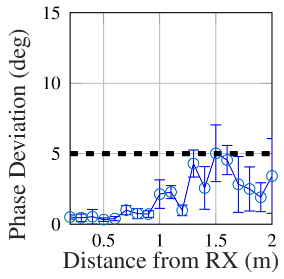
Figure 16: HFSS simulation results: as the ground layer width is increased to allow for easier interfacing with the SMA connector, the ideal height:width ratio decreases from 5:1 to 4:1.

10.3 Performance with distance

We also evaluate our sensor and wireless reader design over a range of distances. For this experiment, we place the TX antenna, sensor antenna and RX antenna along a straight line. The TX antenna is placed 4 m away from the RX antenna, and the sensor is moved from the midpoint, which is 2 m away from both to distances, closer to the RX antenna, and farther away from the TX antenna. The TX power is set to 10 dBm, and the center frequency for this experiment was 900 MHz. We can observe that the sensor gives accurate and satisfying phase stability of less than 1^o even at a distance of 1 m, 3 m from the RX/TX, and acceptable within 5^o performance even at the worst 2 m, 2 m distance from the TX/RX. These operating distances are comparable with previously shown evaluations with RFID based backscatter at 900 MHZ [41], which tested sensing at a maximum distance of 2 m from the RFID reader.



(a) Sensor SNR



(b) Sensor Phase Std. Deviation

Figure 17: Testing WiForce over a range of distances

MAVL: Multiresolution Analysis of Voice Localization

Mei Wang[*], Wei Sun[*], Lili Qiu
The University of Texas at Austin

Abstract

The ability for a smart speaker to localize a user based on his/her voice opens the door to many new applications. In this paper, we present a novel system, MAVL, to localize human voice. It consists of three major components: (i) We first develop a novel multi-resolution analysis to estimate the AoA of time-varying low-frequency coherent voice signals coming from multiple propagation paths; (ii) We then automatically estimate the room structure by emitting acoustic signals and developing an improved 3D MUSIC algorithm; (iii) We finally re-trace the paths using the estimated AoA and room structure to localize the voice. We implement a prototype system using a single speaker and a uniform circular microphone array. Our results show that it achieves median errors of 1.49^o and 3.33^o for the top two AoAs estimation and achieves median localization errors of $0.31m$ in line-of-sight (LoS) cases and $0.47m$ in non-line-of-sight (NLoS) cases.

1 Introduction

Motivation: The popularity of smart speakers has grown exponentially over the past few years due to the increasing penetration of IoT devices, voice commerce, and improved Internet connectivity. The global smart speaker market is estimated to grow at a rate of 21.12% annually and reach 19.91 billion US dollars in 2024.

The ability to localize human voice benefits smart speakers in many ways. First, knowing the user's location allows the smart speaker to beamform its transmission to the user so that it can both hear from and transmit to a faraway user. Second, the user location gives context information, which can help us better interpret the user's intent. For example, as shown in Figure 1, when the user issues the command to turn on the light, the smart speaker can resolve the ambiguity and tell which light to turn on depending on the user's location. In addition, knowing the location also enables location based services. For instance, a smart speaker can automatically adjust the temperature and lighting condition near the user. Moreover, location information can also help with speech recognition and natural language processing by providing important context information. For example, when a user says "orange" in the kitchen, it knows that refers to a fruit; when the same user says "orange" elsewhere, it may interpret that as a color.

[*]Both authors contributed equally to this work

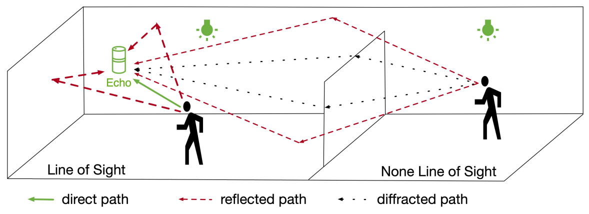


Figure 1: Illustration of application for MAVL under multiple coherent incoming paths in LoS and NLoS scenarios.

There have been a number of interesting works on motion tracking and localization using audio [23, 25, 29, 32, 41, 44, 50, 52], RF [43, 45, 48] and vision-based schemes [8, 53], etc. Cameras cannot be deployed everywhere at home for privacy concerns. Device-based tracking requires carrying a device, which is not convenient for people at home. Device-free RF is interesting, but requires large bandwidth, many antennas, or mmWave chips to achieve high accuracy, which is not easy to deploy at home. Meanwhile, acoustic-based tracking has also been shown to achieve high accuracy. In the past few years, acoustic tracking accuracy has improved from centimeter level [50] to millimeter level [23, 29, 41, 44]. These works focus on tracking users by emitting specially designed acoustic signals. These signals are mostly in inaudible frequency range $16kHz$-$22kHz$.

Challenges: Despite significant acoustic based tracking works, localizing human voice poses new challenges:

- Many of the existing systems require transmission of known signals (*e.g.*, chirps, OFDM symbols, sine waves).

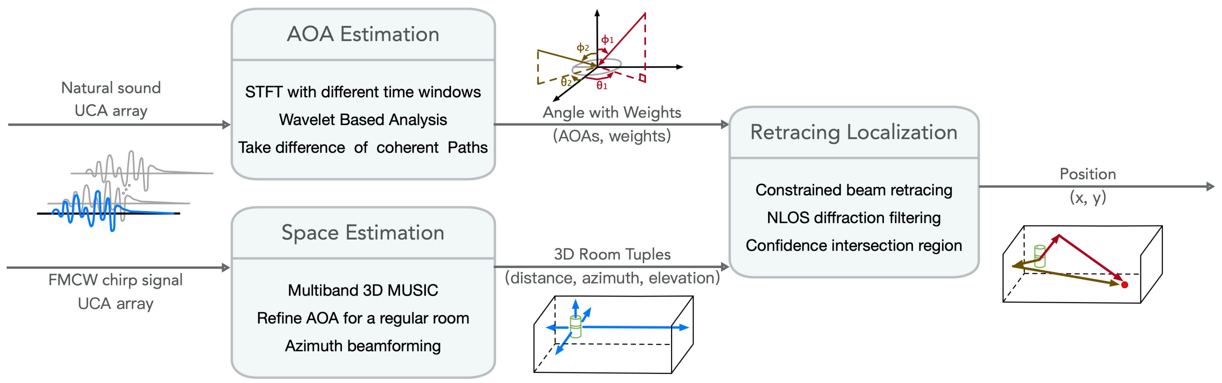


Figure 2: MAVL system involves a three-step process. (1) estimate AoA from multiple paths, (2) recover room structure by actively emitting wideband chirps, (3) localize the voice by retracing back the estimated AoA based on room structure.

In comparison, we can neither control nor predict users' voice signals, including their timing, frequency, and content. This makes it challenging to apply traditional channel estimation and distance estimation based methods.

- In order to localize a user, we need to estimate angle of arrival (AoA) of multiple propagation paths so that we can trace back these paths to localize the user. The signals traversing via multipath are coherent, which significantly degrades the accuracy of AoA estimation methods (*e.g.*, MUSIC requires all signals be independent).

- To enable retracing the location using multiple AoAs, we also need to estimate the indoor environment first. However, depth sensors are not widely deployed at home and vision-based approaches raise privacy concerns.

- The user may not be in line of sight (LoS) from the smart speaker (*e.g.*, the user is behind a wall or in a different room). Localizing the user in NLoS using acoustic signals remains an open problem due to low SNR and detoured propagation paths.

Our approach: In this paper, we build a novel indoor voice localization system, MAVL, by retracing multiple propagation paths that the user's sound traverses, as shown in Figure 2. First, we estimate AoAs of the multiple paths traversed by the voice signals from the user to the microphone array on the smart speaker. The multipath may include the direct path (if available) and the reflected paths. Second, we estimate the indoor space structure (*e.g.*, walls, ceilings) by emitting wideband chirps to estimate the AoA and distance to the reflectors in the room (*e.g.*, walls). Third, we re-trace the propagation paths based on the estimated AoA of the voice signals and the room structure to localize the voice.

We choose AoA since it eliminates the need of distance estimation, which is challenging when we do not know the transmission signals. We use a microphone array widely available on a smart speaker to collect the received signals. While there have been many AoA algorithms proposed, the low frequency of voice signals and the presence of coherent paths pose significant new challenges. To reduce coherence and separate paths, we capture the voice signal that finishes fast so that the signal traversing via the shortest path has small or no overlap with those traversing via the longer paths. We cannot control how many words a user speaks. Instead, we could select the voice signals that occupy some frequencies for a short time period. This requires good time and good frequency resolution. Since there is no single method that can simultaneously achieve good time and good frequency resolutions, we perform wavelet and STFT analyses over different time windows to benefit from both transient signals with low coherence and long signals with high cumulative energy. We further use differencing to cancel the signals in the time-frequency domain to reduce coherence, thereby improving the AoA accuracy.

Next we need to estimate the room contour, *i.e.*, the distances and direction of the walls. Researchers have used depth sensors [2, 15, 31], cameras [9, 18, 21] or multiple sensors [7, 12, 49] to estimate the indoor room contour. However, these systems require extra sensors and some need significant computation cost. It also raises significant privacy concerns. Acoustic has been applied to image the shape of objects [20, 24, 47]. It is promising to use acoustic signals to capture the room contour. Our system emits wide-band Frequency Modulated Continuous Waves (FMCW) chirps and propose the wide-band 3D MUSIC to estimate multiple propagation paths simultaneously. The wide bandwidth not only improves distance resolution, but also allows us to leverage the frequency diversity to estimate AoAs of coherent signals. We improve the AoA estimation by leveraging the assumption of a rectangle room (which is common in real world scenarios), and improve the distance estimation to the wall by using beamforming.

Finally, we develop a constrained beam retracing algorithm based on the estimated AoA candidates and room structure.

We localize the user at the intersection between the propagation paths with only one-time reflection. Our retracing can effectively identify the plausible user location.

We implement and evaluate our AoA and localization approaches in an anechoic chamber, conference room, bedroom and living room. Our results show that our AoA estimation yields median errors of 1.49^o and 3.33^o for the top two paths in LoS, and 2.75^o and 6.49^o in NLoS. Moreover, our retracing algorithm can localize the user with a median error of $0.31m$ in LoS and $0.47m$ in NLoS.

The contributions can be described as follows:

1. We develop a multi-resolution analysis to estimate the AoA of multipath. It combines STFS over different window sizes and wavelet to reduce coherence between signals.
2. We develop an effective method to estimate room structure and retrace the user based on the estimated AoA and room structure.
3. We implement a system to actively map indoor rooms and localize voice sources using only a smartspeaker without additional hardware. Our prototype system can localize voice in both LoS and NLoS. To our knowledge, this is the first indoor sound source localization system that works for None-Line-of-Sight (NLoS) scenarios.

2 Primer on AoA Estimation

In this section, we introduce AoA estimation problem, existing approaches, and challenges.

2.1 Antenna Array Model

We can estimate the AoA using an antenna array. The antenna array can take different forms, such as uniform circular array (UCA), uniform linear array (ULA), or even non-uniform array. This paper uses a uniform circular array consisting of N microphones as shown in Figure 3. The circle has a radius of r. The azimuth and elevation angles of signal arrival are θ and ϕ, respectively.

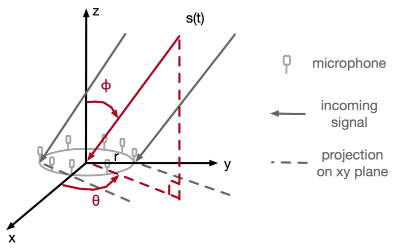


Figure 3: UCA Array model and angle notations.

A general model for the received signal of a single source is

$$x(t) = a(\theta,\phi)s(t) + n(t), \qquad (1)$$

where a is the array steering vector and $n(t)$ is the noise vector. The steering vector for UCA is as follows:

$$a(\theta,\phi) = [1, e^{j2\pi\frac{f}{c}rcos(\theta)sin(\phi)}, \ldots, e^{j2\pi\frac{f}{c}r(N-1)cos(\theta)sin(\phi)}]^T. \qquad (2)$$

where f is center frequency and c is sound propagation speed. For M independent source signals $S(t) = [s_1(t), \ldots, s_M(t)]^T$, we can extend the steering vector to a steering matrix, $A(\theta,\phi) = [a(\theta_1,\phi_1), \ldots, a(\theta_M,\phi_M)]$, where the i_{th} column is the steering vector associated with the i_{th} signal.

2.2 AoA Estimation Algorithms

There are several AoA estimation algorithms, including phase [43], MUSIC [35], ESPIRIT [17], and beamforming. The subspace based MUSIC algorithm is the most accurate. To apply MUSIC, we calculate the auto-correlation matrix R for the received signals x as $x^H x$, where x^H is conjugate transpose of x and R has the size $N \times N$. Following that, we apply eigenvalue decomposition to R, and sort the eigenvectors in a descending order in terms of the magnitude of corresponding eigenvalues. The space spanned by the first M eigenvectors is called *signal space*, and the space spanned by the other eigenvectors is called *noise space*. Let R_N denote the noise space matrix, whose the i_{th} column is the i_{th} eigenvector in the noise space. It can be shown that

$$R_N^H \cdot a(\theta_0,\phi_0) = 0, \qquad (3)$$

when θ_0 and ϕ_0 are the incoming azimuth and elevation angles [35]. Based on this property, we can define a pseudo-spectrum of the mixed signals as

$$p(\theta,\phi) = \frac{1}{a(\theta,\phi)^H R_N R_N^H a(\theta,\phi)}. \qquad (4)$$

Then we can estimate the AoA by locating peaks in the pseudo-spectrum.

2.3 Modeling Multipath Propagation

Now we consider signals under multipath propagation. Most traditional AoA estimation algorithms have the assumption that the signal sources should be independent. In contrast, our system requires estimating the AoA of multipath and have to handle coherent signals. To capture multipath effects, we introduce a channel matrix $H(\alpha,d) = [h(\alpha_1,d_1), \ldots, h(\alpha_M,d_M)]^T$, where α_i, d_i, and $h(\alpha_i,d_i) = \alpha_i\frac{d_0}{d_i}e^{j2\pi\frac{f}{c}d_i}$ are the attenuation, propagation delay, and channel of the $i-th$ path, respectively. The received signal $x(t)$ under multipath is as follows:

$$x(t) = A(\theta,\phi)H(\alpha,d)s(t) + n(t), \qquad (5)$$

For the array model under multipath in Equation 5, we define a transformation matrix $T = A * H$ to capture the array manifold matrix A and propagation paths H. The transformation matrix T is

$$T_{i,j,k} = \alpha_j \frac{d_0}{d_j} e^{j2\pi \frac{d_j}{\lambda_k}} e^{j2\pi \frac{r}{\lambda_k}(i-1)cos(\theta_j)sin(\phi_j)} \tag{6}$$

where $1 \leq i \leq N$ denotes the microphone index, $1 \leq j \leq M$ denotes the j_{th} arrival path, and k denotes the frequency bin index. The transformation matrix T takes three dimensions: spatial dimension i, path delay in time dimension j, and frequency dimension k, which allows us to perform cancellation in the time-frequency domain.

The received signal from all incoming paths to microphone m_i on frequency f_k is

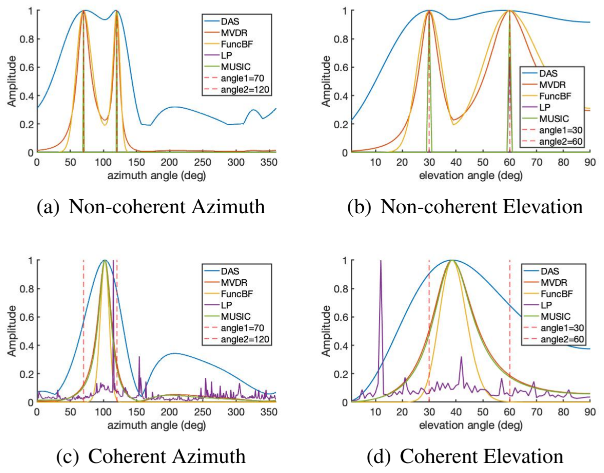
$$x(t)_{i,k} = \hat{T}_{i,k} * s(t) + n(t). \tag{7}$$

where $\hat{T}_{i,k} = \sum_{1 \leq j \leq M} T_{i,j,k}$. In order to estimate the AoA of multipath, we need to deconvolve $\hat{T}_{i,k}$ to each propagation path $T_{i,j,k}$.

2.4 Challenges

Coherent signals: A major source of AoA error comes from the coherence in the incoming signals. In our context, the received signals come from the same voice source and only differ in their propagation paths. Such strong correlation can significantly degrade the AoA estimation accuracy. We quantify the impact of coherent signals on several well-known AoA estimation schemes in the frequency range of human voice. We use a UCA with radius of $9.6cm$, which is approximately the half wavelength of $2kHz$. The two signals are $(70, 120)$ and $(30, 60)$ in the azimuth and elevation angles. Figure 4(a) and (b) are the azimuth and elevation power profiles of five AoA algorithms for two non-coherent signals and Figure 4(c) and (d) are profiles of two coherent multipath signals coming from the same source. MUSIC performs the best in all scenarios. However, when coherence occurs, the estimation errors increase in all algorithms. For example, the two peaks in MUSIC merge into one peak in this case and LP even gives incorrect results.

Impact of frequency: The low frequency of the voice also accounts for part of the error. Existing acoustic tracking schemes (*e.g.*, [23,25,44]) use frequency at $16kHz$ or higher. In comparison, human voice is typically below $6kHz$ [27, 33] and most energy is concentrated in $100Hz$-$3kHz$. The corresponding wavelength ranges between $11cm$ and $3.4m$. The resolution of angle of arrival is determined by the antenna separation distance normalized by the wavelength. Therefore, with centimeter level separation between the microphones and dm-level wavelength, the AoA resolution is very coarse.



(a) Non-coherent Azimuth (b) Non-coherent Elevation

(c) Coherent Azimuth (d) Coherent Elevation

Figure 4: Comparison of power profiles for different AoA algorithms in non-coherent (a,b) and coherent (c,d) scenarios. Coherence makes peaks merged and introduces error.

Summary: The above evaluation shows that MUSIC is competitive for AoA estimation accuracy. However, the accuracy is still insufficient to support coherent low-frequency voice signals. Motivated by these observations, next we will design approaches to explicitly address these major challenges.

3 Multipath Voice Localization

We decompose our approach into the following three steps: (i) estimate the AoA of coherent low-frequency voice signals, (ii) estimate the room structure, (iii) retrace the paths to localize the user. Below we describe each step in turn.

3.1 AoA Estimation of Voice Signals

As shown in Section 2, we should address two major challenges in AoA estimation of human voices: (i) received signals are strongly correlated and (ii) limited resolution due to the low frequency of human voice. Below we describe our sections in turn.

Limitation of existing work: Recently, Voloc [37] proposed an iterative-delay-and-cancellation algorithm to align and cancel the correlated paths to separate multipath signals in the time domain. Their first step, called ICA, is to estimate the AoA of the first reflection by using the initial recording samples before mixing with the second reflection. However, this method introduces two major problems. First, in order to cancel in the time domain, we need to use a small enough time window during which only samples from the direct path are included, usually only tens of samples. A small number of samples limits the AoA estimation accuracy. Moreover, hu-

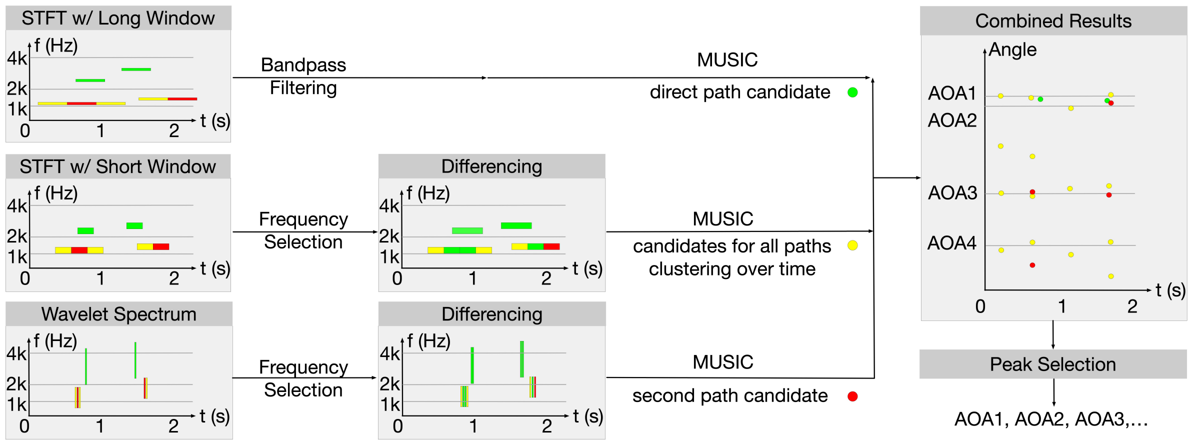


Figure 5: Illustration for multi-resolution analysis algorithm. We perform wavelet and STFT analyses over different time windows followed by the differencing component for small windows. We synthesize the combined results to select the final AoA results.

man voice ramps up slowly. This means the beginning cleaner audio samples for AoA estimation have low SNR, which also limits the accuracy. In addition, the cyclic autocorrelation property of human voice is large, which indicates small alignment error introduces large cancellation error. Therefore, Voloc reports over 10 degrees error for the first path AoA and relies on their second step, which uses joint optimization based on wall geometry to refine the estimation result. This has several limitations: (i) its standalone AoA estimation has limited accuracy, and (ii) the second step requires exploring a large search space, which is very time consuming (*e.g.*, hours to estimate wall parameters and 5 seconds to localize voice).

Overview: Different from [37], we use time-frequency analysis to reduce coherence in voice signals since signals that differ in either time or frequency will be separated out. As the transformation matrix $T_{i,j,k}$ shown in Equation 6, the IAC algorithm in Voloc aligns phases for each microphone i to cancel path delays d_j and get the second reflected path. We first separate coherence in across different frequency bins, and then cancel the paths in each frequency bin by taking the difference between the two consecutive time windows. This is especially useful for voice signals since different pitches may occur at different time. An important decision in time-frequency analysis is to select the sizes of time window and frequency bin to perform the analysis.

On one hand, aggregating the signals over a larger time window and larger frequency bin improves SNR and in turn improves the AoA estimation accuracy according to the Cramer-Rao bound [38]. On the other hand, a larger time window and larger frequency bin also mean more coherent signals. Moreover, the frequency of voice signals varies unpredictably over time, which makes it challenging to determine a fixed time window and frequency bin.

To separate paths with different delay, we desire good time resolution. Small time windows have good time resolution, but poor frequency resolution. To separate paths with different frequencies, we desire good frequency resolution. Small frequency bins have good frequency resolution, but poor time resolution. Therefore, there is no single time window or frequency bin that works well in all cases.

To address this challenge, we use multi-resolution analysis as illustrated in Figure 5. Specifically, we use Short Term Fourier Transform (STFT) with different window sizes and wavelet as they are complementary to each other. Our first method performs STFT using a large time window and feeds the spectrogram to MUSIC. While STFT results with large window have more coherent signals, which results in more outliers, their peaks also include points that are close to the ground truth, likely due to the stronger cumulative energy. Our second method is to perform frequency analysis using smaller windows and take difference between adjacent windows to reduce the coherent signals and improve AoA estimation under coherent multipath. Our third method uses wavelet. It has higher time resolution for relatively high frequency signals. This allows us to capture the transient voice signals that has low or no coherence, thereby reducing outliers in MUSIC AoA estimation. However, since transient signals have low cumulative energy and cause non-negligible AoA estimation errors, we combine Wavelet with STFT with different window sizes. Below we elaborate these three methods.

STFT using a large window size: We perform STFT using a larger time window. A larger window yields higher SNR and hence higher accuracy according to the Cramer-Rao bound [38]. On the other hand, a larger window tends to have more coherent multipath, which may degrade the accuracy. This is shown in Figure 4(c), where we see a merged peak near the ground truth. So this approach can provide information about the AoA of the direct path, but not sufficient on its own.

STFT using a short window: Using a smaller time window gives good time resolution and helps separate paths with different delays. We choose to use a smaller time window and select the evanescent pitches in the time-frequency domain to reduce error from coherence. The next step is to further reduce coherent signals by taking difference between two consecutive time windows for each antenna. This cancels the paths with different delay in the time-frequency domain, and is more effective than cancelling in the time-domain alone. If the difference between two adjacent windows is greater than the delay difference of any two paths, this process can remove the old paths. This cancellation is not perfect since the amplitude may vary over time and each window may contain different sets of paths. Nevertheless it reduces coherence in a short time window.

Wavelet based analysis: Wavelet is a multi-resolution analysis. We can use both short basis functions to isolate signal discontinuities and long basis functions to perform detailed frequency analysis. It has super resolution for relatively high frequency signals. Transient signals in small time window have less energy and may yield large errors. To improve the accuracy, we also take difference of wavelet spectrum in the two consecutive time windows to further reduce the coherence.

Comparison: We compare the AoA derived from applying MUSIC to STFT and wavelet. Figure 6 shows the result for the case where a woman speaks at $2.4m$ away from the microphone array. The dashed red lines are ground truth AoAs of different paths. The STFT results without taking difference, shown in the blue circles, deviate from the right angles due to coherence even after using different window sizes. The wavelet results without taking difference are plot as yellow circles, which also deviates a lot from red dashed lines because of low energy. The stared orange and purple points are the AoA estimates derived from MUSIC when we apply differencing to STFT and wavelet, called STFT Diff and Wavelet Diff methods. Compared with the original results (shown in blue and yellow circles), differencing brings the estimation closer to the ground truth angles (shown as dashed lines). It is interesting to observe that there are false peaks in STFT Diff but the peaks in the Wavelet Diff are all close to the ground truth though STFT Diff may have peaks closer to the ground truth than the wavelet. This suggests that it is beneficial to combine STFT Diff and wavelet Diff results.

Final algorithm: Figure 5 shows our final algorithm. For each algorithm, we derive the results using different time windows. Then we compute weighted cluster of these points where the weight is set according to the magnitude of the MUSIC peak. We select the top K clusters from each algorithm. Our evaluation uses $K = 6$. To combine the results across different algorithms, we use nearest neighbors. Since STFT with a large window provides more stable results without significant outliers, we use them to form the base. For each point in the base, we search for the nearest neighbor in the results of the other two methods as they contain both more accurate real peaks and outlier peaks. Finally, we pick the top P peaks from the selected nearest neighbors as the final AoA estimates. Our evaluation uses $P = 5$.

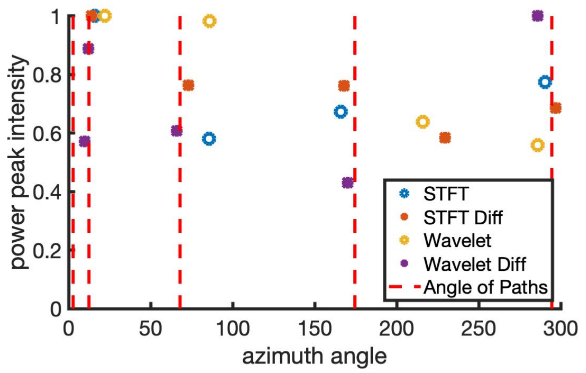


Figure 6: Comparison of AoA derived from STFT, Wavelet with and without differencing.

Algorithm 1 Multi-resolution analysis algorithm.

```
1: function [AoAs, w] = MultiResolutionAoA(signal)
2: Bandpass filter in voice frequency range
3: spectLong = STFT(signal,LongWindow);
4: spectShortDiff = diff(STFT(signal,ShortWindow));
5: spectWaveletDiff = diff(Wavelet(signal));
6: Select frequency and time ranges based on spectrograms
7: for method in STFTLong,STFTDiff,WaveletDiff do
8:     for time in SelectedTimeSlots do
9:         for frequency in SelectedFrequencies do
10:            forward backward smoothing;
11:            compute MUSIC profile;
12:        end for
13:        accumProfile = SUM(profile)
14:        [results,weights] = findPeaks(accumProfile);
15:        estimate $candidateAoAs_m$ and $weights_m$;
16:    end for
17: end for
18: AoAs = select top P peaks from $candidateAoAs_m$ for m=1..3
```

3.2 Room Structure Estimation

In order to localize the user, we need not only the AoAs of the propagation paths of the voice signals, but also the room structure information so as to retrace back the paths. In this section, we estimate the room contour using wideband 3D MUSIC algorithms. We improve the accuracy by leveraging constraints of the azimuth AoA and applying beamforming.

3.2.1 3D MUSIC

The smart speaker estimates the room structure once unless it is moved to a new position. The smart speaker estimates room structure by sending FMCW chirps. Let f_c, B and T denote the center frequency, bandwith, duration of the chirp. Upon receiving the reflected signals, it applies the 3D MUSIC algorithm.

We generalize 2D Range-Azimuth MUSIC algorithm [5,6,22] to 3D joint estimation of distance, azimuth AoA and elevation AoA. 3D MUSIC has better resolution than 2D MUSIC since the peaks that differ in any of the three dimensions are separated out. Our basic idea is to transform the received signals into a 3D sinusoid whose frequencies are proportional to the distance and a function of the two angles. We extend the steering vector to have three input parameters: distance R, azimuth angle θ, and elevation angle ϕ.

$$\hat{a}(R,\theta,\phi) = e^{j2\pi \frac{r}{c} f_c \sin\phi \cos(\theta - \frac{2\pi i}{N}) + j4\pi \frac{RB}{cT} N_s M_s T_s}, \qquad (8)$$

where i is the array index, N is the number of microphones, r is the radius of the microphone array, c is sound speed, N_s is the subsampling rate, M_s is the temporal smoothing window and T_s is the time interval.

3.2.2 Our Enhancements

However, there are several challenges in applying the 3D MUSIC algorithm to indoor environments. First, the number of microphones and their sizes are both limited, which limits the resolution of 3D MUSIC. Second, there is significant reverberation in indoor scenarios. Third, large bandwidth is required to get accurate distance estimation, but MUSIC requires narrowband signals for AoA estimation. Therefore, we develop three techniques to improve the 3D MUSIC algorithm: (i) leveraging frequency diversity, (ii) incorporating the fact that rooms are typically rectangular shaped, and (iii) using beamforming to improve distance estimation.

Multiband 3D MUSIC: We use FMCW signals from $1kHz$ to $3kHz$ for AoA estimation. To satisfy the narrowband requirement in the MUSIC algorithm [35], we divide the 2 KHz bandwidth into 20 subbands each with $100Hz$. Since the frequency of FMCW signal increases linearly over time, we can divide the FMCW signal into multiple subbands in the time domain, run 3D MUSIC in each subband, and then sum up the MUSIC profiles from all subbands.

In order to use the $100Hz$ subband for 3D MUSIC, we should properly align the transmission signal with the received signal so that they span the same subband. The alignment is determined by the distance. Therefore, we search over the azimuth and distance for a peak in the 3D MUSIC profile obtained by mixing the received signal with the transmitted signal that is sent δT ago, where δT is the propagation delay and determined based on the distance.

We use the azimuth AoA and distance output from the 3D MUSIC. Figure 7 shows an example of azimuth-distance profile. Note that we adjust the elevation angle to the horizontal AoA since the elevation AoA estimation from the UCA (which has all antennas on the same horizontal plane) is not very accurate. However, despite a larger error in elevation AoA, the 3D MUSIC is more effective in separating the paths than the 2D MUSIC.

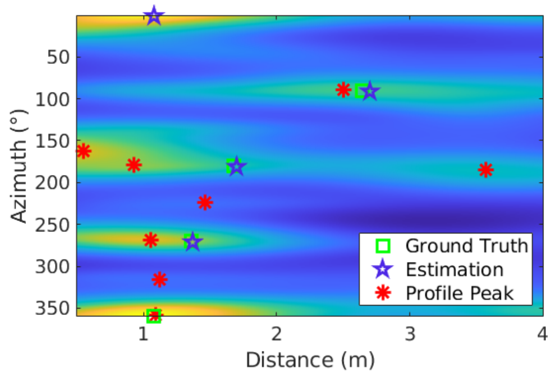


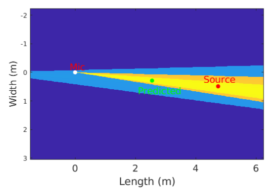
Figure 7: An example of azimuth-distance profile from real trace. Azimuths are accurate and distances requires further the fine granularity search.

Refine AoA for a regular room: Due to multipath, the MUSIC profile can be noisy, which makes it hard to determine the right peaks to use for distance and AoA estimation of walls. Since most rooms take rectangular shapes, we leverage this information to improve peak selection. Specifically, we select the peaks such that the difference in the azimuth AoA of two consecutive peaks are as close to 90^o as possible. That is, we search for the 4 peaks $\{\theta_0, \theta_1, \theta_2, \theta_3\}$ from the 3D MUSIC profile that minimizes the fitting error with a rectangular room (*i.e.*, $min \sum_i |PhaseDiff(\theta_i - \theta_{i+1}) - \pi/2|$, where $PhaseDiff(.)$ is the difference between the two angles by taking into account of the phase wraps every 2π. After finding these peaks, we further adjust the solutions so that the difference between the adjacent AoA is exactly $\pi/2$. This can be done by find θ_1' that minimizes $\sum_i |PhaseDiff(\theta_1' + \pi/2(i-1) - \theta_i)|$ and the final AoA is set to $(\theta_1', \theta_1' + \pi/2, \theta_1' + \pi, \theta_1' + 3/2\pi)$.

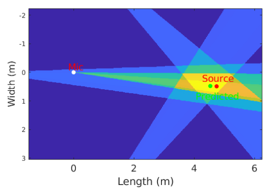
Improve distance estimation by beamforming: Accurate distance estimation requires a large bandwidth and high SNR. Therefore, to improve distance estimation, we send $1kHz$-$10kHz$ FMCW chirps. Among them, we only use $1KHz - 3KHz$ for AoA estimation to reduce computational cost since MUSIC requires expensive eigenvalue decomposition, but use the entire FMCW for distance estimation. We increase the SNR using beamforming. We use delay-and-sum (DAS) beamforming algorithm towards the estimated azimuth AoAs. Then we search a peak in the beamformed FMCW profile. We find that the peak magnitude increases significantly and get more accurate distance estimation after beamforming.

3.3 Constrained Beam Retracing

We can localize the user by retracing the paths using the estimated AoA of the voice signals and room structure. As shown in left figure of Figure 8, we can first find the reflection points on the walls by the propagation path derived from the estimated AoA. Then we trace back the incoming path of voice signals before the wall reflection based on the reflection property. If we have at least two paths, the user is localized at the intersection between the incoming paths. However, the above method is not robust against AoA estimation error. When simulating the retracing algorithm, we find that even when the AoA estimation errors of 2 paths are only 0.5 degrees, it can cause a localization error of more than 60 cm at a distance of 4 meters. A small AoA error can result in a large localization error at a large distance. Moreover, an AoA error in the outgoing path can result in an error in the incoming path, thereby further amplifying this effect. To enhance robustness



(a) Two near parallel paths



(b) More paths

Figure 8: Retracing using ray or cone.

against AoA estimation, we employ two strategies. First, instead of treating each propagation path as a ray defined by the estimated AoA, we treat it as a cone where the cone center is determined by the estimated AoA and the cone width is determined by the MUSIC peak width. This allows us to capture the uncertainty in the AoA estimation.

Second, while theoretically two paths are sufficient to perform triangulation, it is challenging to select the right paths for triangulation. Therefore, instead of prematurely selecting incorrect paths, we let the AoA estimation procedure return more paths so that we can incorporate the room structure to make informed decision on which paths to use for localization. Specifically, for each of the K paths returned by our AoA estimation, we trace back using the cone structure as shown in Figure 8. We observe that the azimuth AoA is reliable for the strongest path, which is the direct path in LoS or the path from the user to the ceiling and then to the microphone in NLoS. Therefore, within the cone corresponding to the strongest path we search for a point O such that the circle centered at the point with radius of 0.5m overlaps with the maximum number of cones corresponding to the other $K-1$ paths. We localize the user at the point O. Our evaluation sets $K = 4$.

4 Implementation

Setups: We implement our system on a Bela platform [4]. It is connected with a JBL Clip 3 or an echo dot speaker and a circular microphone array with 8 microphones. Figure 9 shows an example setup in a conference room. Each microphone uses a sampling rate of 22.05kHz. Many commercial smart speakers have similar numbers of speakers and microphones. We test our system using two microphone arrays: a larger array has radius of 9.6cm and a smaller one has radius of 5.0cm. We use the smaller array to compare with VoLoc [37] since its size is similar to their setup. The Bela board uses a 1 GHz ARM Cortex-A8 single-core processor. The Bela is connected to a laptop with Intel I5 processor and 8GB memory. We use javaosc protocol to listen and continuously transmit the audio signals in WAV format encapsulated in OSC packets to the laptop through USB in real time and run the processing program in MATLAB on the laptop to derive the AoAs and localize the user. In MAVL , AoA estimation takes 2.35 seconds, room estimation takes 87 seconds, and retracing takes 0.16 seconds. In comparison, VoLoc spends hours in estimating wall parameters and 5 seconds in AoA estimation.

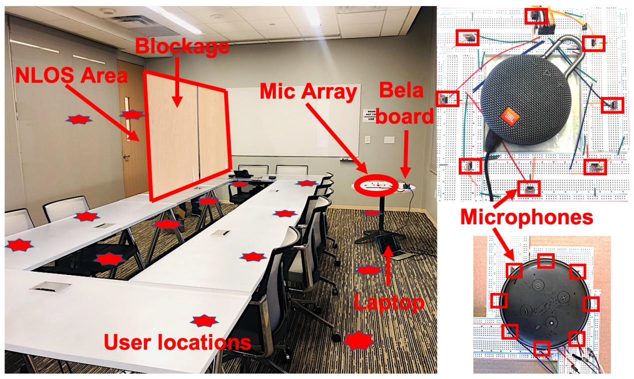


Figure 9: System setups in conference room and mic arrays.

Evaluation environments: We evaluate our system in different environments, including an anechoic chamber, conference room, bedroom and living room. These rooms take different sizes: 2.5m×3.5m, 3.5m×4.0m, and 5.1m×7.5m. We use a wooden board as a blockage in NLoS cases as shown in Figure 9. We let a person speak at $1-6$ meters away from the microphone array in the room. We also vary the distance, users, type of voices (*e.g.*, man, women, child and applause), smartspeaker positions, clutter and noise levels to assess their impacts.

Ground truth: We measure the relative locations of the smartspeaker, user and walls using a measuring tap. We derive the ground truth AoAs of the direct path and 5 reflected paths (*i.e.*, the paths from 4 side walls and ceiling) in LoS scenarios. In NLoS scenarios, we derive the AoAs of the 4 reflected paths and 1 diffraction path.

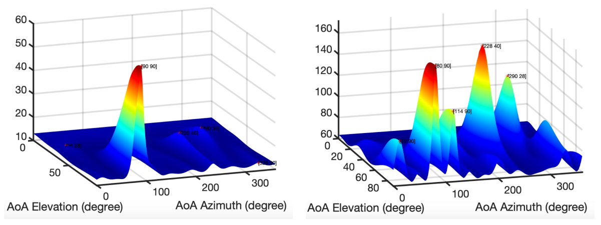
Metrics: We quantify the errors using both AoA estimation error and localization error. The localization error is computed based on the Euclidean distance between the ground truth and estimated positions.

5 Evaluation

In this section, we evaluate our AoA estimation, room contour estimation, and voice localization accuracy.

5.1 Performance of AoA Estimation

Two paths in anechoic chamber. We start from testing our AoA estimation algorithm in the anechoic chamber, where there is no reflection in the room. We put our microphone array on the ground and place an acrylic board to act as a wall to introduce a reflection path. The ground truth of two angles are 81.95^o and 112.68^o. Figure 10 shows the MUSIC power profile. It has a single merged peak around 90^o, which results in 8^o and 22.68^o errors for the two paths. In comparison, our algorithm accurately estimates these two paths within the error of 1.5^o. We can clearly see there are two separate peaks in our MUSIC profile Figure 10. We also change the acrylic board reflector to other places, and find that MUSIC can separate the two paths only when the difference between two ground truth angles is greater than 90^o. This resolution is not sufficient for voice localization since it is quite likely to have reflected paths within 90^o. In comparison, our approach can separate the two paths as long as they are 30^o apart.



(a) MUSIC with merged peak. (b) MAVL with separated peaks.

Figure 10: Comparison of power profiles in anechoic chamber.

AoA accuracy for LoS and NLoS: Next we conduct experiments in three rooms. Figure 11 shows the CDF of LoS AoA estimation error of six methods for the top 3 angles across all experiments. We use a large UCA of radius $9.6cm$, comparable to Amazon Echo Studio, Google Home Max and Apple HomePod. The median error of our approach for the top two paths are 1.49^o and 3.33^o, respectively. This accuracy is sufficient for retracing. In comparison, the corresponding numbers for MUSIC are 2.55^o and 14.54^o , which are significantly worse.

Figure 12 shows the CDF of NLoS AoA estimation error for the top 3 angles across all experiments. The median errors of the top two paths are 2.75^o and 6.49^o. We also plot the CDF for the third angle estimation. In theory, one can retrace the user's location using two paths. However, a median error around 10^o for the third path is too large to be used directly for triangulation. Nevertheless our cone-based retracing algorithm can still leverage the AoA of the paths beyond the top two paths to improve the localization accuracy despite their relatively high errors.

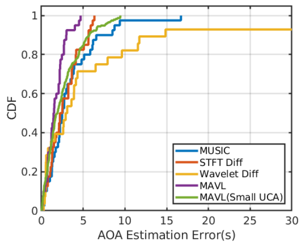
We also evaluate MAVL using a smaller UCA with a radius of $5cm$, comparable to the size of Echo Dot, Amazon Echo and Google home. Figure 13 compares the AoA accuracy of the first path with MAVL using small UCA, VoLoc using ICA algorithm only and VoLoc using joint estimation. Using our approach, the median AoA error of the first path is 1.98^o and the second path is 4.08^o, both of which are larger than the errors from the larger UCA, which are 1.49^o and 3.33^o, respectively. In comparison, Voloc yields median errors of 18.04^o and 5.28^o before and after joint optimization, respectively, much larger than the errors of MAVL.

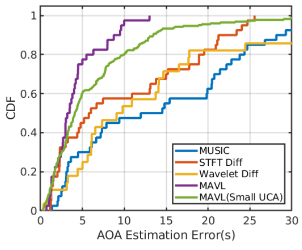
AoA performance to distance: Figure 14 plots the AoA error versus the distance between the user and smart speaker in a $7.5m \times 5.1m$ conference room. Overall, the accuracy degrades slightly as the user moves away from the microphone array. The SNR of voice is not a serious problem because its frequency is low and it attenuates slowly in the air.

Interestingly, the AoA error of our approach at $4m$ is better than many other distances. This could be due to the specific room structure and user's distance to the nearby wall. Measurements at the distance around $4m$ were collected when the user is near the middle of the room, which makes the propagation delay from the reflected path well separated from the direct path and alleviates the coherence effects. The measurements at a larger distance (*e.g.*, $5m$) were collected when the user was close to the wall and the difference between the direct path and reflect path is smaller, which makes it more challenging to separate in the MUSIC profile.

Performance to different voices: We classify our measurements into four groups: *i.e.*man, woman, child, and applause. Figure 15 shows the sensitivity to different users' voices. The bars are centered at the mean error and their two ends denote the minimum and maximum values across all traces. Our system is fairly robust across the users and the voice they produced. We also evaluate the applause sound, and find the AoA errors of the two paths are about 1.4^o and 3.0^o. The applause sound has smaller AoA error because it is shorter than the human voice, which reduces coherence and improves AoA estimation accuracy.

Impact of smartspeaker positions: The relative positions between the microphone array and walls have direct influence on multiple propagation paths. VoLoc requires the microphone array to be close to a wall to ensure that the first two


(a) AoA 1 CDF.


(b) AoA 2 CDF.

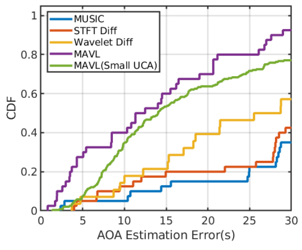

(c) AoA 3 CDF.

Figure 11: Comparison of LoS CDF of AoA estimation.

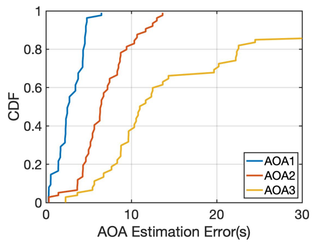


Figure 12: CDF of AoAs error for NLoS.

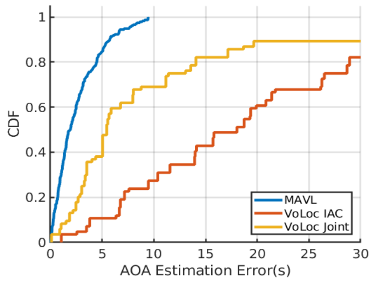


Figure 13: Comparison of AoA estimation for the small UCA.

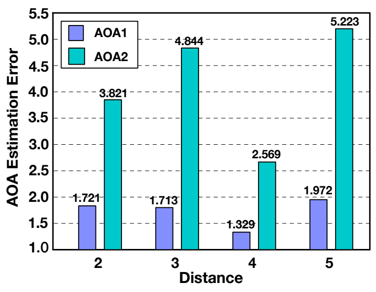


Figure 14: AoA accuracy vs distance.

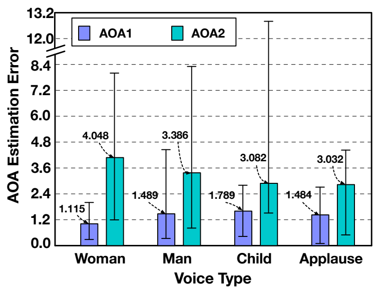


Figure 15: AoA accuracy to voices.

paths come much earlier than other paths. We evaluate the robustness of MAVL against smartspeaker positions. We evaluate UCA setup at three positions:

(1) *center*: 2.35m and 2.92m to the two closest walls;
(2) *close to one wall*: 0.3m and 2.4m to the two closest walls;
(3) *corner*: 0.26m and 0.39m to the two closest walls.

The median AoA errors of MAVL are 1.80^o, 1.97^o, 2.08^o for the direct path, when the smartspeaker is at *center*, *close to one wall* and *corner*, respectively; the corresponding AoA errors are 3.07^o, 4.51^o, 4.37^o for the second path AoA, respectively. MAVL performs best at the *center* and worst near the *corner*. The latter is because the second and third paths have comparable SNR and closer AoAs to the direct path, which increases coherence. But overall it is fairly robust to different placement. In comparison, the median AoA error of VoLoc before its joint optimization is 18.04^o for direct path, when the UCA is placed *close to one wall*. It does not work at the *center* or *corner*. VoLoc only works when the UCA is *close to one wall* and users are not close to any wall.

5.2 Performance of Room Estimation

Next we evaluate our room structure estimation algorithm using different room sizes and microphone placements.

Overall Room estimation Performance: We use room sizes of 2.5m×3.5m, 3.5m×4.0m, and 5.1m×7.5m. The median distance error for all walls is 2.8cm and azimuth error is 1.8^o. We can reduce the azimuth error to 1.4^o by leveraging the knowledge of room shape (*i.e.*, the azimuth angles of walls differ by 90 degrees for rectangular rooms). VoLoc jointly estimates the wall parameters. We follow the VoLoc's setup that the UCA is close to one wall. We speak 5 commands to find the best parameters. The distance error is 2.5cm and azimuth error is 12^o. Its performance is sensitive to the selection of the beginning samples and window size for cancellation.

Impact of smart speaker positions: We also vary the positions of the smart speaker in the rooms to evaluate its impact. We plot the median AoA and distance errors in Figure 16 as we vary the distance between the smart speaker and the wall from 5cm to 20cm. We find an interesting trade-off between the distance error and azimuth error. For the shortest distance range ($< 0.5m$), it has a small distance error of 1.5cm and a larger azimuth error 5.1^o. For the longest distance range ($> 2m$), it has an azimuth error of 1.1^o and a distance error of 5.4cm. The worse distance error for the far away wall has little impact on the final localization error, because the reflected signals from this wall always have a much lower SNR and these results are rarely used for retracing.

5.3 Overall localization results

Localization accuracy: Figure 17 shows the CDF of MAVL localization errors in LoS (blue line) and NLoS (or-

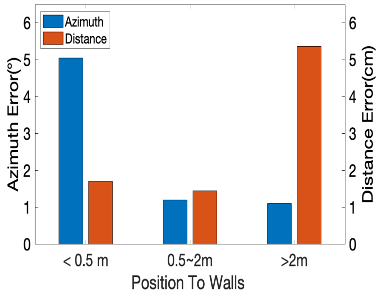


Figure 16: Wall estimation performance over distance.

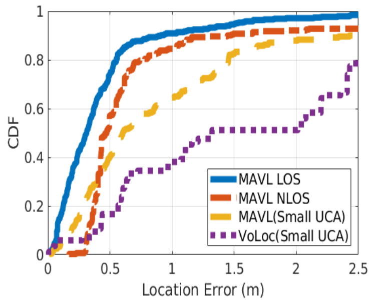


Figure 17: CDF of Localization error for LoS and NLoS, small UCA and Voloc.

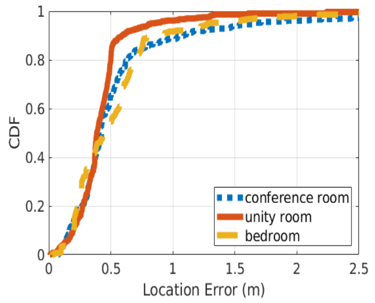


Figure 18: CDF of MAVL Localization error in different rooms.

ange line) scenarios. The median error is 0.31m for LoS and 0.47m for NLoS across all ranges and environments in our evaluation. The accuracy decreases slightly in the NLoS scenario compared to LoS because the diffraction path has lower SNR. The overall localization error for smaller UCA is 0.56m in MAVL . VoLoc [37] reports an overall median error of 0.44m in LoS and a median error of 1.7 m at a large distance (>4m). In our setup, we put the smart speaker *close to one wall*, which is the only setup that VoLoc can work, and find the median error of 1.32 m. This error is larger than the one reported in [37] likely due to different distances and environments.

Performance in different rooms: Figure 18 presents the CDF of localization errors in different rooms. We select three representative environments: a 7.5m x 5.1m conference room with a large desk and many chairs, a 4m x 3.5m bedroom with strong reflectors, such as monitors and wooden furniture, a 3.5m x 2.5m utility room with soft reflectors. We can see that localization error increases with the increasing room size and the number of strong reflectors. A larger room size reduces SNR. For many locations in a large room, the directions of reflected paths are close to each other, which makes it more difficult to separate difference paths. Strong reflection from walls and large furniture may produce merged peaks in the MUSIC profiles. Nevertheless, MAVL still achieves 0.45m median error for the complex bedroom .

Impact of UCA size: As discussed earlier, a smaller UCA size degrades the accuracy of AoAs. The overall localization error for smaller UCA is 0.56m. The yellow line in Figure 17 shows how small UCA works in our system. Although it is worse than that of the larger UCA size, the error can still support many indoor localization applications (*e.g.*, providing useful context information for speech recognition and beamforming to strengthen SNR).

Impact of different positions of UCA: Position of the microphone array have impact on both room contour estimation and source AoA estimation. We place the UCA at three predefined locations, *center*, *close to one wall* and *corner* and evaluate our system. The median localization errors are 0.41m, 0.59m, 0.76m at *center*, *close to one wall*, and *corner*, respectively. Our system works the best when the UCA is placed at the *center*. The accuracy degrades significantly if the UCA is placed at the corner due to increased coherence. VoLoc reports 0.44m overall error and 1.7m error beyond 4m when UCA is placed *close to one wall*. But in our settings with a larger room size and larger distance, VoLoc yields a median error of 1.32 m. VoLoc relies on direct path and reflection path from the close wall in the back. When one retrace using these two paths, a small AoA error may lead to a large localization error. Note that what matters is not the absolute distance to the wall but the ratio between the distance to the wall and the room size. For instance, 0.5m to a wall is considered close for a 5.1$m \times$ 7.5m room and large for a 2$m \times$ 3m room. Our system works best in the center position, but also works well for the other setups. Therefore it can support more flexible placement.

Performance to clutter levels: Nearby objects introduce multipath, which makes the AoA estimation more challenging. Figure 20 shows how the clutter level affects the final localization errors across different types of voice. Increasing the clutter level increases the localization errors as we would expect.

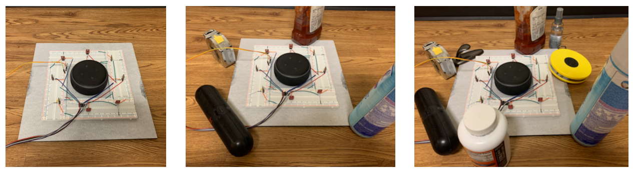

(a) Sparse (b) Moderate (c) Dense

Figure 19: Clutter Setups.

Performance to noise level: MAVL is robust to different background noise. Figure 21 shows the influence of various background noise and noise levels. White noise just degrades the accuracy slightly even when SNR is as low as -10dB,

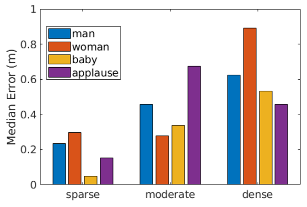


Figure 20: Localization accuracy across clutter levels.

and background music has larger impact than white noise as there are human voices in songs. Our approach is fairly robust against background music unless the SNR is too low (*e.g.*,< -10dB SNR), in which case the error increases to 1.4*m*.

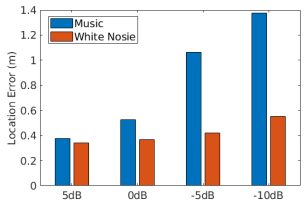


Figure 21: Localization accuracy vs noise levels.

6 Related Work

Acoustic Sensing: A number of systems have been proposed to track a mobile device using acoustic signals [19, 23, 32, 50, 52]. Several recent systems [25, 28, 29, 51] enable device-free tracking using acoustic signals. Many systems generate inaudible acoustic sound for motion tracking. Some use Doppler shift (*e.g.*, AAMouse [50]), time of flight (*e.g.*, BeepBeep [32], or combination (*e.g.*, CAT [23]). Covertband [30] actively sends out OFDM based inaudible signals and builds on top of MUSIC to improve sensing energy. BreathJunior [42] encodes FMCW into white noise to detect motion and breathing of infants. These systems require controlling transmitted acoustic signals and are not suitable for tracking human voice. The most relevant work to ours is VoLoc [37]. Our work advances VoLoc in several important aspects. First, we improve the AoA accuracy from 10 degrees to 1.5 degrees by leveraging multi-resolution analysis in the time-frequency domain. Second, we develop a novel method to automatically estimate the room contour. This significantly eases the deployment effort. Third, we can localize users in both LoS and NLoS whereas they only support LoS.

RF Based Localization: The accuracy of RF based localization approaches are mostly limited by its large wavelength and fast propagation speed for commodity WiFi infrastructure. Chronos [40] can achieve decimeter level localization accuracy by inverting the NDFT. Spotfi [16] incorporates novel filtering and estimation techniques to identify AoA of direct path. Arraytrack [48] designs a novel multipath suppression algorithm to remove reflection between clients and APs. However, they use more than three APs with 16 antennas and require controlling the transmitted signals. Moreover, their approach is focused on eliminating multipath rather than separately estimating each multipath.

Sound Source Localization: There has been a few sound source localization work [26, 34, 46]. [14] builds a real-time system to detect the AoAs of different sound sources. [2] requires a Kinect depth sensor to build a 3D mesh model of an empty room. It estimates multipath AoAs using a cubic microphone array and perform 3D reverse ray-tracing to localize the voice. Its localization error is around 1.12*m*. [1] considers the diffraction path and applies Uniform Theory of Diffraction for voice localization. Its error is 0.82*m*. These works either require multiple specialized sensors to get indoor environment or only estimate AoAs instead of localization. They do not address the coherence arising from multipath, so their AoAs are not reliable. MAVL can localize a user using a single smart speaker without extra hardware and explicitly addresses the coherence of multipath.

Audio-Visual Indoor Representation Learning: Recent work combines sound and vision in multimodal learning frameworks to better understand the environment so that they can track audio-visual targets [3, 11, 13], localize pixels relevant to sound in videos [36, 39], and navigate indoor environments [10]. VisualEchoes [12] emits 3*ms* chirps to combine multipaths and images at different location and learn spatial representation without manual supervision. Soundspaces [7] applies multi-modal deep reinforcement learning on a stream of egocentric audio-visual observations. Our work uses a stand-alone smart speaker, and does not require vision data or pre-training.

7 Conclusion

In this paper, we develop a system, MAVL, to localize users based on their voice using a smartspeaker like device. Our design consists of a novel multi-resolution based AoA estimation algorithm, an easy-to-use acoustic-based room structure estimation approach and a robust retracing to localize the user based on the estimated AoA and room structure. We evaluate MAVL using different sound sources, room sizes, smart speaker setups, noise and clutter levels to demonstrate its effectiveness.

8 ACKNOWLEDGMENTS

This work is supported in part by NSF Grant CNS-1718585 and CNS-2032125. We are grateful to Prof. Shyam Gollakota and anonymous reviewers for their insightful comments and suggestions.

References

[1] Inkyu An, Doheon Lee, Jung-woo Choi, Dinesh Manocha, and Sung-eui Yoon. Diffraction-aware sound localization for a non-line-of-sight source. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 4061–4067. IEEE, 2019.

[2] Inkyu An, Myungbae Son, Dinesh Manocha, and Sung-eui Yoon. Reflection-aware sound source localization. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 66–73. IEEE, 2018.

[3] Yutong Ban, Xiaofei Li, Xavier Alameda-Pineda, Laurent Girin, and Radu Horaud. Accounting for room acoustics in audio-visual multi-speaker tracking. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 6553–6557. IEEE, 2018.

[4] Bela audio expander, 2017. https://github.com/BelaPlatform/Bela/wiki/Using-the-Audio-Expander-Capelet.

[5] Francesco Belfiori, Wim van Rossum, and Peter Hoogeboom. 2d-music technique applied to a coherent fmcw mimo radar. 2012.

[6] Francesco Belfiori, Wim van Rossum, and Peter Hoogeboom. Application of 2d music algorithm to range-azimuth fmcw radar data. In *Radar Conference (EuRAD), 2012 9th European*, pages 242–245. IEEE, 2012.

[7] Changan Chen, Unnat Jain, Carl Schissler, Sebastia Vicenc, Amengual Gari, Ziad Al-Halah, Vamsi Krishna Ithapu, Philip Robinson, and Kristen Grauman. Soundspaces: Audio-visual navigation in 3d environments supplementary materials.

[8] Ross A Clark, Adam L Bryant, Yonghao Pua, Paul McCrory, Kim Bennell, and Michael Hunt. Validity and reliability of the nintendo wii balance board for assessment of standing balance. *Gait & posture*, 31(3):307–310, 2010.

[9] Saumitro Dasgupta, Kuan Fang, Kevin Chen, and Silvio Savarese. Delay: Robust spatial layout estimation for cluttered indoor scenes. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 616–624, 2016.

[10] Chuang Gan, Yiwei Zhang, Jiajun Wu, Boqing Gong, and Joshua B Tenenbaum. Look, listen, and act: Towards audio-visual embodied navigation. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 9701–9707. IEEE, 2020.

[11] Chuang Gan, Hang Zhao, Peihao Chen, David Cox, and Antonio Torralba. Self-supervised moving vehicle tracking with stereo sound. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 7053–7062, 2019.

[12] Ruohan Gao, Changan Chen, Ziad Al-Halah, Carl Schissler, and Kristen Grauman. Visualechoes: Spatial image representation learning through echolocation. *arXiv preprint arXiv:2005.01616*, 2020.

[13] Israel D Gebru, Sileye Ba, Georgios Evangelidis, and Radu Horaud. Tracking the active speaker based on a joint audio-visual observation model. In *Proceedings of the IEEE International Conference on Computer Vision Workshops*, pages 15–21, 2015.

[14] François Grondin and François Michaud. Lightweight and optimized sound source localization and tracking methods for open and closed microphone array configurations. *Robotics and Autonomous Systems*, 113:63–80, 2019.

[15] Brett Jones, Rajinder Sodhi, Michael Murdock, Ravish Mehra, Hrvoje Benko, Andrew Wilson, Eyal Ofek, Blair MacIntyre, Nikunj Raghuvanshi, and Lior Shapira. Roomalive: magical experiences enabled by scalable, adaptive projector-camera units. In *Proceedings of the 27th annual ACM symposium on User interface software and technology*, pages 637–644, 2014.

[16] Manikanta Kotaru, Kiran Joshi, Dinesh Bharadia, and Sachin Katti. Spotfi: Decimeter level localization using WiFi. In *ACM SIGCOMM Computer Communication Review*, volume 45(4), pages 269–282. ACM, 2015.

[17] Tukaram Baburao Lavate, VK Kokate, and AM Sapkal. Performance analysis of music and esprit doa estimation algorithms for adaptive array smart antenna in mobile communication. In *Computer and Network Technology (ICCNT), 2010 Second International Conference on*, pages 308–311. IEEE, 2010.

[18] Chen-Yu Lee, Vijay Badrinarayanan, Tomasz Malisiewicz, and Andrew Rabinovich. Roomnet: End-to-end room layout estimation. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 4865–4874, 2017.

[19] Qiongzheng Lin, Zhenlin An, and Lei Yang. Rebooting ultrasonic positioning systems for ultrasound-incapable smart devices. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–16, 2019.

[20] David B Lindell, Gordon Wetzstein, and Vladlen Koltun. Acoustic non-line-of-sight imaging. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 6780–6789, 2019.

[21] Arun Mallya and Svetlana Lazebnik. Learning informative edge maps for indoor scene layout prediction. In *Proceedings of the IEEE international conference on computer vision*, pages 936–944, 2015.

[22] Gleb O Manokhin, Zhargal T Erdyneev, Andrey A Geltser, and Evgeny A Monastyrev. Music-based algorithm for range-azimuth fmcw radar data processing without estimating number of targets. In *Microwave Symposium (MMS), 2015 IEEE 15th Mediterranean*, pages 1–4. IEEE, 2015.

[23] Wenguang Mao, Jian He, and Lili Qiu. CAT: high-precision acoustic motion tracking. In *Proc. of ACM MobiCom*, 2016.

[24] Wenguang Mao, Mei Wang, and Lili Qiu. Aim: Acoustic imaging on a mobile. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, pages 468–481. ACM, 2018.

[25] Wenguang Mao, Mei Wang, Wei Sun, Lili Qiu, Swadhin Pradhan, and Yi-Chao Chen. Rnn-based room scale hand motion tracking. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–16, 2019.

[26] Kazuhiro Nakadai, Tino Lourens, Hiroshi G Okuno, and Hiroaki Kitano. Active audition for humanoid. In *AAAI/IAAI*, pages 832–839, 2000.

[27] Rajalakshmi Nandakumar, Krishna Kant Chintalapudi, and Venkata N. Padmanabhan. Dhwani : Secure peer-to-peer acoustic nfc. In *Proc. of ACM SIGCOMM*, 2013.

[28] Rajalakshmi Nandakumar, Shyam Gollakota, and Nathaniel Watson. Contactless sleep apnea detection on smartphones. In *Proc. of ACM MobiSys*, 2015.

[29] Rajalakshmi Nandakumar, Vikram Iyer, Desney Tan, and Shyamnath Gollakota. FingerIO: Using active sonar for fine-grained finger tracking. In *Proc. of ACM CHI*, pages 1515–1525, 2016.

[30] Rajalakshmi Nandakumar, Alex Takakuwa, Tadayoshi Kohno, and Shyamnath Gollakota. Covertband: Activity information leakage using music. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 1(3):1–24, 2017.

[31] Sergio Orts-Escolano, Christoph Rhemann, Sean Fanello, Wayne Chang, Adarsh Kowdle, Yury Degtyarev, David Kim, Philip L Davidson, Sameh Khamis, Mingsong Dou, et al. Holoportation: Virtual 3d teleportation in real-time. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, pages 741–754, 2016.

[32] Chunyi Peng, Guobin Shen, Yongguang Zhang, Yanlin Li, and Kun Tan. BeepBeep: a high accuracy acoustic ranging system using COTS mobile devices. In *Proc. of ACM SenSys*, 2007.

[33] Swadhin Pradhan, Wei Sun, Ghufran Baig, and Lili Qiu. Combating replay attacks against voice assistants. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 3(3):1–26, 2019.

[34] Caleb Rascon and Ivan Meza. Localization of sound sources in robotics: A review. *Robotics and Autonomous Systems*, 96:184–210, 2017.

[35] Ralph Otto Schmidt. A signal subspace approach to multiple emitter location spectral estimation. *Ph. D. Thesis, Stanford University*, 1981.

[36] Arda Senocak, Tae-Hyun Oh, Junsik Kim, Ming-Hsuan Yang, and In So Kweon. Learning to localize sound source in visual scenes. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4358–4366, 2018.

[37] Sheng Shen, Daguan Chen, Yu-Lin Wei, Zhijian Yang, and Romit Roy Choudhury. Voice localization using nearby wall reflections. In *Proc. of ACM MobiCom*, 2020.

[38] Petre Stoica and Arye Nehorai. Music, maximum likelihood, and cramer-rao bound. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 37(5):720–741, 1989.

[39] Yapeng Tian, Jing Shi, Bochen Li, Zhiyao Duan, and Chenliang Xu. Audio-visual event localization in unconstrained videos. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 247–263, 2018.

[40] Deepak Vasisht, Swarun Kumar, and Dina Katabi. Decimeter-level localization with a single wifi access point. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, pages 165–178, 2016.

[41] Anran Wang and Shyamnath Gollakota. Millisonic: Pushing the limits of acoustic motion tracking. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–11, 2019.

[42] Anran Wang, Jacob E Sunshine, and Shyamnath Gollakota. Contactless infant monitoring using white noise. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–16, 2019.

[43] Jue Wang, Deepak Vasisht, and Dina Katabi. Rf-idraw: virtual touch screen in the air using rf signals. *ACM SIGCOMM Computer Communication Review*, 44(4):235–246, 2014.

[44] Wei Wang, Alex X Liu, and Ke Sun. Device-free gesture tracking using acoustic signals. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking*, pages 82–94. ACM, 2016.

[45] Teng Wei and Xinyu Zhang. mTrack: high precision passive tracking using millimeter wave radios. In *Proc. of ACM MobiCom*, 2015.

[46] Xinyu Wu, Haitao Gong, Pei Chen, Zhi Zhong, and Yangsheng Xu. Surveillance robot utilizing video and audio information. *Journal of Intelligent and Robotic Systems*, 55(4-5):403–421, 2009.

[47] Shumian Xin, Sotiris Nousias, Kiriakos N Kutulakos, Aswin C Sankaranarayanan, Srinivasa G Narasimhan, and Ioannis Gkioulekas. A theory of fermat paths for non-line-of-sight shape reconstruction. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 6800–6809, 2019.

[48] Jie Xiong and Kyle Jamieson. Arraytrack: A fine-grained indoor location system. In *Proc. of NSDI*, pages 71–84, 2013.

[49] Mao Ye, Yu Zhang, Ruigang Yang, and Dinesh Manocha. 3d reconstruction in the presence of glasses by acoustic and stereo fusion. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4885–4893, 2015.

[50] Sangki Yun, Yi chao Chen, and Lili Qiu. Turning a mobile device into a mouse in the air. In *Proc. of ACM MobiSys*, May 2015.

[51] Sangki Yun, Yi-Chao Chen, Huihuang Zheng, Lili Qiu, and Wenguang Mao. Strata: Fine-grained acoustic-based device-free tracking. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, pages 15–28. ACM, 2017.

[52] Zengbin Zhang, David Chu, Xiaomeng Chen, and Thomas Moscibroda. Swordfight: Enabling a new class of phone-to-phone action games on commodity phones. In *Proc. of ACM MobiSys*, 2012.

[53] Zhengyou Zhang. Microsoft kinect sensor and its effect. *IEEE multimedia*, 19(2):4–10, 2012.

From Conception to Retirement: a Lifetime Story of a 3-Year-Old Wireless Beacon System in the Wild

Yi Ding
Alibaba Group, University of Minnesota

Ling Liu
Shanghai Jiao Tong University

Yu Yang
Rutgers University

Yunhuai Liu
Peking University

Desheng Zhang
Rutgers University

Tian He
Alibaba Group, University of Minnesota

Abstract

We report a 3-year city-wide study of an operational indoor sensing system based on Bluetooth Low Energy (BLE) called `aBeacon` (short for alibaba Beacon). `aBeacon` is pilot-studied, A/B tested, deployed, and operated in Shanghai, China to infer the indoor status of Alibaba couriers, e.g., arrival and departure at the merchants participating in the Alibaba Local Services platform. In its full operation stage (2018/01-2020/04), `aBeacon` consists of customized BLE devices at 12,109 merchants, interacting with 109,378 couriers to infer their status to assist the scheduling of 64 million delivery orders for 7.3 million customers with a total amount of $600 million USD order values. Although in an academic setting, using BLE devices to detect arrival and departure looks straightforward, it is non-trivial to design, build, deploy, and operate `aBeacon` from its conception to its retirement at city scale in a metric-based approach by considering the tradeoffs between various practical factors (e.g., cost and performance) during a long-term system evolution. We report our study in two phases, i.e., an 8-month iterative pilot study and a 28-month deployment and operation in the wild. We focus on an in-depth reporting on the five lessons learned and provide their implications in other systems with long-term operation and large geospatial coverage, e.g., Edge Computing.

1 Introduction

Instant delivery is an emerging business where online orders (e.g., groceries or foods) are delivered within a short time (e.g., 30 mins) from merchants (e.g. grocery stores and restaurants) to customers. This business grows rapidly in recent years with the emergence of several online platforms, e.g., Prime Now [6], Uber Eats [50], Instacart [26], DoorDash [16], Deliveroo [14], and Alibaba Local Services [17]. In an instant delivery service, a customer uses an APP on a platform to place an order at a merchant; the platform assigns a courier to pick up this order at the merchant and then deliver it to the customer. It is essential for the platform to know its couriers' real-time arrival status at merchants, which is used to assign new orders to the most suitable couriers based on their locations to avoid an order delivery overdue given short delivery window [57]. While the outdoor status of couriers can be obtained by smartphone GPS, inferring the indoor status is always challenging due to a lack of infrastructure at scale.

In this paper, we report a 3-year study for a system named `aBeacon` developed by Alibaba Inc. [5] in Shanghai to infer its couriers' indoor status, i.e., arriving and departing at merchants. The indoor status inference is of great significance for Alibaba Local Services (a subsidiary of Alibaba Inc. for instant delivery), since couriers spend almost one third of total working time indoor based on our data. The goal of `aBeacon` is to provide a city-wide indoor sensing solution with practical cost/performance tradeoffs when deploying in the wild. *We share one-month data of `aBeacon` for future research*[1] [1].

Admittedly, indoor arrival and departure status detection is not technically challenging and has been widely investigated in controlled environments, e.g., labs, museums, and airports. However, it is still an open question for city-wide detection in the wild. In industry, current solutions mainly rely on either courier's smartphone GPS (which is inaccurate in indoor environments) [29] or manual reporting (which suffers from intentional or unintentional human errors). In academia, the solutions are based on Wi-Fi [11,13,23,31,39], LED fixtures [32,49,52,54], and RFID [2,51]. However, each of them has limitations for a city-wide deployment with more than 12,000 merchants and 109,000 couriers with only commodity smartphones. Wi-Fi based solutions are limited because continuous scanning is required to keep the Wi-Fi list updated, which brings much extra power consumption for courier's smartphones, and more importantly, for merchants without Wi-Fi Access Point devices, it is costly to deploy new ones [9,30,33,41]. LED solutions do not scale up due to hardware modification costs [49]. RFID solutions require additional equipment on both receivers and transmitters.

In this work, we argue the Bluetooth Low Energy (BLE) device [15,19,24,56] is a promising solution to achieve our goal. BLE is not a new technology, and BLE-based iBeacon was introduced by Apple [25] in 2013. However, the new features provided in BLE 5.0 [45] in 2016 (e.g., longer range and faster speeds) offer us the opportunity to build `aBeacon` starting from 2017/05. We deploy 12,109 customized `aBeacon` devices to 12,109 merchants on Alibaba platform in Shanghai. An `aBeacon` device is a low-cost ($8 USD) broadcast-only BLE device, and does not have GPS or cellular/ Wi-Fi connections, so it cannot receive any update, and it also cannot directly communicate with back-end servers. An `aBeacon` device deployed in a merchant constantly broadcasts its ID tuple (UUID, major, minor) following the BLE protocol, which will be received by couriers' smartphone APP if in proximity

[1] https://tianchi.aliyun.com/dataset/dataDetail?dataId=76359

and then uploaded to a server by APP using smartphones' Internet connection. Based on the uploaded ID tuple, the server is aware of the couriers' arrival to this merchant given previously-mapped device-merchant pairs in the deployment.

BLE devices have several advantages. Continuous scanning in BLE only introduces less than 2% extra power consumption on couriers' smartphones based on our experiments, which is much less than Wi-Fi [7]; compared to RFID-based solutions, no hardware modification is needed on the courier end, since `aBeacon` only requires a courier to have a smartphone; compared to LED, battery-powered BLE devices can be installed in many places due to their small size and portability. We note that a key limitation of `aBeacon` is we need to deploy an `aBeacon` device at every merchant, which introduces both hardware and deployment costs. However, our deployment has a low cost since we utilize an Alibaba in-house team and its members visit merchants periodically for business development; the hardware cost of `aBeacon` can be managed if we only remain core functions, e.g., no GPS, no cellular/Wi-Fi, and no Over-The-Air (OTA) updates.

In a control environment, using BLE devices to detect arrival and departure is straightforward. However, it is nontrivial to build, deploy, and operate `aBeacon` from the ground up, considering the tradeoffs between various practical factors, e.g., cost and performance, in a **metric-based approach**. BLE devices are already applied in real-world applications, e.g., interaction in museums [34] and indoor localization in airports [47]. However, we argue that these indoor environments are normally under the control of BLE system operators. Still, the indoor environments for instant delivery (e.g., shopping mall) are not under the company's control, i.e., in the wild. To our knowledge, there are few studies, if any, on a practical city-wide BLE device deployment in the wild. We introduce `aBeacon` based on Alibaba Local Services for courier indoor status monitoring (i.e., arrival and departure) in a 36 month two-phase study from 2017/5 to 2020/4.

- **Phase I: 8-Month Iterative Pilot Study (2017/5-12)**. We deployed three types of commodity BLE devices in 18 merchants and built an APP to test the feasibility of BLE. Based on the promising results, we customized `aBeacon` devices for lower cost and new functions. We deployed one customized `aBeacon` device and one commodity device in 200 merchants to A/B test their performance.
- **Phase II: 28-Month Deployment and Operation in the Wild (2018/1-2020/4)**. We deploy and operate 12,109 `aBeacon` devices in Shanghai with one device in each merchant. In this phase, `aBeacon` interacts with 109,378 couriers to provide their status to assist the scheduling of 64 million delivery orders for unique 7.3 million customers with a total amount of $600 million USD order values.

As of 2020/4, `aBeacon` is being retired and replacing by a new system `aBeacon+` (introduced in the Discussion section). In this Operational Systems track submission, we focus on 5 lessons we learned in our 3 year study of `aBeacon` from its conception to retirement to provide new insights for the existing design assumptions based on our successes and failures.

Lesson learned 1: Explicitly Quantifying the System Gain. During our interaction with the Alibaba executives team who makes the decision to fund `aBeacon`, we utilize a metric-based approach to quantify `aBeacon`'s monetary gain (i.e., benefit minus cost) to justify `aBeacon`. In particular, we explore the fundamental tradeoff between cost and benefit (proportional to its performance) to optimize the gain of `aBeacon` by (i) reducing the cost by customizing new devices and utilizing our in-house team without technical expertise for configuration-free deployment, and (ii) increasing the benefit by improving lifetime, reliability, and utility. We study the system gain in an evolving cumulative fashion at the fine-grained device level. `aBeacon` achieves the break-even point where its benefit is equal to its cost after 12 months of the deployment, and then generate 14 months of benefits. In retrospect, a batch deployment, instead of an "one-shot" deployment, could make `aBeacon` break even earlier.

Lesson learned 2: System Scale Evolution in the Wild. Even though a device has an expected lifetime of 24 months, `aBeacon`'s scale (i.e., number of live devices) has been constantly shrinking, immediately after fully deployed in the wild, for the entire 26 months of the operation. In particular, the decrease is steady in the first 20 months due to various factors (e.g., deployment, hardware, and merchants closed) yet with a stable citywide spatial coverage; whereas the decrease is dramatic in the last 6 months due to clustered device battery run-outs. This observation has the potential to provide some guidance on the re-deployment strategies (e.g., timing and priority) to keep the system scale and a positive gain (as suggested in the Lesson Learned 1), e.g., large-scale re-deployment much earlier than expected battery lifetime. In retrospect, `aBeacon`'s scale shrinking is much worse than our expectation, making us rethink the initial rationale of deploying physical devices in the wild. It motivated us to virtualize the next generation of `aBeacon`, i.e., `aBeacon+`.

Lesson learned 3: Lifetime in the Wild. During the `aBeacon` operation, the lifetime of 42% devices is longer than deployment environment (e.g., a device is live but the merchant it was deployed is closed). However, once deployed in the wild, large-scale recycling of low-cost ($8 USD) devices from these short-lifetime environments is not practical due to significant labor. In retrospect, `aBeacon` devices could be designed with different energy modules for different environment lifetime (e.g., predicted based on our order data) to minimize the hardware cost.

Lesson learned 4: Reliability in the Wild. Many existing sensing systems (e.g., proximity [36], gesture [58], and breath [55]) are mainly tested in control environments with high reliability [22]. However, we found that even for simple arrival detection in `aBeacon` the reliability is heavily affected by many real-world factors including smartphone diversity (e.g., 52 phone brands and 672 phone models in our platform), device placement (e.g., non-expert installation), and courier mobility behaviors. In retrospect, we could add an OTA update function to some devices (but not all devices) deployed in uncertain environments, and utilize couriers' phones to update them, e.g., increasing their transmission powers.

Lesson learned 5: Utility in the Wild. Unlike other infrastructures, e.g., Wi-Fi, we found that in our aBeacon operation, the locations with more interactions between couriers and devices may not have higher device deployment utility (quantified by the order delivery rate improvement based on courier detection). In contrast, the locations with higher uncertainty of courier mobility behaviors (e.g., higher floors) lead to a higher utility. In retrospect, we could change our deployment strategies to prioritize more uncertain environment.

Based on the above lessons learned, we discuss our limitations and potential applications of aBeacon and then discuss their implications to other systems with long-term broad geospatial coverage (e.g., Edge Computing), and finally share the direction of our ongoing work aBeacon+.

2 aBeacon Design Goal

In aBeacon, a generic workflow is as follows: (1) devices deployed in indoor merchants to continually broadcast their ID tuples; (2) an embedded BLE scanning module in the Alibaba couriers' smartphone APP (mandatory for all couriers) to receive these ID tuples from devices when in proximity and to upload them to a back-end server using the smartphone Internet connectivity; (3) The server updates couriers' arrival and uses them for various functions, e.g., new order scheduling. Based on this workflow, we introduce our metrics as follows.

2.1 Cost and Performance Metrics

Cost C_{Dev}: The costs of a device in aBeacon mainly consist of the hardware cost and the deployment cost (i.e., the shipping and labor cost to deploy a device at a merchant).

Lifetime P^i_{Life}: In our design, we envisioned a lifetime of a device for two years, and then redeploy new devices after two years if (i) aBeacon was successful (Yes); (ii) the deployment cost was still low (Yes); and (3) aBeacon was still the best solution (No since we have aBeacon+). The lifetime of a device i is affected by the design (e.g., battery) and the environment (e.g., the deployed merchant is closed).

Reliability P^i_{Reli}: We quantify a device i's reliability by the percentage of couriers we detected among all arrived couriers. The ground truth of the courier arrival is obtained by the delivery order accounting data. P^i_{Reli} is affected by device deployment, smartphone diversity, and courier mobility.

Utility P^i_{Util}: We quantify the utility of a device i by *overdue delivery rates reduction* for the merchant after i was deployed in it. After a merchant was deployed with a device, the platform can better detect and predict the status of couriers around this merchant, which are used to schedule new orders for this merchant by finding nearby couriers (e.g., a courier just left), thus reducing the overdue rate for this merchant. P^i_{Util} is affected by a merchant's features where i was deployed (e.g., merchant locations, floors).

Table 1: Metric Summary

C_{Dev}:	cost of a device, i.e., hardware & deployment
C_{Over}:	cost of overdue penalty per order, e.g., \$1.
P^i_{Life}:	lifetime of a device i
P^i_{Reli}:	reliability of i
P^i_{Util}:	utility of i
t^i_0:	day of i was deployed
T:	# of days since aBeacon deployed
N_t:	# of deployed devices until the tth day
O^i_t:	# of orders at tth day in the merchant with i

2.2 Metric-based Approach for Trade-offs

We utilize a metric-based approach to explore the trade-off between costs and performance by Eq. (1). Assuming it has been T days since aBeacon was deployed, the cumulative aBeacon gain G_T is given by the difference of (i) the cost C_T of deploying aBeacon until the Tth day; and (ii) the benefit (i.e., monetary saving) brought by performance improvement, i.e., overdue reduction due to better detection by aBeacon.

$$G_T = \sum_{t=1}^{T} \sum_{i=1}^{N_t} B^i_t - \underline{C_T} \tag{1}$$

where N_t is the number of devices deployed until the tth day ($t \leq T$) including live and dead devices. $\underline{C_T} = N_T \cdot C_{\text{Dev}}$ is the cost of all devices until Tth day where C_{Dev} is a device cost. B^i_t is the **Benefit** of a device i in the tth day as

$$B^i_t \;=\; \underline{F_1(\,P^i_{\text{Life}},\, t,\, t^i_0\,)} \;\cdot\; \underline{F_2(\,O^i_t,\, P^i_{\text{Reli}},\, P^i_{\text{Util}},\, C_{\text{Over}}\,)}. \tag{2}$$

$\underline{F_1(P^i_{\text{Life}},t,t^i_0)}$ indicates whether or not a device i reached its lifetime limit by the tth day. It was calculated by remaining lifetime $P^i_{\text{Life}} - (t - t^i_0)$, where P^i_{Life} is the lifetime of i; t^i_0 is number of days that device i has been deployed. $F_1(P^i_{\text{Life}},t,t^i_0) = 1$ if $P^i_{\text{Life}} - (t - t^i_0) \geq 0$; $F_1(\cdot) = 0$ otherwise, i.e., no remaining lifetime, so we do not have to consider F_2. $\underline{F_2(O^i_t,P^i_{\text{Reli}},P^i_{\text{Util}},C_{\text{Over}})}$ indicates the monetary saving by reduced overdue penalty of the orders detected by i. O^i_t is the number of orders at the tth day in a merchant with i, e.g., 100; P^i_{Reli} is the percentage of the orders whose couriers can be detected by i, e.g., 80%; P^i_{Util} is the reduced overdue rate (compared to the overdue rate before the device was deployed) for all orders whose couriers are detected by i, e.g., 20%; C_{Over} is the overdue penalty per order, e.g., \$1. An example of F_2 is the product of all these terms, i.e., $O^i_t \cdot P^i_{\text{Reli}} \cdot P^i_{\text{Util}} \cdot C_{\text{Over}}$ (e.g., saving is $100 \cdot 80\% \cdot 20\% \cdot \$1 = \$16$).

3 aBeacon Life Cycle Overview

Unlike the wireless systems (e.g., Smart Home IoT) that can be updated by OTA, an aBeacon device was not designed to be updated after customization to save the hardware cost. Thus, separated by the time we finished the customization

Table 2: Overview of Two Phases

Phase / Stage & Scale / Goal		Phase I: 8-month Pilot Study (2017/5 – 2017/12)		Phase II: 28-month Operation in the Wild (2018/1 – 2020/4)	
		Conception Stage (2017/5-8) 18 merchants,54 devices	**Customization Stage** (2017/8-12) 200 merchants, 400 devices	**Deployment Stage** (2018/1-3) 12,109 merchants & devices	**Operation Stage** (2018/3-2020/4) Evolving
Cost	Hardware	Commodity (Fig.1)	Commodity ($11, Fig.1, T4) Customized ($8, Fig.2)	Customized	—
	Deployment	Our Team (Fig.1)	Our Team	302 Business Managers	—
Performance	Lifetime	Fig. 1	Commodity (2-3 yrs. advertised) Customized (2 yrs. designed)	*In retrospect, we should have selected the merchants with longer lifetime*	Fig. 4 & 5
	Reliability	98%	Both are Close to 98%	Installation Handbook Provided	Fig. 6-8 & Table 5
	Utility	—	—	Highly Profitable Merchants Selected	Fig. 9-12

Size Comparison	Device 1 (T15)	Device 2 (T4)	Device 3 (T11)
Tx Power	-59 dB	-65 dB	-65 dB
Advertised Lifetime	≤ 3 yr	$2 \sim 3$ yr	≤ 3 yr
Cost	$11 each	$10 each	$10 each
Encapsulation	—	Water, Dust, Shock Proof	Dust Proof Only

Fig 1: Deployment for Conception Stage

(i.e., 2018/01), we divide the entire 3-year study of `aBeacon` into two phases, i.e., Phase I: 8-Month Pilot Study; and Phase II: 28-month Deployment and Operation.

As in Table 2, we carefully designed the stage, scale, cost, and performance to serve each phase's purposes.

3.1 Phase I: 8-Month Pilot Study (2017/5-12)

In this phase, we performed two studies in a conception stage to investigate three commodity devices, and a customization stage to design and evaluate new devices with A/B testing.

(i) Conception Stage (2017/5-8): As shown in Table 2, we aim to understand whether a BLE device system can detect the couriers' indoor arrival and departure with reliability higher than 95%. We bought 54 commodity devices in three brands and deployed them in 18 merchants of a shopping mall in Shanghai. Each merchant was equipped with three commodity devices of different brands, as shown in Fig.1 with technical specifications. We set some key configurations of couriers' mobile APP when interacting with commodity devices as parameters for further developing, e.g., scanning duration and intervals, data upload cycle, and working hours. Note that the couriers' APP and the back-end server developing were also the major works in this stage, but we omit them in this paper since they are standard. After this study, we had average reliability of 98%, so we concluded that a beacon-based solution could achieve high reliability.

Table 3: BLE Chip Comparison

BLE Chip	Link Budget	Tx Power Consump. (curr. at 0dB)	Sleep Power Consump. (curr.)	Price $/unit
CC2540 [27]	97 dB	21 mA	0.9 ua	~1.1
DA14580 [43]	93 dB	12.4 mA	0.5 ua	~1.1
CSR1010 [40]	93 dB	18 mA	5 ua	~1.1
nRF51822 [44]	**96 dB**	**8.06 mA**	**2.6 ua**	**~1.1**

(ii) Customization Stage (2017/8-12): Instead of using commodity devices, we customized our `aBeacon` device for low cost ($8 per device) and longer lifetime. We performed a middle-scale A/B testing between the best one among three commodity devices and our customized device. As in Table 2, after the reliability had been proved in the previous stage, our customization was focused on the hardware cost and lifetime since the large-scale city-wide deployment cost in Phase II is marginal when we utilize our in-house business team. In our customization, three components, i.e., BLE chip, battery, and casing, were carefully customized to achieve overall lower cost and longer lifetime. (1) For the BLE chip, we compared the mainstream BLE chips as in Table 3. Since our BLE devices in `aBeacon` were expected to broadcast for at least two years without external power continuously, we chose the nRF51822 from Nordic Semiconductor as the BLE chip since it has both the minimum Tx power and acceptable other configurations. (2) For the battery, we considered both the lithium battery and the alkaline battery since we expected an `aBeacon` device could operate for at least two years without maintenance. The lithium battery usually has a smaller size, but the alkaline battery has a much better unit capacity (mAh/$), so we used two alkaline AA batteries in cascade . (3) For the casing, we considered dust, water, and shockproof for transportation and operation in various indoor (or potential future outdoor) operations. Finally, we built 200 `aBeacon` devices, as shown in Fig.2. We A/B tested our 200 customized

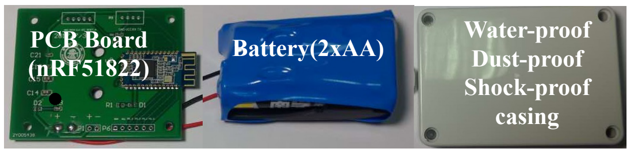


Fig 2: Customized Hardware for `aBeacon`

devices with 200 commodity devices (i.e., Device 2 (T4) [10] in Fig.1). We selected 200 merchants in two malls and placed one customized device and one commodity device side by side to compare their performance. After 2-month testing, we concluded our customized devices are ready for deployment and operation because they have similar reliability with the commodity devices, but have a lower hardware cost and potentially longer lifetime.

3.2 Phase II: 28-month Operation (18/1–20/4)

We introduce a 3-month deployment and a 25-month operation stage of 12,109 devices in Shanghai (visualized in Fig.3.)

(i) Deployment Stage (2018/1-3). After we received all `aBeacon` devices from a manufacturer, we aim to deploy them in 12,109 chosen merchants among 57,223 merchants in Shanghai after consulting with our accounting department to understand these merchants' profitability, potentially decides our `aBeacon`'s utility. We decided to deploy around 12 thousand devices for `aBeacon` because of the approved $100,000 budget, i.e., the system cost. Assuming no benefit at all, based on Eq. (1), our system gain G_T is $-\$100,000$ (i.e., the trivial lower bound in Fig.3 (iii)). In Phase I, our team deployed 200 devices by ourselves, but 12,109 devices were out of our team's capability. As a result, we utilize our in-house regional business development managers who periodically visit all merchants for regular business meetings to install our `aBeacon` device. We mailed our `aBeacon` devices and guided them for `aBeacon` device deployment and mapping between `aBeacon` devices and merchants by a detailed handbook, which shows "Where to attach the device?" (e.g., main entrance), "How to attach the device?" (e.g., double-sided tape) and "How to map the device?". The mapping was achieved by scanning a QR code on an `aBeacon` device and then choosing its merchant from a given merchant list in a business manager APP. 302 managers participated in our deployment process, and it took us around two months to deploy all the devices after one month of shipping and logistics. Since our business managers deploy our devices for free, the main deployment cost is the shipping cost, which is around $1 per device.

(ii) Operation Stage (2018/3-2020/4). After the deployment, `aBeacon` is fully operational, and we have been monitoring its status and utilizing it to detect couriers remotely based on the data we collected from couriers' APPs. We embed the device monitoring function in the official smartphone APP of 109,378 Alibaba couriers in Shanghai. When we first receive an `aBeacon` device ID tuple from a courier's phone,

Table 4: Operation Data Collected

(a) aBeacon monitoring data

Attribute	Example	Attribute	Example
Courier ID	C_000001	RSSI	-80dB
Timestamp	2019/08/15 12:30:23	Phone ID	D_000001
Device ID Tuple	(UUID, Major, Minor)	Phone Brand/OS	Apple/iOS
Merchant ID	M_000001	Phone Model	iPhone X

(b) Courier GPS data

Attribute	Example
Courier ID	C_000001
Timestamp	2019/08/15 12:30:23
Latitude	31.243715
Longitude	121.245847
Speed	3.7 km/h
Altitude	40.2 meters

(c) Courier order report data

Attribute	Example
Courier ID	C_000001
Timestamp	2019/08/15 12:30:23
Merchant ID	M_000001
Order ID	O_000001
Report Type	Acceptance/Arrival/ Departure/Delivery

we need to make sure this live device is correctly deployed and works properly. For all devices, their initial status on our server end is "Not Deployed"; once a manager completes the mapping operation on her APP, a "Not Deployed"device becomes "Online". For all "Online", we use order accounting data to validate if the deployment is correct or not indirectly: (1) if a device is heard by more than three couriers whose current orders or GPS would not let them pass the merchant mapped to this device, this "Online" device would be changed to "Wrongly Deployed"; (2) if no ID tuples were received from a device for 24 hours, and if the mapped merchant still has orders during these 24 hours (e.g., more than ten orders), then this "Deployed" device would be considered as "Offline" or "Retired" based on its expected lifetime is reached or not since deployment; (3) if ID tuples were received from a device, but the merchant was closed (based on our accounting data), it would be considered "Closed".

(iii) Operation Data Collected. During our operation, we collected three kinds of data sets to monitor and validate `aBeacon`. **(a) `aBeacon` Monitoring Data**. As in Table 4(a), every time an `aBeacon` broadcast was received by a courier's phone, we recorded the information of the `aBeacon` device, phone, and the Received Signal Strength Index (RSSI) of the broadcast. **(b) Courier GPS data**. As in Table 4(b), GPS data were collected under courier consent since customers also like to know where his/her order is, and the platform needs to know couriers' locations for order assignment. **(c) Courier Order Report Data**. As in Table 4(c), for each delivery order, the courier needs to report when he/she arrives at or leaves from the merchants manually for real-time order status updates. These report data are used as the ground truth for `aBeacon` detection. However, in our previous study, we found couriers often forgot to report their status and exaggerate their status (e.g., early reporting) to game the scheduling system for better order assignment. That is why these report data can only be used as post-hoc ground truth, i.e., we know that a courier arrived at a merchant after an order was delivered since a courier often forgets or falsely reports their arrival. Please

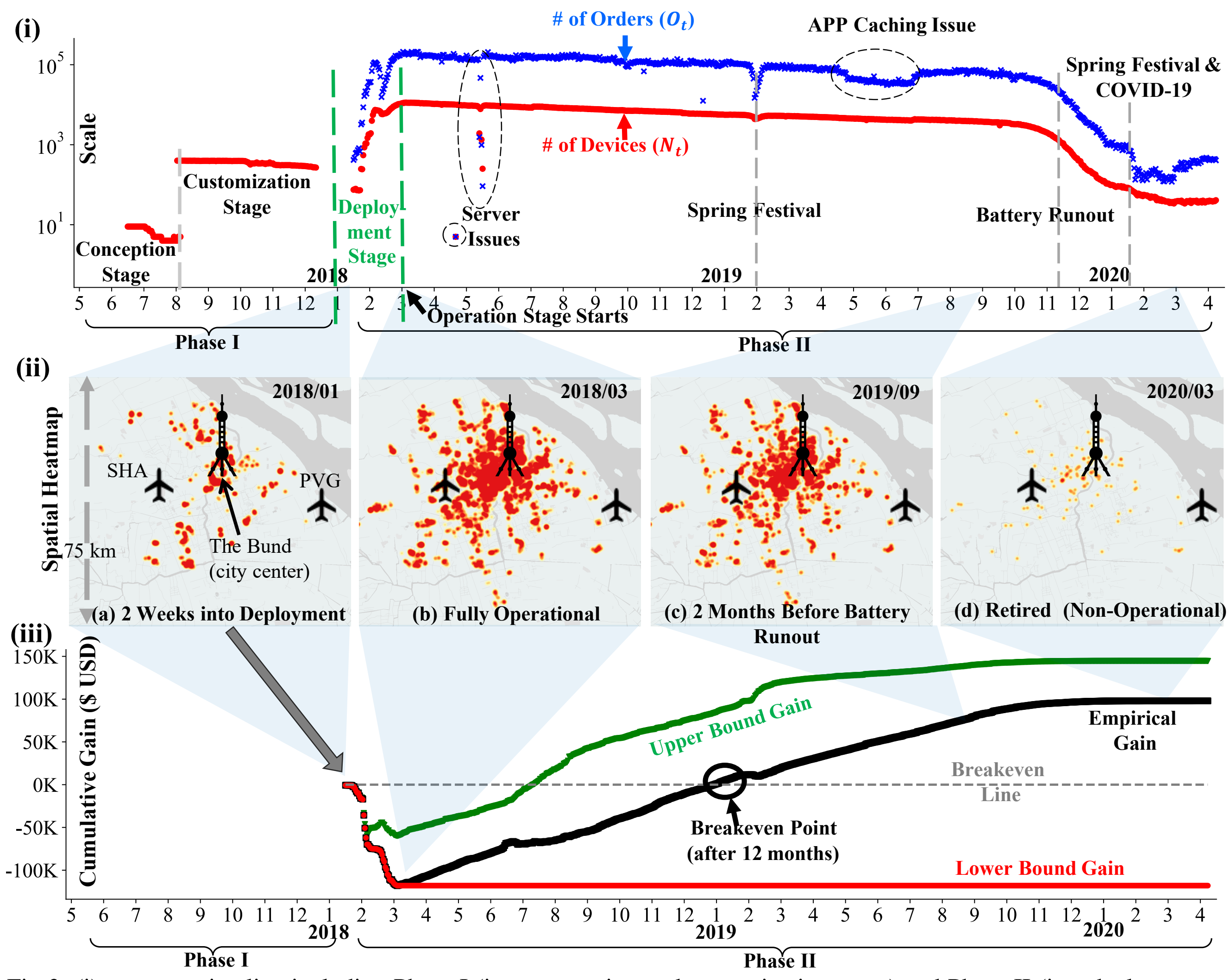


Fig 3: (i) aBeacon timeline including Phase I (i.e., conception and customization stage) and Phase II (i.e., deployment and operation stage); (ii) aBeacon device heatmaps in Shanghai for four key stages; the background shapes are expended to the corresponding time in (i) and the system gains (iii); (iii) Three aBeacon gains with three kinds of utility.

see our Discussion section for details on using aBeacon for Anomaly Detection.

4 aBeacon Operation Results

4.1 Result Overview

In Fig.3, we show a panorama of aBeacon life cycle with our two phases from 2017/5 to 2020/4.

Quantitative System Evolution Overview: In Fig.3 (i), given a day t, we show both the number of aBeacon devices N_t with "Deployed" status in t and the number of delivery orders O_t whose couriers are detected by aBeacon in t. We omit the number of couriers detected since it is highly correlated with the number of delivery orders. The detailed analysis on N_t and O_t in Sec.4.3, but we would like to highlight two technical incidents affecting both N_t and O_t as indicated by three circles in Fig.3 (i). On May 16th and 21st, 2018, a configuration exception occurred on the APP server and led to data loss, and our team diagnosed and fixed it quickly. In May 2019, we found an unusual decrease in detected orders, which took our team around two months to diagnose the root cause, i.e., a caching problem in the courier APP of some phone brands. In particular, since the courier APP is not always connected to the server, it would cache some received aBeacon device data if the network connection is unavailable. But when the local cache was full, received data got lost without exceptions raised in some smartphones brands. By the end of June, the problem was fixed, and the detected orders increased.

Qualitative Spatial Coverage Evolution: In Fig.3 (ii), we visualize the aBeacon spatial evolution in Shanghai at four critical periods. (a) 2018/01: 2 weeks into the deployment stage where aBeacon has not been uniformly deployed; (b) 2018/03: aBeacon is fully operational, reaches its spatial scale peak, and covers all the central business districts in Shanghai; (c) 2019/09: aBeacon has been operating for 20 months, and the spatial cover remains relatively similar, and it is two months away from the starting of clustered battery run-out; (d) 2020/03: aBeacon drops below a critical level and is being retired and replaced by aBeacon+ (see the Discussion section). We found even the scale of aBeacon has been shrinking right after the full deployment due to various real-world issues, the spatial coverage has been relatively stable for 20 months. Based on our field study, we found the most of the dead devices are related to merchants closed, deployment imperfection, hardware malfunction, and vandalism. It provides some practical guidelines for our current project of the next generation of aBeacon, i.e., aBeacon+.

4.2 System Gain Evolution

System Gain Overview: In Fig.3 (iii), we utilize Eq. (1) to show the system gain, i.e., the monetary saving minus the system cost. All metrics in Eq.(1) can be directly measured by our aBeacon data except the system utility P^i_{Util}. We show three cumulative gains (defined in Sec.2.2) based on the empirical value of system utility P^i_{Util} (overdue rates reduction after device i was deployed, discussed in Sec.4.6), along with its lower bound $\underline{P}^i_{\text{Util}}$ (no overdue reduction at all) and its upper bound $\overline{P}^i_{\text{Util}}$ (complete overdue reduction), respectively. We found aBeacon achieved a break-even point after 12 months , which provides empirical guidance for our aBeacon+. Some additional applications of aBeacon for the Alibaba group are shared in the Discussion section.

Lesson Learned 1: **Explicitly Quantifying the System Gain.** Even though the cost of a real-world system can be often explicitly quantified, the benefit of a system is often hard to be, which makes the justification of deploying a system challenging when convincing the decision-makers. Based on our interactions with the Alibaba executive team, who made decisions to initiate and fund aBeacon, we utilized a metric-based approach to quantify the cumulative system gain to justify aBeacon development. In particular, we explore the cumulative system gain by (i) reducing the cost by customizing new devices (e.g., 20% less than commodity devices yet with more functionality) and utilizing our Alibaba in-house business development team without technical expertise for large-scale deployment due to our configuration-free setting, and (ii) increasing the performance by extending device lifetime, improving reliability, and enhancing utility. As shown in Fig.3 (iii), aBeacon achieves a break-even point after 12 months. In retrospect, a few approaches could be used to make sure aBeacon achieves break-even earlier. The most promising one is a batch deployment instead of a “one-shot" deployment in a short time, which have been used in our other physical device deployment projects.

In-depth System Gain Investigation Overview: To provide an in-depth investigation on the cumulative system gain, we analyze seven metrics in Eq.(1) and (2) individually: (i) C_{Dev} and C_{Over} are the individual device cost and the order overdue penalty, which are almost fixed in our setting; (ii) N_t and O_t are related to the system scale and we study them in Sec. 4.3; (iii) P^i_{Life}, P^i_{Reli}, P^i_{Util} are related to the system performance in terms of lifetime, reliability, and utility, which are studied in Sec. 4.4, 4.5, and 4.6, respectively. The correlation between different metrics is introduced in Sec. 4.7.

4.3 Scale Metric: Number of Device & Order

Scale Metric 1: Number of Devices N_t. In Fig.3 (i), starting from our deployment stage in Phase II, the number of aBeacon devices increased significantly until the end of our deployment stage in 2019/3. However, after aBeacon scale peaked in 2019/3, two decreasing trends are observed. (1) The first one is a slow decrease throughout the major part of Phase II from 2018/3 to 2019/10, where we lost some devices every day. In addition to vandalism, deployment, and hardware issues, the primary reason is that some merchants terminate their business with Alibaba every day. The merchant turnover rate in China online platforms is high, and almost 70% of new merchants were closed within one year of the opening in 2017 [20]. We report our empirical merchant lifetime data in Fig.4 and analyze it in detail later. (2) The second one is the sharp decrease from 2019/11 to 2020/2, due to the clustered battery running out after 20 months of operations since 2018/3. Such an observation provided some insights about our potential re-deployment strategies, which we will discuss in the Lesson Learned 2.

Scale Metric 2: Number of Orders O_t. As shown in the Cumulative System Gain Eq.(1), the number of orders O_t whose couriers were detected by aBeacon is the central part of deciding the gain of aBeacon. In Fig.3 (i), we found in the full operation stage of Phase II (from 2018/3 to 2019/11), the number of orders detected is around ten times the number of aBeacon devices, which implies each device serves ten orders on average every day. This ratio remains similar throughout Phase II except for the mid-February, during which the overall number of orders decreases sharply. Mid-February is typically the Chinese Spring Festival, i.e., the biggest holiday where the number of total orders reduced since many merchants closed during this time. We observed sharp decreases and recoveries during February of 2018, 2019, and 2020 in Fig.3 (i), and the corresponding impact on the system gain in Fig.3 (iii). In 2020, the impact of COVID-19 lasts after February, so we do not see an apparent recovery at the end of February.

Lesson Learned 2: **System Scale Evolution in the Wild.** The scale of aBeacon (quantified by the number of devices N_t and the number of associated orders O_t) is essential to ensure the cumulative system gain. As in Fig.3 (i), after fully deployed in the wild (2018/3), aBeacon scale has been continuously shrinking for 26 months until 2020/4, even though devices have an expected lifetime of 24 months. In particular, the decrease is steady in the first 20 months (from 2018/3 to 2019/10) due to various factors (e.g., vandalism, hardware malfunction) yet with a stable city-wide spatial coverage in Shanghai (Fig.3 (ii)). In contrast, the decrease is quite sharp in the last six months (from 2019/10 to 2020/4) due to clustered battery run-out. It suggests that if we want to keep the system scale, we should start a full-scale re-deployment much earlier than expected, or perform batch-based small re-deployment continuously if we want to keep `aBeacon` at scale. However, we did neither of them in practice since we move on to a new system `aBeacon+` without deployed devices as introduced in future work. Further, by using the number of orders as a bridging factor, our results also provided some insights on how to link the traditional system scale (i.e., number of devices) to the business revenue (i.e., reduced order overdue penalty) to justify their potential correlation. The insights help us communicate with the Alibaba executive team when reporting the impact of `aBeacon` on the overall Alibaba ecosystem.

4.4 Performance Metric 1: Lifetime P^i_{Life}

Lifetime Overview: The lifetime of an `aBeacon` device is decided by two primary factors: (i) the battery size, which was considered in the customization stage of Phase I when we design our hardware; (ii) the merchant lifetime, which unfortunately was not considered in the deployment stage of Phase II as shown in Table 2 since we mainly consider the profitability of merchants. In our defense, there should be a strong correlation between the profitability and lifetime of a merchant, but we found that the profitability of many merchants has been rapidly changing, especially on the online platform. Further, given the high real estate rental fees in Shanghai, many merchants move their physical stores frequently. If a merchant deployed with an `aBeacon` device is closed or moved, there is a high chance that the deployed device would be thrown away. The CDF of devices' and merchants' lifetime are given in Fig.4. Around 23% and 40% of merchants closed within 1 or 2 years, respectively; whereas more than 50% of devices died within one year, much less than the expected lifetime based on battery alone.

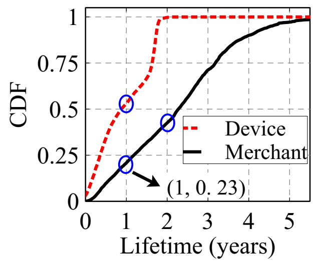


Fig 4: CDF of Device Lifetime and Merchant Lifetime

Lifetime Correlation: An in-depth visualization of the correlation between these two factors is scatter-plotted in Fig.5 where we record the last day a device i was heard as X-axis; the last day the corresponding merchant had orders on our platform as Y-axis. We have three observations: (1) points (15%) around the diagonal ($|x-y| \leq 14$, i.e., 14 days) suggest devices died within two weeks of the closure of the corresponding merchants on our platform; (2) another cluster of points (17%) is around $x = 640$, which means a device is dead after 21 months of operation, as observed in Fig.3 (i). (3) for the points above the diagonal (26%, $x < y$), it indicates the merchant has active orders from our platform, but the `aBeacon` device is dead; for the points below the diagonal (42%, $x > y$), it indicates the merchant closes on our platform (i.e., no orders) but the device can still be heard by couriers, i.e., the device may be in the original locations or nearby, and can be heard when our couriers in proximity. We note that a merchant has no orders on our platform does not necessarily mean the merchant is closed, but it can be used to approximate the merchant's lifetime on our platform. For a closed merchant, an intuitive idea is to recycle the device for re-deployment, but in practice, we did not do it due to two reasons: (1) the platform is not generally informed in advance when the merchant is closing so we cannot prepare in advance to recycle the device; (2) the device recycling introduces significant labor and shipping costs, and the recycled devices may be damaged or with low battery, which makes purchasing a new device a better choice overall. As a result, we did not perform large-scale device recycling in practice.

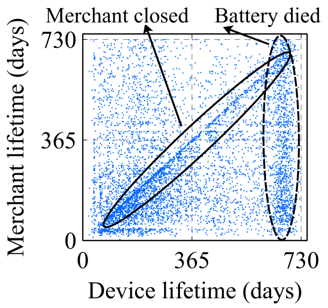


Fig 5: Last day a Device was Heard and Last day the Corresponding Merchant has Orders

$\boxed{\textbf{Lesson Learned 3:}}$ **Lifetime in the Wild.** The lifetime of 42% devices is longer than the lifetime of their deployed environment (e.g., merchants). It provides new insights on our design assumption on mobile device energy since a longer battery life may not increase the device lifetime due to uncertainty of the deployed environment but introduce higher costs. This lesson is especially true when the large-scale device recycling and re-deployment are not practical due to higher labor cost. It motivated us to design devices with different battery capacity and then deploy devices in batches to accommodate the environment's lifetime, which can be predicted by our platform data. We apply this lesson in our `aBeacon+` where we use merchant phones as our *virtual devices* (instead of deploying physical devices) to broadcast their ID so that the couriers can receive them in proximity. In `aBeacon+`, embedded in merchants' smartphone APPs, the virtual device broadcasting module has different versions, whose parameters were set differently for different merchants.

4.5 Performance Metric 2: Reliability P^i_{Reli}

We quantify the reliability P^i_{Reli} of an `aBeacon` device i with a percentage indicating among all the orders from a merchant deployed with the device i, how many orders we detect. There are three major factors impacting P^i_{Reli}: Stay Duration, Device Deployment, and Smartphone Hardware.

Impact of Stay Duration on P^i_{Reli}. The stay duration is the time between a courier arrives at and departs from a merchant. The stay duration varies due to multiple factors such as the layout of a merchant, the courier's walking speed, and whether an order is ready when the courier arrived, i.e., waiting for the order or not. In Fig.6, we found that the longer that a courier stays, the higher the P^i_{Reli}. The stay duration is computed as the differences of departure and arrival time from the couriers' order report data in Table 4(c) Even though there were inaccurate report data due to human errors, our results are based on 76 million orders for two years, ensuring our results are statistically significant. Two observations can be made from Fig.6: (1) The reliability increases with the staying duration, but does not change much after 7 mins; (2) iOS has a much better performance than Android.

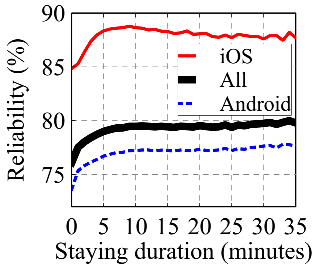


Fig 6: Impact of Stay Duration

Impact of Deployment Position on P^i_{Reli}. The deployment position is an essential factor for reliability, as we found some merchants with an exceptionally low detection ratio. Although our deployment handbook suggested that "Beacons should be attached around the order pickup area", some business managers put devices somewhere else due to various reasons. For example, some merchants do not have a fixed "meal/groceries pickup area"; some merchants prefer the device to be placed somewhere else, e.g., under the counter. We performed some field study, and our findings can be clearly explained with Fig.7 that depicts the layout of a real-world restaurant. In this merchant, there are two entrances with a horizontal wall in the center. Two couriers may pick up orders from both entrances, which leads to the different indoor pickup traces. Unfortunately, because the wall obstructs the device broadcast, only the Courier 1's arrival was detected, which results in a reliability P^i_{Reli} of 46% in our observed period. If the aBeacon device were placed in the pickup area, we could have better reliability since both courier traces can be detected. In short, the impact of deployment position is difficult to estimate due to the uncontrollable deployment quality. The reason is we utilize our in-house business team with no deployment expertise (or incentive), and a deployed device can be moved as well, both of which typically leads to low reliability at some merchants.

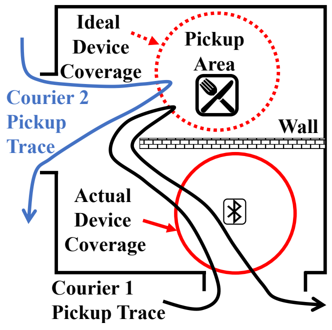


Fig 7: A Field Study of Deployment Position Impact

Impact of Phones Brands and OS on P^i_{Reli}. Our goal is to have most courier smartphones (if not all) to be compatible with aBeacon at both the hardware (i.e., phone brands and models) and software level (i.e., OS types). Given more than 109,000 couriers in Shanghai, it is challenging to either force the couriers to use specific smartphone brands or know if a courier uses an un-supported smartphone. To analyze the impact of smartphone OS, we divide all the orders in aBeacon merchants into two dimensions: whether its courier was detected by aBeacon or not; whether its courier was using an Android or iOS phone. As in Table 5, 63.4% of the orders

Table 5: Detected Ratio of Device OSs over All the Orders

Devices	Detected	Undetected
iOS	13.4%	2.4%
Android	63.4%	20.8%

were detected with the Android couriers (including 52 brands and 672 models), and their average P^i_{Reli} is $\frac{63.4\%}{63.4\%+20.8\%} = 75.2\%$; 13.4% of the orders were detected with the iOS couriers, and their average P^i_{Reli} is $\frac{13.4\%}{13.4\%+2.4\%} = 84.8\%$. We found iOS performs significantly better than Android. For different phone brands (different hardware), the average P^i_{Reli} varies. We show the average P^i_{Reli} of nine well-known brands in China in Fig.8, in which Nexus has the highest P^i_{Reli} of 92%, and iPhones has a P^i_{Reli} of 84%.

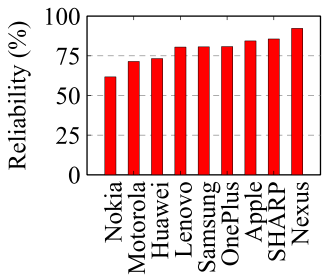


Fig 8: Impact of Smartphone Brand on Reliability

Lesson Learned 4: **Reliability in the Wild.** Many existing wireless sensing systems (e.g., proximity [36], gesture [58], breath [55], human-object interaction [22], and indoor pathway mapping [46]) are mainly tested in the environments with little uncertainty, so they have high reliability. However, we found that even the reliability of a simple presence detection (i.e., courier arrival) is far from guaranteed in a wild, and it is affected by many real-world factors including smartphone software& hardware combination (e.g., 52 phone brands and 672 phone models in Table 5 and Fig.8), and installation position (e.g., low-cost yet unprofessional installation and obstacles in Fig.7), and stay duration (e.g., no waiting time for couriers in Fig.6). In retrospect, we could add an OTA function to some of our devices (but not all devices) deployed in uncertain environments and update them with couriers' phones, e.g., increasing transmission power.

4.6 Performance Metric 3 Utility: P^i_{Util}

The overdue rate reduction is the metric we use to measure the utility P^i_{Util} of deploying an aBeacon device *i* at a particular merchant. For an overdue delivery order (e.g., longer than 30 mins for food), there is an overdue penalty C_{Over} with which the platform will compensate the customer. A typical C_{Over} is \$1, but an overdue penalty could be as high as 200% of the average profit per order if a customer brought delivery insurance. Specifically, the overdue rate is the percentage of the overdue orders among the total orders. So the overdue rate reduction is the difference between the overdue rates before and after an `aBeacon` device deployment. We note that other factors impact the overdue rates of a merchant, e.g., holidays and weathers, but they are out the scope of our paper. We use six months of data before `aBeacon` deployment and 24 months of data after `aBeacon` deployment in the evaluation. There are many features of a merchant that decide the utility of deploying an `aBeacon` device. We study two of them, i.e., Building Floor and City District, due to the space limitation.

Impact of Different Building Floors On P^i_{Util}. To evaluate the impact of deploying an `aBeacon` device on different floors on utility, we aggregate the overdue rate reduction on different floors and compare them with the average overdue rate of all merchants in Shanghai city before and after our `aBeacon` is deployed. The device scale distribution on different floors is given in Fig.9. As shown in Fig.10, the utility is higher

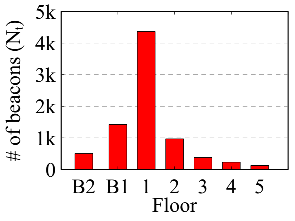


Fig 9: Device Distribution in Different Floors

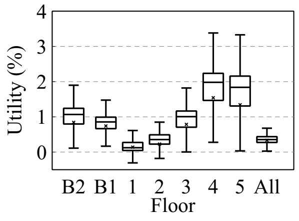


Fig 10: P_{Util} after Deployment in Different Floors

on higher floors or lower basements than the ground floor. This is because the stability of the courier indoor mobility is disproportional to the distance after they enter a building. The higher the floor, the longer the distance, the less stable of courier mobility behaviors (e.g., arrival), the higher benefit for `aBeacon` to detect these behaviors for later order scheduling.

Impact of Different City Districts on P^i_{Util}. To evaluate the impact of districts, we choose five typical districts in Shanghai and compare their average utility, i.e., the overdue rate reduction after `aBeacon` was deployed. As shown in Fig.11, Huangpu is a central business district with a population density of $32,004/km^2$, about three times of New York City ($10,194/km^2$). Songjiang is a suburban area with a population density of $2,892/km^2$, comparable to Los Angeles ($2,910/km^2$). As shown in Fig.12, P^i_{Util} for all merchants with `aBeacon` devices in Songjiang is lower than Shanghai city average; whereas P^i_{Util} in Huangpu is much higher than the average. This is because (1) there are more orders in a more populous area such as Huangpu where each device can serve more orders (we omit the results due to space limitation); (2) the overdue rate is more severe in the city center, and the `aBeacon` can detect couriers more effectively, which leads to better scheduling and thus higher overdue rate improvement.

$\boxed{\textbf{Lesson Learned 5:}}$ **Utility in the Wild.** Unlike other wireless infrastructures, e.g., Wi-Fi, we found the deployment locations with more courier interactions (i.e., demand) during

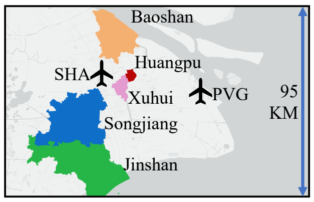


Fig 11: Five Districts

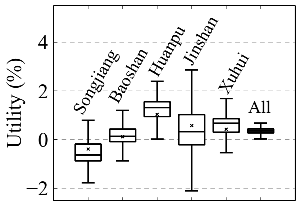


Fig 12: Districts' Utility P_{Util}

our system operation may not have higher utility (quantified by delivery overdue improvement). Instead, we found that the utility of an `aBeacon` device is proportional to the uncertainty of the courier behaviors it can detect (e.g., couriers in higher floors and basements or the downtown area as shown in Fig.9 and Fig.11) because detecting couriers in these uncertain environments can improve the later order scheduling, thus higher utility. This is different from Wi-Fi or cellular device deployment, which are mostly focused on the high user density area. In short, our above findings can provide **practical design guidance** for battery capacity, transmission frequency and power, OTA interface, better installation, and deployment strategies for future wireless systems in the wild.

4.7 Correlation between Different Metrics

Due to the space limitation, we briefly report the results of the correlation between different performance metrics. Our main finding is that for the same `aBeacon` device, when its reliability is low, usually its utility and lifetime are below average; whereas when its reliability is high, the utility is more impacted by the merchant's floors and districts. When reliability $P^i_{\text{Reli}} < 0.5$, this correlation might be caused by improper deployment, which (i) weakens the device utility due to limited data gathered for order scheduling, and (ii) reduces the device lifetime due to potential damage from improper deployment. It also implies that we need to consider other factors if we want to have a longer lifetime and push the utility to the limit when P^i_{Reli} is already high. In our analyses, lifetime and utility are not strongly correlated.

5 Discussions

5.1 Limitations of `aBeacon`

Manual Deployment: For a real-world deployment in the wild, hiring professional teams ensure reliable deployment results but introduces a higher deployment cost per device. In our project, we use our in-house business team to deploy and install `aBeacon` devices at more than 12,000 merchants. Our `aBeacon` system works well in general under this deployment strategy. We admit that our approach may not apply to other settings where such an in-house team is not available. However, we believe our approach can be implemented by Crowdsourcing [42] to deploy wireless devices with a lower labor cost, given little configuration is needed.

No Precise Locations: Another critical limitation of `aBeacon` is that it can only detect couriers' arrival at merchants but cannot perform localization. Given our design goal, a fine-grained localization would be a nice feature to have, but it may not provide higher system gain. Localization increases both hardware cost and deployment cost significantly (it may need onsite configuration or fingerprinting [19]), and may not reduce the overdue rates substantially since the order scheduling only needs coarse-grained locations of couriers.

5.2 System Security

In addition to the hardware cost reduction, another significant improvement in our customization stage is that we enable our devices' security functions. In traditional iBeacon protocol [25], a device ID tuple is fixed for each hardware and broadcast in clear words. It reduces the system complexity while making a device sniffer possible. For example, (1) malicious attackers or unauthorized users (free-riders) can easily restore the device map through war-driving around the devices; (2) if they replicate some device IDs somewhere else, wrong detection information will be collected by `aBeacon`. To address this problem, we designed and implemented a Time-based One-Time Password (TOTP) [53] algorithm to encrypt the device ID broadcast by changing the *major* and *minor* in the ID tuple periodically. A shorter period makes the mapping harder to be restored, whereas a longer period reduces the complexity and the server workload. We set a daily periodical change after exploring the trade-off. The mapping of the device IDs and the merchant locations was stored on the server so that only authorized users can access it. A detailed study of system security is out the scope of this work and will be reported in future work.

5.3 Courier Survey

Feedback is collected from couriers every month regarding multiple aspects, e.g., APP performance, order scheduling, employee care, and penalty appeal. Among the 433 negative feedback on "APP performance" in a recent month, we found the following feedback potentially related to `aBeacon` (# of reports): inaccurate localization (23), slow localization (14), cannot report arrival at the merchants (11), too much battery consumption (8), too much data consumption (2), mandatory Bluetooth on (2). The top three criticisms are all about localization. The underlying reason is that the couriers must report "arrival" at the merchants and customers, and the report must be conducted near (e.g. within 500m) the merchants or customers based on the courier's GPS and the latitude/longitude of the merchant or customer. GPS drifting due to the indoor environment is the main reason for failed reports. The feedback results indicate that alternatives are need besides GPS. `aBeacon` can help in some cases, but we still need to fix the cases when GPS and `aBeacon` fail at the same time. We should also take care of the battery and data consumption.

5.4 Ethics and Privacy

All the data sets are collected under the consent of the couriers. In all our analyses, we have been working on aggregate data. As a result, our results cannot be used to trace back to individuals. The courier ID is an anonymous key to join different data sets, and any other ID information cannot be tracked or identified in practice. We did not utilize personal information from the couriers, e.g., age, gender, income, to protect the couriers' privacy.

5.5 Additional Applications of `aBeacon`

In addition to the direct system gain we measure in this paper, Alibaba has been using `aBeacon` data for a few additional applications based on courier arrival detection.

Order Delivery Time Estimation: The Estimated Time of Arrival (ETA) problem is one of the critical issues in the delivery industry, especially hard for the indoor environment. Based on `aBeacon` data, we obtain travel time between different indoor merchants and build a data-driven model for delivery time prediction, which has been used by other Alibaba teams to predict the overdue rate for the order scheduling.

Merchant Location Correction: Accurate merchant locations are essential in the delivery service. Currently, these locations are provided by merchants themselves and consist of unintentional or intentional errors. Based on the `aBeacon` data, we can measure the travel time between different merchants, cross-validate the accuracy of these locations, and potentially correct them based on massive traveling data.

Anomaly Detection: Unlike GPS data that can be faked on the smartphone [37], `aBeacon` data provide a physical presence confirmation. `aBeacon` data have been used to detect cheating in the delivery process, e.g., frauds conducted by merchants and couriers for the platform subsidy. A detailed courier behavior study measured by manually reported data and automatically collected `aBeacon` data is out of this paper's scope and merits additional investigation.

5.6 `aBeacon+`: Next Generation of `aBeacon`

Since it is expected the maximum lifetime of `aBeacon` is two years, we have been working on a new system called `aBeacon+` built upon `aBeacon` to retain its strengths and address its limitations. In `aBeacon+`, under the merchants' consent, we use merchants' smartphones as `aBeacon` devices instead of deploying `aBeacon` devices, to avoid the hardware and deployment cost. `aBeacon+` does not suffer from vandalism, hardware malfunction, and merchant closures. We embed a broadcasting module in the official merchant APP based on the opportunity that almost every merchant owner needs to install a merchant APP to manage orders. The deployment and operation insights we obtained from `aBeacon` have guided our development of `aBeacon+`, e.g., batch-based deployment and merchant targeting (see our Lessons Learned for details).

We acknowledge the incentives and privacy issues need our attention to make `aBeacon+` practical and salable. However, we argue that the APP users may be willing to provide their locations voluntarily with appropriate incentives in some settings. In our case, a merchant provides this virtual `aBeacon` service on its smartphone. The virtual aBeacon can help the platform decide whether an overdue order is because of the merchant's long order preparation time or the courier's late pickup. Similar applications have been launched in Singapore and potentially in the US for public health purposes during the recent COVID-19 pandemic. TraceTogether [35], a BLE based APP developed in Singapore operates similarly to an aBeacon+ scheme that users nearby can detect each other for contact tracing purpose in response to COVID-19, which is another example of smartphone users' voluntary participation under some practical incentive.

5.7 Implications on Others Systems

Our study offers some interesting implications for current and future networked systems' design, verification, and operation.

Offline Ground Truth Collection for the Verification of Wi-Fi based Solutions: Along 48% of our merchants have stable Wi-Fi access, aBeacon can be used to collect offline ground truth for various applications based on Wi-Fi in the wild to verify existing assumptions or models on Wi-Fi systems and contribute to the community.

Deployment Strategies for 5G and Edge Computing: It has been widely accepted that the extreme densities of base stations and devices are needed to support 5G applications due to its high carrier frequencies with massive bandwidths [3, 8]. Edge computing networks also have a similar setting. Although these systems may need professional teams for the deployment since their devices typically require configuration, our five lessons learned on quantifying system gain, scale evolution, and performance metrics (e.g., lifetime, reliability, and utility) may reduce their indoor operation efforts.

6 Related Works

Table 6: Operational BLE Device Systems

Nation	Deployment Site	Application	Scale
Iceland	Eldheimar museum [34]	Localization	54 devices
U.S.	Beale Street [48]	Presence detection	100 devices
U.K.	Gatwick airport [21]	Localization	2,000 devices
India	Railway station [18]	Presence detection	2,000 devices
Brazil	Tom Jobim airport [4]	Localization	3,000 devices

Operational BLE Device System: To our knowledge, as we are proposing one of the largest BLE device systems in the world, it is worthwhile to give a summary of existing operational BLE device systems. As shown in Table 6, most BLE systems are operated in public sites such as airports or museums for presence detection or indoor localization. The largest BLE system we found is deployed in Tom Jobim airport in Brazil with 3,000 devices, which is fewer than the 12,109 devices in aBeacon we deployed in Shanghai, China. More importantly, these existing systems are operated in a controlled environment (e.g., airports, museums, train stations), but our operating environment is in the wild and out of control. It enables our system to provide some new insights from our lessons learned from large-scale system lifetime, reliability, utility, and cost.

BLE Device Studies: Existing BLE system studies can be categorized according to their applications: localization or presence detection. Indoor localization with BLE systems is similar to works done with Wi-Fi. Fingerprinting is studied in [19] to achieve the accuracy of < 4.8 m at the density of one device per 100 m^2, compared with < 8.5 m for Wi-Fi. Map matching is used in [56] to estimate a user's route based on devices with known locations. 1,600 BLE devices are deployed in all the classrooms and corridors of an institute for evaluation. Dynamic RSSI propagation modeling is proposed in [12] to achieve fine-grained (< 2 m) localization and tracking. There are also studies exploring the proximity information provided. Dining hall usage and student check-in are studied in [38] and [24] with BLE device proximity information. Hardware modifications such as energy harvester are also studied in [28] for better performance.

Real-world Sensing Systems: Another related topic is the large-scale real-world sensing system. These studies lay more emphasis on the system implementation and operation for practical problems. LiveTag is proposed in [22] to sense human-object interaction passively. [51] attempts to answer why RFID sensing systems remain research prototypes and have not been widely deployed in practice with theoretical analysis and real-world experiments.

7 Conclusion

This paper introduces aBeacon, a wireless indoor BLE device system in Alibaba, from its conception to its retirement by a unique operation study in Shanghai. We quantify aBeacon's performance by scale, lifetime, reliability, and utility, for all of which we provide some new insights obtained in our 3-year system operation in the wild. In particular, we built aBeacon from the ground up in a metric-based approach consisting of two phases, i.e., an 8-month pilot study and a 28-month deployment and operation in the wild, including devices in 12,109 merchants and interactions with 109,378 couriers. From the long-term city-wide study, we identify five key observations and lessons regarding system gain quantification, system scale evolution, lifetime, reliability, and utility in the wild. We believe these in-depth lessons learned have implications for other systems requiring long-term operations and broad geospatial coverage such as 5G and Edge Computing.

8 Acknowledgments

We sincerely thank reviewers for their insightful feedback. This work is supported in part by Alibaba Group, Local Service Company, National Key R&D Program of China 2018YFB2100300, 2018YFB0803400, and National Natural Science Foundation of China (NSFC) 61925202, 61772046. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the funding agencies or others.

References

[1] Alibaba Group, Local Service BU. abeacon data-set link. https://tianchi.aliyun.com/dataset/dataDetail?dataId=76359, 2020.

[2] Fadel Adib, Zachary Kabelac, and Dina Katabi. Multi-person localization via rf body reflections. In *NSDI'15 Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, pages 279–292, 2015.

[3] Mamta Agiwal, Abhishek Roy, and Navrati Saxena. Next generation 5g wireless networks: A comprehensive survey. *IEEE Communications Surveys & Tutorials*, 18(3):1617–1655, 2016.

[4] Airport Benchmarking. Airlines and airports are beaconizing. Webpage, 2016.

[5] Alibaba Group. Alibaba group. Webpage, 2020.

[6] Amazon. Amzon prime now. Webpage, 2020.

[7] Ganesh Ananthanarayanan and Ion Stoica. Blue-fi: enhancing wi-fi performance using bluetooth signals. In *ACM MobiSys*, pages 249–262, 2009.

[8] Jeffrey G Andrews, Stefano Buzzi, Wan Choi, Stephen V Hanly, Angel Lozano, Anthony CK Soong, and Jianzhong Charlie Zhang. What will 5g be? *IEEE Journal on selected areas in communications*, 32(6):1065–1082, 2014.

[9] Niranjan Balasubramanian, Aruna Balasubramanian, and Arun Venkataramani. Energy consumption in mobile phones: a measurement study and implications for network applications. In *ACM SIGCOMM*, pages 280–293, 2009.

[10] Bright Beacon. Indoor lbs, 2020.

[11] Vladimir Brik, Suman Banerjee, Marco Gruteser, and Sangho Oh. Wireless device identification with radiometric signatures. In *ACM MobiCom*, pages 116–127, 2008.

[12] Dongyao Chen, Kang G Shin, Yurong Jiang, and Kyu-Han Kim. Locating and tracking ble beacons with smartphones. In *ACM CoNEXT*, pages 263–275, 2017.

[13] Krishna Chintalapudi, Anand Padmanabha Iyer, and Venkata N Padmanabhan. Indoor localization without the pain. In *ACM MobiCom*, pages 173–184, 2010.

[14] Deliveroo. Deliveroo. Webpage, 2020.

[15] Patrick Dickinson, Gregorz Cielniak, Olivier Szymanezyk, and Mike Mannion. Indoor positioning of shoppers using a network of bluetooth low energy beacons. In *2016 International Conference on Indoor Positioning and Indoor Navigation (IPIN)*, pages 1–8. IEEE, 2016.

[16] DoorDash. Doordash. Webpage, 2020.

[17] Eleme. Eleme. Webpage, 2020.

[18] Fablian Technologies. Eddystone beacon installation at indian railway stations by google. Webpage, 2018.

[19] Ramsey Faragher and Robert Harle. Location fingerprinting with bluetooth low energy beacons. *IEEE journal on Selected Areas in Communications*, 33(11):2418–2428, 2015.

[20] Fast Casual. Everything's bigger in ... china? Webpage, 2017.

[21] Future Travel Experience. Gatwick's beacon installation provides partners with blue dot navigation and augmented reality wayfinding. , 2017.

[22] Chuhan Gao, Yilong Li, and Xinyu Zhang. Livetag: Sensing human-object interaction through passive chipless wifi tags. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 533–546, 2018.

[23] Andreas Haeberlen, Eliot Flannery, Andrew M Ladd, Algis Rudys, Dan S Wallach, and Lydia E Kavraki. Practical robust localization over large-scale 802.11 wireless networks. In *ACM MobiCom*, pages 70–84, 2004.

[24] Yun Huang, Qunfang Wu, and Yaxing Yao. Bluetooth low energy (ble) beacons alone didn't work! In *ACM UbiComp*, pages 62–65, 2018.

[25] Apple Inc. ibeacon. Webpage, 2020.

[26] InstaCart. Instacart. Webpage, 2020.

[27] Texas Instruments. Cc2540 datasheet [eb/ol](2013-6-24).

[28] Kang Eun Jeon, James She, Jason Xue, Sang-Ha Kim, and Soochang Park. luxbeacon-a batteryless beacon for green iot: Design, modeling, and field tests. *IEEE Internet of Things Journal*, 6(3):5001–5012, 2019.

[29] Junghyun Jun, Yu Gu, Long Cheng, Banghui Lu, Jun Sun, Ting Zhu, and Jianwei Niu. Social-loc: Improving indoor localization with social sensing. In *ACM SenSys*, pages 1–14, 2013.

[30] Bryce Kellogg, Vamsi Talla, Shyamnath Gollakota, and Joshua R. Smith. Passive wi-fi: Bringing low power to wi-fi transmissions. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016.

[31] Swarun Kumar, Stephanie Gil, Dina Katabi, and Daniela Rus. Accurate indoor localization with zero start-up cost. In *ACM MobiCom*, pages 483–494, 2014.

[32] Ye-Sheng Kuo, Pat Pannuto, Ko-Jen Hsiao, and Prabal Dutta. Luxapose: Indoor positioning with mobile phones and visible light. In *ACM MobiCom*, pages 447–458, 2014.

[33] Jiayang Liu and Lin Zhong. Micro power management of active 802.11 interfaces. In *ACM MobiSys*, pages 146–159, 2008.

[34] Locatify. Eldheimar museum, intuitively designed beacon based museum audio guide, 2020.

[35] Government of Singapore. Help speed up contact tracing with tracetogether, 2020.

[36] Timothy J Pierson, Travis Peters, Ronald Peterson, and David Kotz. Proximity detection with single-antenna iot devices. In *ACM MobiCom*, pages 1–15, 2019.

[37] Google Play. Fake gps location, 2020.

[38] Rachael Purta and Aaron Striegel. Estimating dining hall usage using bluetooth low energy beacons. In *ACM UbiComp*, pages 518–523, 2017.

[39] Kun Qian, Chenshu Wu, Yi Zhang, Guidong Zhang, Zheng Yang, and Yunhao Liu. Widar2. 0: Passive human tracking with a single wi-fi link. In *ACM MobiSys*, pages 350–361, 2018.

[40] Qualcomm. Csr1010 datasheet. Webpage, 2015.

[41] Ahmad Rahmati and Lin Zhong. Context-for-wireless: context-sensitive energy-efficient wireless data transfer. In *ACM MobiSys*, pages 165–178, 2007.

[42] Sanae Rosen, Sung-ju Lee, Jeongkeun Lee, Paul Congdon, Z Morley Mao, and Ken Burden. Mcnet: Crowdsourcing wireless performance measurements through the eyes of mobile devices. *IEEE Communications Magazine*, 52(10):86–91, 2014.

[43] Dialog Semiconductor. Da14580 low power bluetooth smart soc. *DataSheet], https://support. dialog-semiconductor. com/downloads/DA14580_DS _v3*, 1, 2015.

[44] Nordic Semiconductor. nrf51822. *Online at http://www. nordicsemi. com/eng/Products/Bluetooth-R-low-energy/nRF51822, visited Nov*, 2013.

[45] Nordic Semiconductor. Bluetooth 5, 2020.

[46] Guobin Shen, Zhuo Chen, Peichao Zhang, Thomas Moscibroda, and Yongguang Zhang. Walkie-markie: Indoor pathway mapping made easy. In *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pages 85–98, 2013.

[47] TechCrunch. Gatwick airport now has 2000 beacons for indoor navigation, 2017.

[48] THINKPROXI. Thinkproxi announces famous beale street implemented beacon technology. Webpage, 2017.

[49] Zhao Tian, Yu-Lin Wei, Wei-Nin Chang, Xi Xiong, Changxi Zheng, Hsin-Mu Tsai, Kate Ching-Ju Lin, and Xia Zhou. Augmenting indoor inertial tracking with polarized light. In *ACM MobiSys*, pages 362–375, 2018.

[50] Uber. Uber eat. Webpage, 2020.

[51] Wang, Ju, Liqiong Chang, Omid Abari, and Srinivasan Keshav. Are rfid sensing systems ready for the real world? In *ACM MobiSys*, pages 366–377, 2019.

[52] Yu-Lin Wei, Chang-Jung Huang, Hsin-Mu Tsai, and Kate Ching-Ju Lin. Celli: Indoor positioning using polarized sweeping light beams. In *ACM MobiSys*, pages 136–147, 2017.

[53] Wikipedia. Time-based one-time password. , 2020.

[54] Bo Xie, Guang Tan, and Tian He. Spinlight: A high accuracy and robust light positioning system for indoor applications. In *ACM SenSys*, pages 211–223, 2015.

[55] Xiangyu Xu, Jiadi Yu, Yingying Chen, Yanmin Zhu, Linghe Kong, and Minglu Li. Breathlistener: Fine-grained breathing monitoring in driving environments utilizing acoustic signals. In *ACM MobiSys*, pages 54–66, 2019.

[56] Daisuke Yamamoto, Ryosuke Tanaka, Shinsuke Kajioka, Hiroshi Matsuo, and Naohisa Takahashi. Global map matching using ble beacons for indoor route and stay estimation. In *ACM SIGSPATIAL*, pages 309–318, 2018.

[57] Yu Yang, Yi Ding, Dengpan Yuan, Guang Wang, Xiaoyang Xie, Yunhuai Liu, Tian He, and Desheng Zhang. Transparent indoor localization with uncertain human participation for instant delivery. In *ACM MobiCom*, pages 116–127, 2020.

[58] Yue Zheng, Yi Zhang, Kun Qian, Guidong Zhang, Yunhao Liu, Chenshu Wu, and Zheng Yang. Zero-effort cross-domain gesture recognition with wi-fi. In *ACM MobiSys*, pages 313–325, 2019.

EarFisher: Detecting Wireless Eavesdroppers by Stimulating and Sensing Memory EMR

Cheng Shen[1], Jun Huang[2]*
[1]*Peking University,* [2]*Massachusetts Institute of Technology*

Abstract

Eavesdropping is a fundamental threat to the security and privacy of wireless networks. This paper presents EarFisher – the first system that can detect wireless eavesdroppers and differentiate them from legitimate receivers. EarFisher achieves this by stimulating wireless eavesdroppers using bait network traffic, and then capturing eavesdroppers' responses by sensing and analyzing their memory EMRs. Extensive experiments show that EarFisher accurately detects wireless eavesdroppers even under poor signal conditions, and is resilient to the interference of system memory workloads, high volumes of normal network traffic, and the memory EMRs emitted by coexisting devices. We then further propose a method to detect eavesdropper's countermeasure, which deliberately emits strong memory EMR to interfere with EarFisher's detection.

1 Introduction

Rendered by the broadcast characteristic of wireless medium, eavesdropping has been a fundamental threat to the security and privacy of wireless networks ever since the invention of wireless communication. While significant cryptographic research has been devoted to tackling this threat, in this paper, we take an orthogonal angle to explore the feasibility of wireless eavesdropper detection. A security primitive capable of this task is essential in a wide range of scenarios. First, in wireless networks serving public areas (e.g., airports, campus, malls, etc), Layer-2 encryption is commonly disabled to facilitate open access. Second, encryption algorithms are subject to extensive side channel analysis [7, 8, 11, 15, 16, 21, 22], which allow attackers to decipher encryption keys. Third, cryptographic protocols themselves often suffer fatal flaws that are difficult to identify before universal adoption. For example, in 2017, researchers uncovered that the four-way handshake of WPA2 is vulnerable to the key re-installation attack [35], which allows eavesdroppers to compromise encryption keychains. The flaw had been present since the release of 802.11i in 2004, leaving billions of Wi-Fi users potentially exposed to eavesdropping for more than 13 years.

*Corresponding author: junhuang@mit.edu.

Beyond complementing encryption schemes, an eavesdropper detector can be an essential building block of a secure network. For example, in quantum networks, legitimate receivers can detect eavesdroppers by leveraging the quantum physic law, where the state of a quantum bit 'collapses' whenever it is intercepted. Quantum key distribution protocols [9] use this law to verify the confidentiality of encryption keys, which leads to fundamentally assured communication security in quantum networks.

Unfortunately, to date, there has been no effective method to detect eavesdroppers in wireless networks. Unlike quantum eavesdroppers, wireless eavesdroppers can be completely passive without actively transmitting or altering signals in the air. Recent studies exploit the RF leakages of radio circuits to detect wireless receivers [13, 24, 26, 27, 32, 34, 36]. However, such leakages are extremely weak, limiting detection range to only a few feet. More importantly, because all wireless radios emit leakages during signal reception, this method cannot differentiate eavesdroppers from legitimate receivers.

In this paper, we present EarFisher – the first system that can detect wireless eavesdroppers and differentiate them from legitimate receivers. The key idea is based on the observation that, unlike legitimate receivers who drop others' packets in network interface cards (NICs), only eavesdroppers digest all packets in their CPU-memory systems. Inspired by this observation, EarFisher stimulates eavesdroppers by transmitting a flow of *bait* packets forged with a virtual receiver address, and then detects eavesdroppers by sensing the surge of their electromagnetic radiations (EMRs) when they write baits into their memories. Recent studies show that the multi-channel architecture of modern memories amplifies memory EMR [17], which helps EarFisher extend detection range.

To realize this idea, we tackle four key challenges. First, when multiple devices having the same memory frequency coexist in an environment, their memory EMRs mix in frequency spectrum, making it difficult to accurately sense them separately. Second, memory workloads of operating systems and applications also produce memory EMRs, which are difficult to distinguish from the eavesdropper's responses when the memory activities coincidentally occur at the same time of bait packet transmissions. Third, despite the amplification of multi-channel architecture, memory EMR is still very weak, requiring a long time of stimulus to trigger a sufficiently strong response at the eavesdropper. However, intensive bait packet transmissions in a large time window can block the wireless channel, and may present a distinguishable traffic pattern that can be noticed by crafty eavesdroppers. Fourth, eavesdroppers knowing the design and presence of EarFisher may deliberately write its memory, which will produce strong memory EMR that interferes with EarFisher's detection.

To address these challenges, EarFisher employs new signal processing algorithms to sense and separate the memory EMRs of different devices, obscures and disguises stimulus traffic as short bursts of normal packets, incorporates statistical tool to tolerate the interfering EMRs produced by system memory workloads, and exploits a fundamental dilemma of eavesdroppers to detect deliberate writing-based countermeasure. Extensive experiments show that EarFisher accurately detects eavesdroppers even under poor signal conditions, and is resilient to the interference of system memory workloads, high volumes of normal network traffic, and the memory EMRs of coexisting devices. We then further demonstrate EarFisher's effectiveness in a real testbed, where three EarFisher nodes are deployed to monitor an indoor area of 1600 ft^2. Experiment results show that EarFisher reliably detects eavesdroppers placed at different locations despite the complexity of indoor environments, such as the block of walls.

2 Related Work

Cryptographic attacks. Since the invention of wireless communication, encryption has been the primary security measure to combat eavesdropping. However, extensive research shows that eavesdroppers can compromise encryption based on a variety of side-channel attacks [7, 8, 15, 16, 21, 22]. In particular, Camurati et al. [11] show that the EM leakage of the processor on wireless chips can be inadvertently amplified by RF front-end, allowing an eavesdropper to decipher the encryption key from a distance.

Moreover, cryptographic protocols themselves often suffer fatal flaws that are difficult to identify before universal adoption. For example, Wired Equivalent Privacy (WEP), a security protocol ratified as a part of 802.11 in 1997, was found to have fatal flaws in 2001 [14]. WEP was then superseded by Wi-Fi Protected Access (WPA) in 2004, but history repeated itself. In 2017, researchers demonstrated that the four-way handshake of WPA has a fatal vulnerability, which allows eavesdroppers to compromise the encryption keychain using key reinstallation attacks [35]. From 2004 to 2017, the vulnerability of WPA left billions of Wi-Fi users potentially exposed to eavesdropping for more than 13 years.

Eavesdropper detection. Prior eavesdropper detectors commonly rely on sensing RF signals leaked from the front-end circuit of wireless receiver. This method was first proposed in [27, 36] to detect primary receivers in cognitive radio networks, and then extended to UWB, WiMAX, and MIMO channels to detect hidden radios [24, 26]. Ghostbuster [13] further augments this method using spatial cancellation of interference and frequency cancellation of signal artifacts, which allow it to extract leakages under poor signal conditions in the presence of ongoing wireless transmissions. However, Ghostbuster still suffers limited detection range (less than 1m for COTS Wi-Fi receivers) because of the weak power of leakage from RF circuit.

Recent studies [32, 34] show that superheterodyne and superregenerative receivers can be detected from a longer distance by exploiting the known characteristics of their RF leakages. However, these techniques are highly dependent on the receiver architecture. Unfortunately, superheterodyne and superregenerative receivers are commonly used in low-power wireless remote, but are far less common in mainstream wireless communication systems such as Wi-Fi.

More importantly, because all radios emit RF leakages during signal reception, existing eavesdropper detectors suffer a common limitation as they cannot differentiate eavesdroppers from legitimate receivers. To sidestep this limitation, Ghostbuster assumes a threat model where the number of legitimate receivers is known a priori [13]. This assumption restricts its usability to a very narrow range of specific scenarios.

EM side-channels. Recently, researchers leverage the EM side channels of CPU and memory for attestation [29], memory profiling [30], and malware detection [18, 25, 37]. Different from these passive analysis, EarFisher features a new paradigm that actively stimulates memory EMR. Moreover, EarFisher employs a signal processing algorithm that bases on fine-grained measurement and theoretical characterization of memory clock, which makes it possible to not only extract weak memory EMRs under poor signal conditions, but also separate and track individual memory EMRs when multiple devices having the same memory frequency coexist in a crowded environment.

3 Threat Model

A wireless eavesdropper is a malicious receiver who sniffs on other devices' packets. Typically, eavesdropping can be implemented by modifying the device driver to enable *monitor mode*, in which a wireless chip transfers all received packets to the host. A recent study shows that most drivers on major operating systems (e.g., Linus, Windows, macOS) support monitor mode by default [1].

By eavesdropping on network traffic, the attacker's goal is to gather sensitive data, such as personal and business related information, or secrets necessary to enable decipher and man-in-the-middle attacks. Typically, wireless chips are integrated with a microcontroller and a small RAM of at most a couple of MiBs. The on-chip system is tasked for simple computation such as checking receiver address and verifying packet checksum, but is far from capable of security- and privacy-intrusive processing. As a result, the eavesdropper must digest sniffed packets in the CPU-memory system of the host.

To this end, the wireless chip of the eavesdropper needs to write all sniffed packets to the memory of the host. This is typically under the control of DMA, and will produce EMR when sniffed packets are transferred over memory bus. Specifically, for SDRAMs, memory EMR can be observed as a radio signal around the frequency of memory clock. Because DDR uses double pumping, the clock frequency is a half of memory speed. In practice, the clock frequency of a modern DDR has 13 possible values ranging from 200 MHz of DDR2-400 to 1600 MHz of DDR4-3200. To sense memory EMR, one can scan DDR frequencies one by one, or use a group of sensors to monitor multiple frequencies in parallel. In the rest of this paper, we assume the memory clock frequency of the eavesdropper is known.

4 Characterizing Memory EMR

In this section, we present measurements and model to characterize memory EMR. Our measurements are conducted on two laptops of DDR3-1600 and DDR4-2133, respectively. A BladeRF with an omni-directional 5 dBi antenna is employed as the receiver.

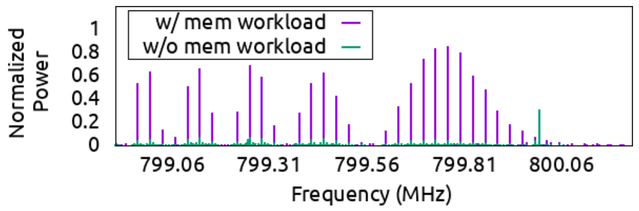
4.1 Spectrum Pattern

Measurement-based characterization. To measure the frequency spectrum of a device's memory EMR, we take FFT over an 1s window of signals captured around the clock frequency of the device's memory. To study how memory workload impacts on memory EMR, we created a process to write memory intensively[1], and then compare the spectrum patterns measured before and after the start of the process.

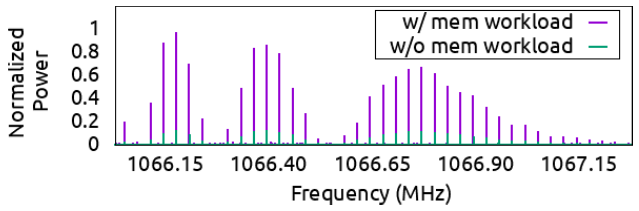
As shown in Fig. 1, we observe that the frequency spectrum of memory EMR features a series of energy peaks distributed over a side-band of about 1 MHz below the frequency of memory clock. In the presence of intensive memory workload, the amplitudes of all energy peaks increase simultaneously and significantly. To further characterize the spectrum pattern, we examine the auto-correlation of peak frequencies and find that the frequency interval between consecutive peaks is constant. Specifically, for the DDR3-1600 and the DDR4-2133, the frequency intervals are 31.8 KHz and 31.2 KHz, respectively.

Theoretical characterization. We then further characterize memory EMR based on the theoretical model of memory

[1]To write memory directly, we used SSE instructions to bypass cache.



(a) DDR3-1600.



(b) DDR4-2133.

Figure 1: The spectrum pattern of memory EMR with and without memory workload.

clock. Because memory clock injects fluctuating current into memory bus, it produces EMR at the clock frequency. As memory reads/writes are performed at clock edges, they amplify EMR amplitude of memory clock but will not affect the pattern of frequency spectrum.

The simplest form of a clock is a sine wave of which the energy is all concentrated at clock frequency. However, this leads to a high EMR intensity that may violate regulatory requirement for electromagnetic compatibility. To walk-around this issue, modern clock generators use spread spectrum techniques to reshape the energy distribution of clock. Denote an unspread clock as,

$$V_{clk}(t) = A\cos(2\pi f_0 t),$$

where f_0 is the frequency of memory clock. A spread spectrum clock is the frequency modulation of $V_{clk}(t)$, which can be expressed as,

$$V_{ssc}(t) = A\cos(2\pi f_0 t + \frac{\Delta f}{f_m}\sin(2\pi f_m t)), \tag{1}$$

where f_m and Δf are the modulation frequency and peak frequency deviation, respectively. By expanding Eqn. 1 using the Jacobi-Anger expansion, the frequency spectrum of $V_{ssc}(t)$ can be expressed as [10],

$$\|\mathcal{F}_{ssc}(f)\| \overset{\text{def}}{=} \|\sum_n J_{\text{n}}(\frac{\Delta f}{f_m})(\delta(f - f_0 + nf_m) \\ -\delta(f - f_0 - nf_m))\|.$$

where $J_{\text{n}}(\cdot)$ is the Bessel function of the first kind, and $\delta(\cdot)$ is the Dirac delta function.

To maintain a stable synchronization between memory and memory controller, $V_{ssc}(t)$ needs to be band-pass filtered before being used as a memory clock. Typically, the band-pass filter removes frequency components higher than memory clock frequency and then imposes a low cutoff to limit the

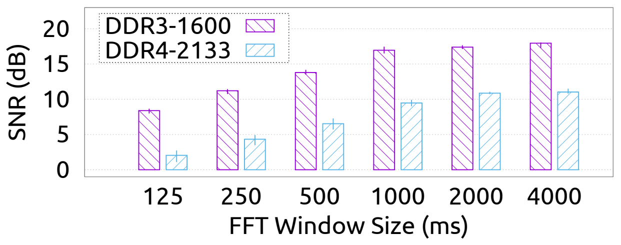


Figure 2: The SNR of received memory EMR as a function of FFT window size.

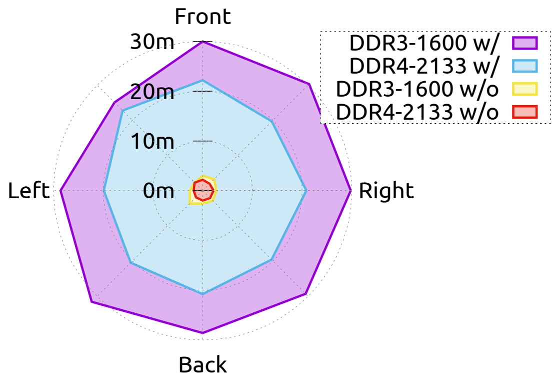


Figure 3: The 3 dB range of memory EMRs with and without memory activities.

maximum frequency deviation. After band-pass filtering, the frequency spectrum of $V_{ssc}(t)$ is transformed to,

$$\|\mathcal{F}_{ssc}(f)\| \overset{\text{def}}{=} \|\mathcal{A}(f)\sum_n J_n(\frac{\Delta f}{f_m})\delta(f - f_0 + nf_m)\|, \quad (2)$$

where $\mathcal{A}(f)$ is the impulse response of the band-pass filter.

Summary of characteristics. Eqn. 2 indicates that the energy of memory EMR is non-zero only at $f_0 - nf_m$, which interpreted the spectrum pattern shown in Fig. 1. In other words, memory EMR is composed of a series of *sub-clock components*, where the frequency interval between consecutive sub-clocks is f_m, and the first sub-clock is at f_0.

4.2 Directivity and Range

We then place receiver at different directions to measure the *3 dB range* of memory EMR, which is defined as the maximum distance from which the SNR of received memory EMR is higher than 3 dB. In this experiment, we calculate SNR of memory EMR using the sub-clock of the highest power.

It is worth noting that the SNR of received memory EMR is dependent on the size of FFT window. As shown in Fig. 2, the larger the FFT window, the higher the SNR. However, the gain yielded by enlarging FFT window gradually diminishes as the size of FFT window increases. To understand why, consider a FFT bin (whose size equals the inversion of FFT window size) that is large enough for containing one sub-clock. In this case, one can always reduce the FFT bin to suppress noise without affecting the sub-clock, thereby increasing SNR. On the other hand, if the size of FFT bin is already smaller than the bandwidth of sub-clock, then further reducing FFT bin will also reduce the energy of contained memory EMR, thus providing no SNR gain. Based on the results shown in Fig. 2, we set FFT window size to 1s in the following measurements.

Fig. 3 plots the 3 dB range of memory EMR measured from different directions. We observe that receiving direction has a slight impact on range, which should be attributed to the shape of shielding cases of particular devices. Moreover, because of the lower operating voltage, the range of DDR4-2133's EMR is about 25% shorter than that of DDR3-1600. Nevertheless, even when receiving from the worst direction, the 3dB ranges of DDR3-1600 and DDR4-2133 exceed 25m and 20m, respectively.

4.3 Response to Stimulus

To understand the impact of stimulus network traffic on eavesdropper's memory EMR, we setup an experiment where the laptop equipped with DDR4-2133 is employed to eavesdrop on an 802.11n transmitter. The experiment is conducted on a clean channel to avoid the interference of uncontrolled network traffic. The 802.11n transmitter is configured to send an 100 ms UDP flow every 200 ms. We then vary the rate of the UDP flow and repeat the experiment.

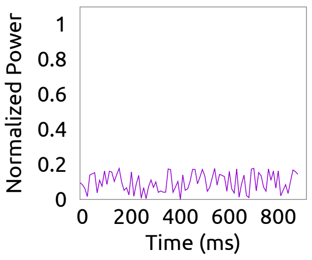
Fig. 4 shows the time varying amplitude of memory EMR measured in close proximity to the eavesdropper using a sliding FFT window of 100 ms. As shown in Fig. 4, the eavesdropper's memory EMR demonstrates a clear responsive pattern when the rate of UDP flow increases to only 2 Mbps. The amplitude of response can be significantly boosted by further increasing the rate of stimulus network traffic.

5 EarFisher Overview

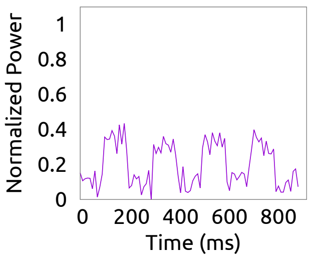
EarFisher is designed as a standalone system to detect eavesdroppers in a wireless network without the cooperation of other network nodes. It differentiates eavesdroppers from legitimate receivers based on an architectural criteria, where receivers are convicted of eavesdropping as long as they transfer other devices' packets to memory. In contrast, a legitimate receivers should drop other devices' packets immediately in wireless NICs.

As illustrated in Fig. 5, EarFisher consists of a stimulator and a detector. The stimulator is a wireless network of two cooperative nodes, which exchange packets to generate stimuli[2]. The detector senses memory EMR using a software defined radio, which is hosted by one of the stimulator nodes and is synchronized with the wireless NIC to monitor the variations of memory EMRs under traffic stimuli.

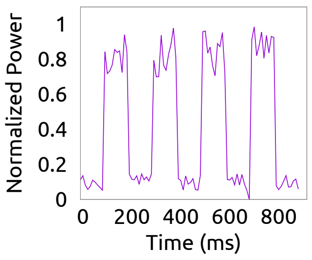
[2]A simpler method is to use a single wireless device to forge a packet flow. However, crafty eavesdroppers may detect the forged packet flow by noticing that the sender and the receiver manifest the same channel state information.



(a) Rate = 600 Kbps.



(b) Rate = 2 Mbps.



(c) Rate = 20 Mbps.

Figure 4: Time varying amplitude of memory EMR under the stimuli of periodic UDP flows.

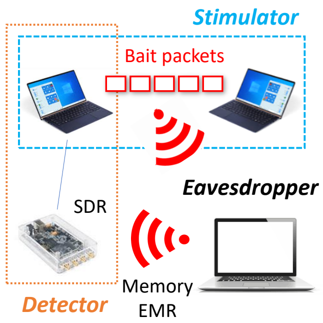


Figure 5: EarFisher architecture.

At a high-level, EarFisher features four key designs.

Sensing memory EMRs. Despite the amplification by multichannel architecture, memory EMR is still very weak and thus can be easily buried by noise when sensing from a distance. Moreover, when devices having the same memory frequency coexist in an environment, their memory EMRs will be mixed together in frequency spectrum, making it difficult to track them separately. EarFisher addresses these challenges by using folding [39] – a signal processing algorithm originally used by large radio telescopes to amplify astronomical signals – to sense weak memory EMRs. It then leverages the fingerprint of memory clock to separate and track the memory EMRs of different devices.

Obscuring stimulus traffic. Sensing weak memory EMRs requires a large FFT window to suppress noise. However, intensive transmission of stimulus traffic in a large time window may interfere with normal network traffic. In addition, it may introduce a distinguishable traffic pattern to alert crafty eavesdroppers. To address this challenge, EarFisher first splits a large block of bait packets into small pieces and then disguises them as normal network traffic. At the detector side, EarFisher stitches signal samples captured at the time instants of stimuli into a complete window before taking FFT.

Tolerating system memory workloads. Memory workloads of operating systems and applications also produce memory EMR, which is difficult to distinguish from the response of eavesdropper when memory activities coincidentally occur at the time instants of traffic stimuli. To avoid false alarm, EarFisher profiles the memory EMR incurred by system memory workloads for each device at runtime. It then tests the hypothesis if the surge of memory EMR under stimuli is sufficiently significant to claim a detection of response.

Detecting countermeasure. Eavesdroppers knowing the design and presence of EarFisher may actively write memory to emit strong EMR, which will interfere with EarFisher's detection. To detect this countermeasure, EarFisher exploits a fundamental dilemma faced by the eavesdropper, where intermittent writing of memory leaves significant chance of exposing the response to stimuli, while consistent writing presents an abnormal EMR pattern that can be distinguished from normal system memory workloads.

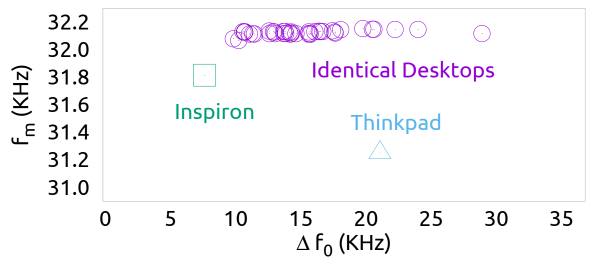


Figure 6: The distribution of memory clock fingerprints.

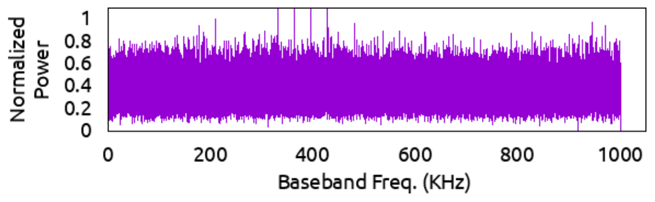
6 System Design

This section presents the design of EarFisher in detail. We first propose a sensing primitive to monitor memory EMRs, and then describe the design of stimulator and detector. Finally, we discuss how to detect eavesdroppers who deliberately write memory to interfere with EarFisher's detection.

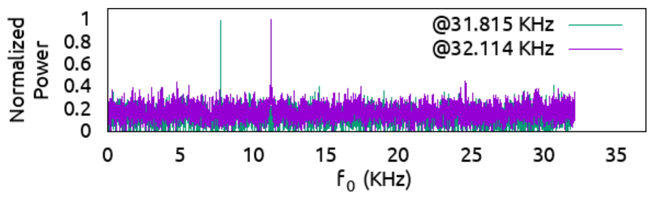
6.1 Sensing Memory EMRs

As discussed in section 4, memory EMR consists of a series of sub-clocks, where the i-th sub-clock is at $f_0 - if_m$. Due to minute manufacturing deviations, f_0 and f_m present a unique fingerprint, which distributes the sub-clocks of different devices to different frequencies. In the following, we first characterize memory clock fingerprint to study if it is sufficiently diverse to allow the separation of memory EMRs. We then discuss how to fuse sub-clocks to sense memory EMRs buried by noise, and develop a signal processing algorithm that exploits memory clock fingerprint to separate and track the memory EMRs of individual devices.

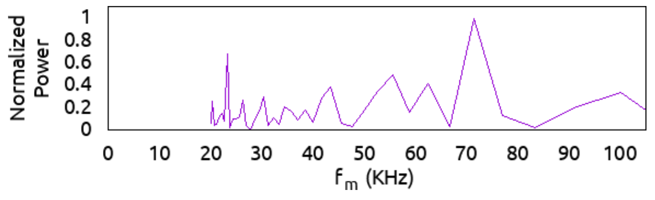
Characterizing memory clock fingerprints. To characterize the fingerprint of memory clock, we conduct measurements on 32 devices equipped with DDR3-1600, including a Dell Inspiron, a Thinkpad, and 30 identical desktops in the computer room of a university library. Fig. 6 plots the distribution of their memory clock fingerprints. We observe that f_0 differs even across the 30 identical desktops. In comparison, the offset of f_m is only significant across different devices, which should be attributed to the different modulation frequencies of their clock generators.



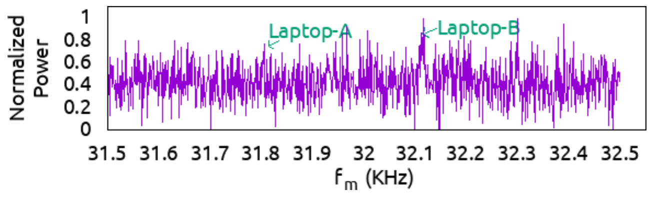
(a) Noisy spectrum after FFT.



(b) Results of folding.



(c) FFT.



(d) Auto-correlation.

Figure 7: Comparison between folding, auto-correlation and FFT over a noisy spectrum containing the memory EMRs of two computers whose f_m are 31.815 KHz and 32.114 KHz, respectively.

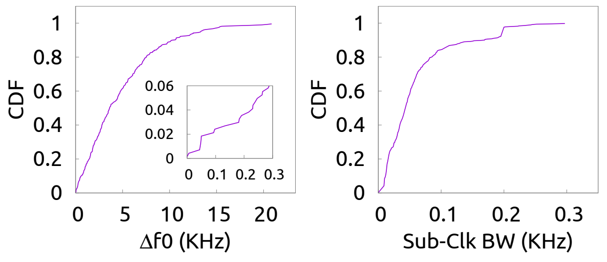


Figure 8: The CDFs of Δf_0 and sub-clock bandwidth.

To validate if the fingerprints are sufficiently diverse, consider an arbitrary pair of identical desktops. As they have similar f_m, the separation of their sub-clock components solely depends on the offset of f_0. Specifically, if the offset of f_0 is too small, sub-clock components will overlap in frequency in a pair-wise fashion. In this case, the overlapped bandwidth can be computed as,

$$\Delta BW = \frac{BW_a + BW_b}{2} - \Delta f_0,$$

where Δf_0 is the offset of f_0; BW_a and BW_b are the sub-clock bandwidth of the two devices, respectively. Clearly, their memory EMRs are separable if and only if $\frac{BW_a+BW_b}{2}$ is smaller than Δf_0. Fig. 8 further compares the distributions of Δf_0 and $\frac{BW_a+BW_b}{2}$ measured on the 30 identical desktops. We observe that Δf_0 is significantly larger than sub-clock bandwidth. Specifically, Δf_0 is larger than 300 Hz in 94% cases, whereas the bandwidth of all sub-clocks are smaller than 300 Hz.

Based on f_0 and sub-clock bandwidth measured on the 30 identical desktops, we further conduct a simulation to test the capacity of a memory ‘channel’, i.e., the maximum number of identical devices that can coexist on the same memory frequency without mixing memory EMRs. In each run of the simulation, we randomly add desktops to the memory channel until the produced memory EMRs become inseparable. After 10000 runs, we find that the average capacity is 7. We note that the capacity should be significantly higher if coexisting devices are different. In particular, for a pair of different devices, even if some of their sub-clock components are overlapping, others are likely separated because of different f_m.

Fusing sub-clocks. We then discuss how to fuse sub-clocks to boost the SNR of memory EMR. A key design requirement is to achieve computational efficiency, because the signal processing targets at high-resolution frequency spectra obtained by large FFT windows.

To this end, we propose a novel use of *folding* – a fast algorithm originally used by large radio telescopes to amplify periodic astronomical signals [23, 31, 39]. EarFisher utilizes folding to search for sub-clock components distributed over frequency. Suppose $\mathcal{P}$ represents the series of N points of the spectrum and $\mathcal{P}[i]$ ($i \in [1,N]$ is the amplitude of the ith point. The objective of folding is to search for a signal with a period of T. The spectrum is first divided into small windows of T points and then added in a window-wise fashion as,

$$\mathcal{F}_T[i] = \sum_{j=0}^{\lfloor \frac{N}{T} \rfloor - 1} \mathcal{P}[i + j * T].$$

When folding up the spectrum using a window size of f_m, the energies of sub-clock components will be fused while the sum of noise is likely smaller due to their non-periodicity. The position of *folding peak*, i.e., the i that maximizes $\|\mathcal{F}_T[i]\|$, is dependent on the offset between receiving frequency and the memory clock's f_0. Because the f_m of memory clock is unknown, EarFisher performs folding at each possible f_m to search for memory EMRs. Fig. 7a shows an example of a noisy spectrum containing the memory EMRs of two laptops, whose f_m is 31.815 KHz and 32.114 KHz, respectively. Fig. 7b plots the folded spectra where the peaks corresponding to the fused energies of sub-clocks can be clearly identified. In comparison, as shown in Fig. 7c and Fig. 7d, the performance of auto-correlation and FFT – two widely used signal processing algorithm of periodic signal detection – is worse than folding despite their higher computational overhead. Specifically, FFT fails to identify memory EMRs due to its poor resolution. Auto-correlation identified one of the laptops but is significantly more susceptible to noise than folding.

Separating memory EMRs. EarFisher exploits the diversity of memory clock fingerprints to separate and track memory EMRs. To this end, EarFisher performs two steps of processing iteratively. First, it folds up the spectrum at all possible f_m, and then identifies the highest folding peak caused by the device that has the strongest memory EMR. Name this device as Alice. Second, EarFisher outputs the highest folding peak, which reflects the fused amplitude of Alice's memory EMR, and then removes the sub-clock components of Alice from the spectrum. The goal is to eliminate possible peaks yielded by Alice in subsequent rounds of folding, which may prevent EarFisher from identifying the folding peaks of other devices. Note that each sub-clock component may include multiple spectral points depending on its bandwidth. EarFisher identifies sub-clock bandwidth using standard edge detection algorithm [19], and then removes all points included in the peak. The above procedure is repeated until the highest folding peak falls below a predefined threshold.

In practice, the spectrum may contain other signals produced by wireless communication. EarFisher classifies memory EMR and wireless communication based on two simple rules. First, the highest folding peak should trace-back to at least two peaks separated by the folding period in the spectrum. Second, the bandwidth of each trace-backed peak should not exceed 300 Hz – an empirical upper bound on sub-clock bandwidth obtained through extensive measurements. In comparison, the signal bandwidth of wireless communication is typically orders-of-magnitude higher in order to achieve a meaningful data rate.

Tracking memory EMRs. EarFisher tracks memory EMRs of different devices by using (f_0, f_m) as a device identity. In practice, (f_0, f_m) may experience small variance over time. To address this issue, EarFisher clusters folding peaks obtained at different time instants based on the euclidean distance of (f_0, f_m), where each cluster corresponds to one device. It then assigns a unique ID to each device and tracks the variation of f_0 and f_m using standard phase-locked loops.

6.2 Stimulator

The stimulator of EarFisher consists of two cooperative devices, which exchange packets to generate stimulus traffic. To detect eavesdroppers in a specific wireless network, the bait packets should be transmitted on the same frequency channel. In case the network to protect is operated on multiple channels, the stimulator can hop across channels to inject baits. In the following, we focus on the design of a Wi-Fi stimulator. The principle of the presented design is broadly applicable to other types of wireless networks.

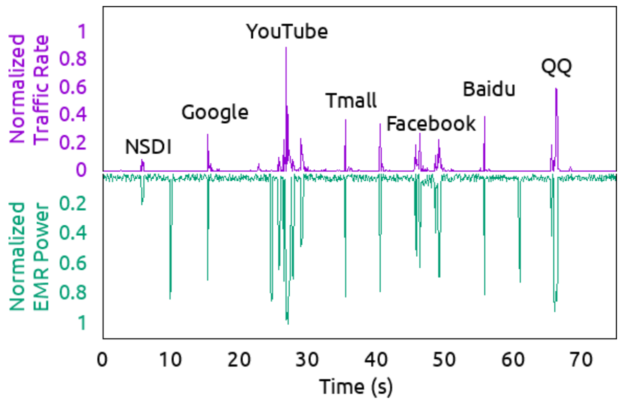


Figure 9: Eavesdropper's response to the stimuli of web page downloads.

Engineering stimulus traffic. EarFisher disguises the network of Wi-Fi stimulator as a WLAN where one device hosts a virtual access point and the other is attached as a client. To stimulate a Wi-Fi eavesdropper without incurring its alert, the client launches a sequence of webpage downloads at random time instants, where each download generates a short stimulus consisting of several MBs of data depending on the size of webpage[3]. Fig. 9 shows an eavesdropper's response to stimuli when it is sniffing on the downloads of the homepage of NSDI as well as 6 popular pages top-ranked in Alexa [2]. We observe that YouTube triggered the strongest response due to its large page size, suggesting that the stimulator should leverage media-rich pages for stimuli.

In practice, round-trip delays occurred at upper-layers prevent the stimulator from achieving a high throughput, thereby degrading the intensity of stimulus. For example, as can be seen in Fig. 9, the downloading traffic of Facebook page was divided into two parts due to large upper-layer delay, which significantly weakens the eavesdropper's response despite the large size of Facebook page. To address this issue, EarFisher first records the real traffic of webpage downloads, and then replays the traffic in the local network of stimulator.

Media access control. To further improve the effectiveness of stimulus, EarFisher leverages the following MAC-layer schemes. First, EarFisher utilizes the frame aggregation feature of 802.11 to bundle multiple bait packets in a single transmission, which effectively increases the throughput of stimulus traffic. Second, before transmitting bait packets, EarFisher uses RTS and CTS to mute normal network traffic. The goal is to reduce the chance of false alarm, which may happen if legitimate network nodes transmit or receive and thereby produce memory EMRs during stimulus period. One of our future work is to further control the timing of stimuli to minimize the interference with normal network traffic. This can be achieved by predicting the variation of normal network traffic using theoretical models [28] and then sending stimulus traffic only when the wireless channel is under-utilized.

[3]According to httparchive [4], the average webpage size has increased from 1.6 MB in 2014 to 4 MB in 2019.

6.3 Detector

At a high-level, the detector of EarFisher separates and tracks memory EMRs using the sensing primitive proposed in section 6.1, and then inspects each memory EMR to infer the response to traffic stimuli. Note that whenever transmitting bait packets, the stimulator will also emit memory EMR, which is difficult to distinguish from the response of an eavesdropper. To address this issue, a straightforward method is to count bait receivers and check if the number is larger than expected. In the design of EarFisher, we employ a simple alternative to walkaround receiver counting. Specifically, EarFisher employs two stimulators of different memory frequencies. In this case, because the eavesdropper's memory frequency must differ from that of at least one stimulator, the detector can quickly identify eavesdropper's memory EMRs based simply on emission frequency.

Profiling memory EMRs. To detect the presence of eavesdroppers, EarFisher compares the amplitudes of memory EMRs measured in the presence and absence of stimulus traffic, named as *stimulus set* and *baseline set*, respectively.

To profile the stimulus set, EarFisher first captures signals around the transmission time of bait packets, and then puts captured signals in a large FFT window. Once the window is completely filled, EarFisher runs the algorithm proposed in section 6.1 to identify memory EMRs. Suppose n memory EMRs are observed at $I^i = (f_0^i, f_m^i)$, $i \in (1,n)$. For each I^i, EarFisher establishes a stimulus set to record the amplitude of memory EMR, and then populates the set when new FFT windows are available.

Note that a surge of memory EMR under traffic stimuli could be a coincidence caused by system memory workload. To profile the probability of such coincidence, EarFisher keeps tracking the amplitude of memory EMR for each I^i to build the baseline set. To prevent memory EMRs caused by network activity from polluting the baseline set, EarFisher purges signals captured in the presence of ongoing network traffic, and then stitches remaining signals into FFT windows.

Statistical hypothesis testing. EarFisher then compares the stimulus and baseline set using a statistical hypothesis test called t-test, which is widely used to decide whether a drug has had a significant effect on the studied population. A t-test takes the means, variances, and the number of samples of the two compared sets, and then computes a t-value as follows,

$$t = \frac{\mu_{\text{stimulus}} - \mu_{\text{reference}}}{\sqrt{\frac{\sigma_{\text{stimulus}}}{n_{\text{stimulus}}} + \frac{\sigma_{\text{reference}}}{n_{\text{reference}}}}}, \tag{3}$$

where μ and σ are the mean and variance of a set, respectively; n is the number of samples, which determines the degrees of freedom of the test.

Once the t-value and degrees of freedom are determined, a p-value can be calculated. A large positive p-value is an evidence that μ_{stimulus} is significantly larger than μ_{ref}. EarFisher then compares the p-value with a a chosen level of statistical significance, denoted as α. Basically, a high α assures low false alarm rate but may raise miss detection. EarFisher exposes the configuration of α, which allows users to tune the balance between detection rate and false alarm.

6.4 Detecting Countermeasure

To counteract the detection of EarFisher, eavesdroppers knowing the design and presence of EarFisher may actively write memory to emit strong EMR, which will pollute the baseline set profiled by EarFisher, thus defeating the statistical test given in Eqn. 3. In the following, we propose a simple method to detect this countermeasure.

The dilemma of eavesdropper. Our insight is that, when actively writing memory, the eavesdropper faces a fundamental dilemma where intermittent writing leaves significant chance of exposing the response to stimuli, while consistent writing presents an abnormal pattern of memory EMR that can be distinguished from normal system memory workloads.

Specifically, to mask the response to traffic stimuli, the eavesdropper must erase the difference between the reference and stimulus set. However, because the stimulus set is built by stitching signals received under traffic stimuli, it captures the eavesdropper's memory EMR in a status of virtually consistent writing of memory. In order to defeat the statistical test, the eavesdropper must write memory at a comparable intensity to corrupt the baseline set.

On the other hand, due to the presence of hierarchical cache, a normal system rarely writes memory consistently. In particular, unlike wireless NICs that write received packets directly into memory, legitimate programs only read/write memory under cache miss, while modern Intel and AMD CPUs can maintain cache hit rate above 90%. Moreover, due to the high bus bandwidth of modern memory (ranging from a couple of GB/s of DDR to tens of GB/s of DDR4), the data transfer caused by normal memory workload typically completes in very short time, producing only intermittent bursts of memory EMRs. As an example, Fig. 10 shows the time-varying amplitude of memory EMR measured on a laptop running Ubuntu 18.04 and Windows 10 with no other programs. We observe that the operating systems alone yield noticeable variance of memory EMR. In contrast, consistent writing of memory can easily erase the variance, presenting a distinguishable pattern.

Exploiting the dilemma. To detect eavesdroppers who deliberately write memory to mask response to traffic stimuli, EarFisher complements the detector proposed in section 6.3 with an auxiliary detector to inspect abnormal EMR pattern caused by deliberate writing.

The auxiliary detector uses the variance of normalized memory EMR amplitude as a feature to investigate if the baseline set has been polluted by deliberate writing. However, because measuring memory EMR requires a large FFT window to suppress noise, obtaining a fine-grained estimation of EMR

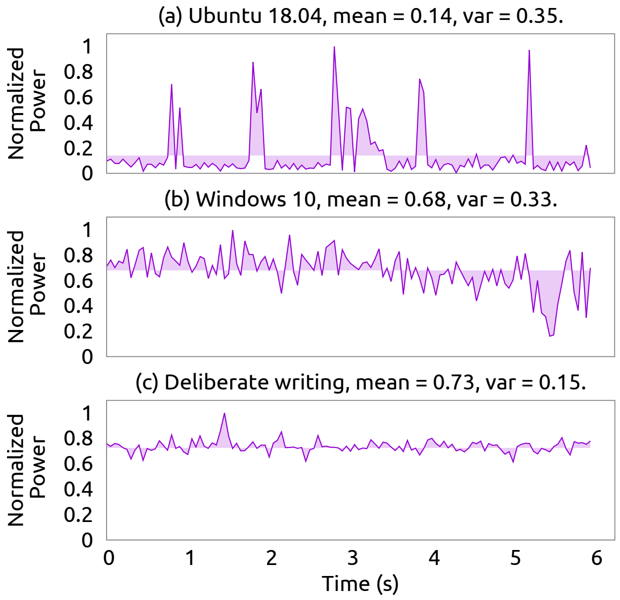


Figure 10: The time varying memory EMR induced by operating systems and deliberate writing.

variance may incur a long delay. EarFisher circumvents this issue by *oversampling* the baseline set. Specifically, EarFisher divides each FFT window in the baseline set into small blocks of signals, and then randomly picks blocks from different FFT windows to create new EMR samples. EarFisher then calculates the variance of created EMR samples, and repeats this for multiple rounds to profile a fine-grained distribution of EMR variance using a small number of FFT windows.

Once the distribution of EMR variance has been profiled, EarFisher performs t-test again to check if the mean of the distribution is smaller than an empirical threshold. To determine the threshold, we conduct extensive measurements on devices of different configurations. Our measurements lead to several findings. First, the variance is smaller on devices of larger cache and faster memory. Second, the variance is minimum when a devices is running no program except the operating system. Third, Windows typically demonstrates smaller variance than other operating systems. In our measurements condcuted on 39 devices, the minimum and maximum variance are 0.32 and 0.35, which are measured on a Dell Inspiron equipped with an 8 MB cache and DDR4-2133 running Windows 10, and a Thinkpad equipped with a 4 MB cache and DDR3-1600 running Ubuntu 18.04, respectively. Notice that the measured EMR variance may vary under different SNRs, we further profile the minimum EMR variance on Inspiron running Windows under different attenuation conditions. Before taking statistical test, EarFisher chooses a threshold based on the measured SNR of memory EMR. To avoid false alarm and account for devices of higher configurations, the current design of the auxiliary detector adopts a conservative threshold that is 10% lower than the empirically profiled minimum variance. Further refining the threshold for higher-end cache and memory configurations is left to our future work.

7 Evaluation

This section evaluates the performance of EarFisher in an 802.11n network. The current prototype of EarFisher employs BladeRF [3] to sense memory EMRs. To generate stimulus traffic, EarFisher replays the recorded traffic of YouTube page download at about 10 MB/s in its local stimulator network. Based on empirical measurements shown in Fig. 2, the FFT window size of memory EMR sensor is set to 1s, as larger windows increase detection latency but yield limited SNR gain. When performing statistical test to detect eavesdroppers, the stimulus and baseline set are profiled based on 1s and 3s of memory EMR signals, respectively. To improve the granularity of profiling, the baseline set is oversampled to create 10 FFT windows using the method described in section 6.4.

In the following, we first evaluate the accuracy of EarFisher in detecting eavesdropper and countermeasure, and then study its performance in a real deployment scenario where three EarFisher nodes are deployed to monitor an indoor environment of about 1600 ft^2.

7.1 Eavesdropper Detection

We conduct experiments on two commodity laptops, including a Thinkpad with DDR3-1600 and a Dell Inspiron with DDR4-2133. To evaluate the detection performance of EarFisher, we let the laptops act as eavesdropper (EV) and legitimate receiver (LR), and then compare the p-values computed by EarFisher. It is important to note that the p-values of EV and LR are NOT equivalent to the probability of detection and false alarm. Instead, the final detection result depends on the choose of p-value threshold, which EarFisher exposes to the user for configurable trade-off between detection rate and false alarm. In the following, we study the impacts of four key factors on EarFisher's detection performance, including the attenuation of memory EMR, system memory workloads, normal network traffic, and the interfering EMRs emitted by coexisting devices that have the same memory frequency.

Attenuation. We first study the impact of EMR attenuation on EarFisher's detection performance. We note that accurately controling EMR attenuation is difficult because we cannot connect an attenuator to the 'antenna' of the EMR emitter, i.e., the memory bus. To walk-around this issue, we first record the memory EMR of eavesdropper in close proximity, and then emulate a certain level of attenuation by mixing the recorded signal with an equivalent amount of white noise. In this experiment, the eavesdropper runs no software except OS, which allows us to exclude the interference of system memory workloads and study the optimal detection performance of EarFisher as a function of EMR attenuation.

As shown in Fig. 11, we observe that the p-values of DDR3 and DDR4 eavesdroppers are consistently higher than 0.9 before the amount of attenuation exceeds 29 dB and 21 dB, which typically translate to a line-of-sight path loss of about

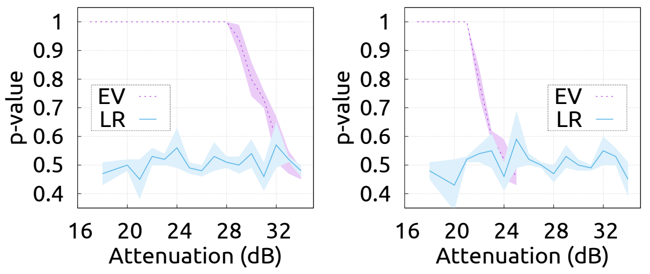
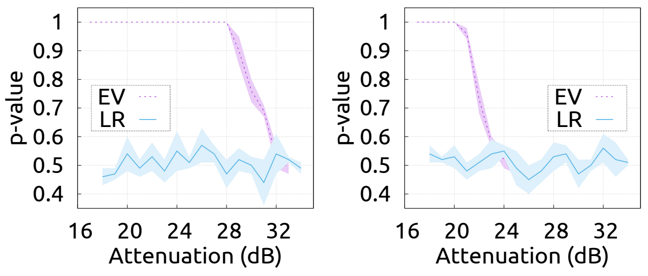
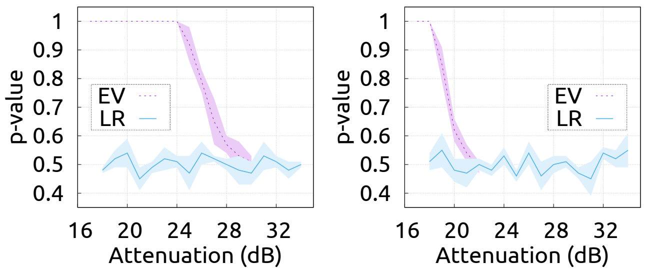


Figure 11: The p-values for the eavesdropper (EV) and the legitimate receiver (LR) as a function of EMR attenuation (left: DDR3, right: DDR4).



(a) Wireshark.



(b) VLC media player.

Figure 12: The impact of memory workloads (left: DDR3, right: DDR4).

30 m and 24 m, respectively. When the amount of attenuation further increases, the p-value for eavesdropper begins to decrease, as the surge of eavesdropper's memory EMR corresponding to the response to stimuli is gradually submerged by noise. We note that DDR3 is more resistant to attenuation than DDR4 because of the stronger EMR attributed to the higher operating voltage. In comparison, the p-value of legitimate receivers fluctuates in between 0.4 to 0.6 consistently despite the increase of attenuation. The results indicate that, if the user chooses a p-value threshold of 0.9, then EarFisher will detect all DDR3 and DDR4 eavesdroppers before attenuation reaches 29 dB and 21 dB while without miss-classifying any legitimate receivers as eavesdroppers.

System memory workload. Memory EMR produced by system memory workload will pollute the baseline profiled by EarFisher and thus will affect the result of statistical test. To study the impact of system memory workload, we let the laptops run two representative applications, including Wireshark – a widely used packet analyzer, and VLC media player, which reads memory intensively to load a high-definition video.

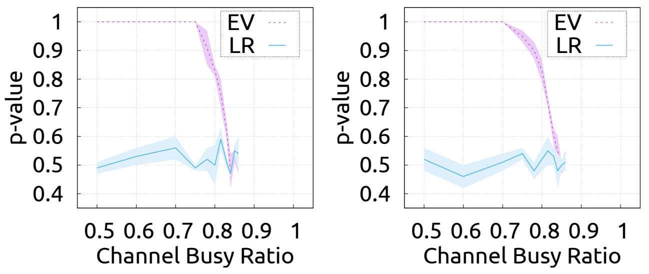


Figure 13: The impact of network traffic (left:DDR3, right:DDR4).

As shown in Fig. 12, compared with the results shown in Fig. 11, the attenuation resistance of EarFisher degrades by less than 1 dB and 5 dB when the eavesdroppers run Wireshark and VLC, respectively. VLC imposes a higher impact because it involves more frequent memory activities and thus produces a higher level of pollution to the baseline set. Nevertheless, even under the interference of VLC, the p-values of the DDR3 and DDR4 eavesdroppers are consistently higher than that of legitimate receivers as long as the levels of attenuation are below 25 dB and 19 dB, respectively.

We also observe that the p-values of legitimate receivers are not affected by Wireshark and VLC on both DDR3 and DDR4 laptops. This is because system memory workload will impact the baseline and stimulus set uniformly, hence the results of statistical test will remain unbiased.

Although we focus on only two representative applications in this experiment, we note that the interference caused by memory workload can be generally quantified using the busy ratio of memory bus. We will study EarFisher's performance as a function of memory busy ratio in section 7.2.

Network traffic. To study the impact of network traffic, we employ two additional 802.11n nodes to inject normal network traffic using iPerf [5]. In this experiment, we place eavesdroppers at 10m away from the EarFisher's detector. We then compare the p-values for eavesdroppers and legitimate receivers in the presence of different volumes of network traffic. We use channel busy ratio to quantify the interference produced by normal network traffic, because absolute volume can be misleading when characterizing interference intensity in wireless networks of different data rates.

As shown in Fig. 13, we observe that the detection performance of EarFisher is reliable as long as the channel busy ratio is below 73%. As channel busy ratio further increases, it becomes increasingly difficult to profile a clean baseline set not affected by network traffic. As a result, the p-value of the eavesdropper will begin to degrade.

Coexisting devices. We next evaluate EarFisher in a crowded environment where devices having the same memory frequency introduce interfering memory EMRs. We place the DDR3 eavesdropper in a library computer room that has 40 identical desktops, all having the same memory frequency as the eavesdropper. We then turn on desktops one by one and study the impact on EarFisher's performance.

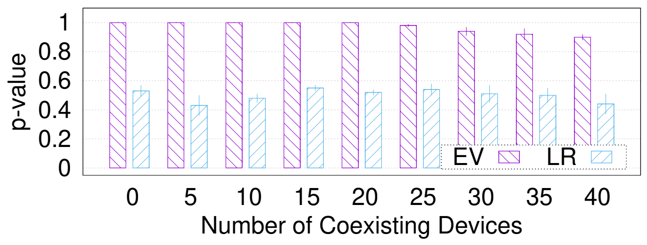


Figure 14: The impact of coexisting devices.

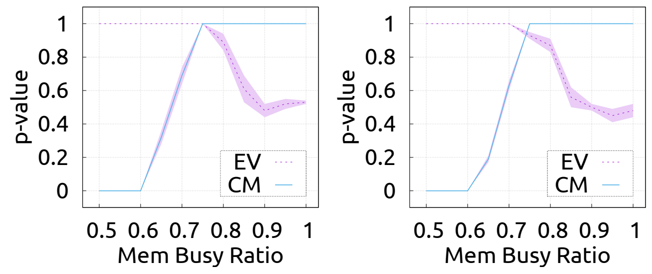


Figure 15: The p-values of the eavesdropper (EV) and the countermeasure (CM) detectors under different memory busy ratios (left: DDR3, right: DDR4).

Fig. 14 compares the p-values for eavesdropper and legitimate receiver as the number of coexisting desktops increases. We observe that EarFisher is fairly resistant to the interference of coexisting devices. Even when all 40 desktops are active simultaneously, the p-value for eavesdropper remains higher than 0.8, while the p-value of legitimate receiver fluctuates around 0.5. This should be attributed to the diversity of memory clock fingerprint, which allows EarFisher to separate the memory EMRs of different devices.

7.2 Countermeasure Detection

We next evaluate EarFisher's performance of detecting eavesdroppers that deliberately write memory to mask response to stimuli. We explore the parameter space of eavesdropper's countermeasure by tuning the ratio of deliberate writing, which results in different memory busy ratios. We then study the impacts on the p-values calculated by EarFisher's eavesdropper and countermeasure detector, denoted as EV and CM in Fig. 15, respectively. In this experiment, the eavesdropper is placed at 12 m away from EarFisher, which causes a path loss of about 15 dB.

As shown in Fig. 15, we observe that, as memory busy ratio increases, the p-value of eavesdropper detector decreases because the baseline set becomes increasingly polluted. In contrast, the p-value of countermeasure increases because the baseline set demonstrates increasing abnormality. For example, as the p-value for the DDR4 eavesdropper begins to drop when memory busy ratio increases to above 0.75, the p-value of countermeasure detector has reached 1.0. The results validate that the eavesdropper cannot defeat EarFisher's eavesdropper and countermeasure detectors at the same time.

7.3 A Deployment Case

In the following, we evaluate EarFisher on a testbed where three EarFisher nodes are deployed at different locations (i.e., S1-S3) in an indoor environment of 1600 ft^2, as shown in Fig. 16. We place the eavesdroppers at 9 locations in different rooms. We then randomly choose a node at another deployment location to stream a live video. The eavesdropper runs Wireshark and sniffs on the streamed video. All doors are closed during experiment. We study the precision and recall of EarFisher when the p-value threshold is set to 0.6 and 0.75, respectively.

As shown in Fig. 17, when the p-value threshold is set to 0.75, 8 of 9 DDR3 and 7 of 9 DDR4 eavesdroppers can be detected by at least one of the three EarFisher nodes with a recall higher than 90% while incurring no false alarms. In particular, eavesdroppers at location B and C can be accurately detected by EarFisher deployed at S3 and S2, despite the block of doors and walls. When the p-value threshold is relaxed to 0.6, the recall for the DDR4 eavesdropper at location F increases from 0% to 83%. However, this is at the cost of slightly reducing the precision from 100% to 94% and 97% at location A and B, respectively. We find that location G demonstrates as a blind spot due to severe EMR attenuation. We note that the performance of EarFisher can be further improved by leveraging advanced radio equipment such as high-gain LNA, better planing the deployment of EarFisher, or deploying more EarFisher nodes.

8 Discussion

In this section, we discuss important issues related to the design of EarFisher, including response mitigation-based countermeasures and EarFisher's limitations.

8.1 Response Mitigation

As discussed in section 6.3, EarFisher detects eavesdroppers by comparing the baseline and stimulus set to identify eavesdroppers' responses to stimuli. Accordingly, eavesdroppers can counter EarFisher by either polluting the baseline or mitigating the evidence of response in the stimulus set. In section 6.4 and section 7.2, we have shown how EarFisher effectively detects the first countermeasure. In the following, we discuss and analyze the second.

Eavesdropping on specified receivers. By modifying the firmware of wireless NIC, eavesdroppers can be configured to only sniff on packets transmitted to a specified receiver. EarFisher can detect such attacks by counting the receivers of packets sent to a specific address. This is feasible as long as the memory EMRs of the eavesdropper and the legitimate receiver can be separated in frequency spectrum, which is at a high probability as measured and discussed in section 6.1.

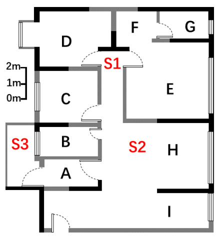


Figure 16: Testbed deployment.

	DDR3, p-value threshold = 0.75						DDR4, p-value threshold = 0.75						DDR4, p-value threshold = 0.60					
	S1		S2		S3		S1		S2		S3		S1		S2		S3	
	Rc	Pr	Rc	Pr	Rc	Pr	Rc	Pr	Rc	Pr	Rc	Pr	Rc	Pr	Rc	Pr	Rc	Pr
A	0	0	0.34	1.00	1.00	1.00	0	0	0	0	1.00	1.00	0	0	0.52	1.00	1.00	0.94
B	0	0	0.05	1.00	1.00	1.00	0	0	0	0	0.93	1.00	0	0	0.07	1.00	1.00	0.97
C	1.00	1.00	1.00	1.00	0	0	0	0	0.90	1.00	0	0	0.63	1.00	1.00	1.00	0	0
D	1.00	1.00	0	0	0	0	1.00	1.00	0	0	0	0	1.00	1.00	0	0	0	0
E	1.00	1.00	0.97	1.00	0	0	1.00	1.00	0	0	0	0	1.00	1.00	0	0	0	0
F	1.00	1.00	0	0	0	0	0	0	0	0	0	0	0.83	0.96	0	0	0	0
G	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
H	0.57	1.00	1.00	1.00	0	0	0	0	1.00	1.00	0	0	0.61	1.00	1.00	1.00	0	0
I	0.12	1.00	1.00	1.00	0	0	0	0	1.00	1.00	0	0	0	0	1.00	1.00	0	0

Figure 17: The recall (Re) and precision (Pr) for eavesdroppers deployed at different locations.

Weaponizing low-power wireless. Eavesdroppers may circumvent EarFisher's sensing by repurposing low-power wireless devices. As an example, ESP8266 [33] – a Wi-Fi enabled IoT – has a microcontroller that directly controls Wi-Fi chip via UART, thus avoiding strong memory EMR when receiving packets. However, such low-power architectural features typically limit the computation and storage capability of IoTs, making them ill-suited for security-intrusive tasks. For example, the on-board memory of ESP8266 is only 32 KiB for instruction and 80 KiB for data. In comparison, to crack a WEP key, the attacker needs to perform a computation over millions of encrypted packets [14]. As a result, to work as a full-fledged eavesdropper, low-power wireless devices have to rely on a host, which will leak strong memory EMR that can be captured by EarFisher.

Physical shielding. A physical method to mitigate response is to shield eavesdropper's memory bus to attenuate EMR. However, unlike shielding external emanation sources such as monitor cables, shielding memory bus can be prohibitively challenging and expensive.

8.2 Limitations

Excessive verdicts. By EarFisher's detection methodology, any device that digests others' packets in CPU-memory system will be convicted of eavesdropping. Unfortunately, it is difficult, if not impossible, to differentiate benign or malicious use of other's packets. As a result, all software radios will be identified as eavesdroppers as long as they transfer baseband signals to a host or process signals on board, both will emit strong memory EMRs. This is harsh but reasonable, because software radios process other devices' signal (albeit such processing may be only at the PHY layer) in an intrusted context with rich storage and computational resources capable of security- and privacy-intrusive tasks.

Besides software radios, recent wireless sensing and communication primitives such as backscatter [20] and localization [6, 38], may require Wi-Fi NICs to operate in monitor mode. To authenticate these applications, a possible method is to register legal eavesdropping devices a priori, and then let EarFisher count the number of eavesdroppers to determine the presence of illegitimate ones.

Low rate wireless networks. Experiment results shown in Fig. 4 suggest that a traffic stimulus of as slow as 2 MB/s suffice to trigger the eavesdropper's response. Unfortunately, this is still beyond the maximum data rate of many low-power wireless networks such as ZigBee. However, we expect that EarFisher will achieve better detection performance in the coming generation of high-rate wireless networks such as IEEE 802.11ax, which features GB/s level data rate, thus allowing for traffic stimuli of much higher intensities.

Blind spots. Eavesdroppers knowing the deployment of EarFisher may exploit locations subject to severe EM attenuation, such as room G shown in Fig. 16. Another example is to deploy eavesdroppers as *hidden terminals*, where the eavesdropper can hear the packets of a transmitter-of-interest, but is at a location relatively distant to EarFisher, such that the memory EMR cannot be accurately sensed. Such blind spots of detection can be mitigated by extending the coverage of EarFisher. Possible methods include but not limited to using high-gain LNA, leveraging advanced signal processing such as blind beamforming [12], or deploying more EarFisher nodes to monitor the area-of-interest.

9 Conclusion

This paper presents EarFisher – a system that detects wireless eavesdroppers by stimulating and sensing memory EMRs. Experiment results show that EarFisher accurately detects eavesdroppers despite poor signal conditions and the interference of normal network traffic, system memory workloads, and the interfering EMRs emitted by coexisting devices. We believe EarFisher provides an important block for building secure wireless networks. Incorporating EarFisher in wireless security protocols, such as to verify the confidentiality of key establishment, remains an important problem for future work.

Acknowledgments

We are grateful to NSDI reviewers and our shepherd, Andreas Haeberlen, for their insightful comments. This research was supported, in part, by funds from BvTech S.p.A. and the members of the Cybersecurity at MIT Sloan (CAMS) consortium (https://cams.mit.edu)

References

[1] Aircrack-ng. https://aircrack-ng.org.

[2] Alexa topsites. https://www.alexa.com/topsites.

[3] Bladerf. https://www.nuand.com.

[4] httparchive. https://httparchive.org/.

[5] Iperf. https://iperf.fr/.

[6] Fadel Adib, Zachary Kabelac, and Dina Katabi. Multi-person localization via rf body reflections. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[7] Dakshi Agrawal, Bruce Archambeault, Josyula R. Rao, and Pankaj Rohatgi. The em side-channel(s). In *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2002.

[8] Monjur Alam, Haider Adnan Khan, Moumita Dey, Nishith Sinha, Robert Callan, Alenka Zajic, and Milos Prvulovic. One&done: A single-decryption em-based attack on openssl's constant-time blinded rsa. In *USENIX Security Symposium*, 2018.

[9] Charles Bennett and Gilles Brassard. Quantum cryptography: Public key distribution and coin tossing. In *The International Conference on Computers, Systems and Signal Processing*, 1984.

[10] Boualem Boashash. Time-frequency signal analysis and processing: A comprehensive reference. 2003.

[11] Giovanni Camurati, Sebastian Poeplau, Marius Muench, Tom Hayes, and Aurelien Francillon. Screaming channels: When electromagnetic side channels meet radio transceivers. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[12] J. F. Cardoso and A. Souloumiac. Blind beamforming for non-gaussian signals. *IEE Proceedings F - Radar and Signal Processing*, 1993.

[13] Anadi Chaman, Jiaming Wang, Jiachen Sun, Haitham Hassanieh, and Romit Roy Choudhury. Ghostbuster: Detecting the presence of hidden eavesdroppers. In *ACM International Conference on Mobile Computing and Networking (MobiCom)*, 2018.

[14] Scott R. Fluhrer, Itsik Mantin, and Adi Shamir. Weaknesses in the key scheduling algorithm of rc4. In *Annual International Workshop on Selected Areas in Cryptography*, 2001.

[15] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic analysis: Concrete results. In *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2001.

[16] Daniel Genkin, Lev Pachmanov, Itamar Pipman, Eran Tromer, and Yuval Yarom. Ecdsa key extraction from mobile devices via nonintrusive physical side channels. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[17] Mordechai Guri, Assaf Kachlon, Ofer Hasson, Gabi Kedma, Yisroel Mirsky, and Yuval Elovici. Gsmem: Data exfiltration from air-gapped computers over gsm frequencies. In *USENIX Security Symposium*, 2015.

[18] Yi Han, Sriharsha Etigowni, Hua Liu, Saman Zonouz, and Athina Petropulu. Watch me, but don't touch me! contactless control flow monitoring via electromagnetic emanations. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[19] R. M. Haralick. Digital step edges from zero crossing of second directional derivatives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1984.

[20] Vikram Iyer, Vamsi Talla, Bryce Kellogg, Shyamnath Gollakota, and Joshua Smith. Inter-technology backscatter: Towards internet connectivity for implanted devices. In *Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2016.

[21] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *International Cryptology Conference (CRYPTO)*, 1999.

[22] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *International Cryptology Conference (CRYPTO)*, 1996.

[23] RVE Lovelace, JM Sutton, and EE Salpeter. Digital search methods for pulsars. *Nature*, 1969.

[24] Amitav Mukherjee and A Lee Swindlehurst. Detecting passive eavesdroppers in the mimo wiretap channel. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2012.

[25] Alireza Nazari, Nader Sehatbakhsh, Monjur Alam, Alenka Zajic, and Milos Prvulovic. Eddie: Em-based detection of deviations in program execution. In *The International Symposium on Computer Architecture (ISCA)*, 2017.

[26] Sanghoon Park, Lawrence E Larson, and Laurence B Milstein. Hidden mobile terminal device discovery in a uwb environment. In *IEEE International Conference on Ultra-Wideband*, 2006.

[27] Sanghoon Park, Lawrence E Larson, and Laurence B Milstein. An rf receiver detection technique for cognitive radio coexistence. *IEEE Transactions on Circuits and Systems*, 2010.

[28] V. Paxson and S. Floyd. Wide area traffic: the failure of poisson modeling. *IEEE/ACM Transactions on Networking*, 1995.

[29] Nader Sehatbakhsh, Alireza Nazari, Haider Khan, Alenka Zajic, and Milos Prvulovic. Emma: Hardware/software attestation framework for embedded systems using electromagnetic signals. In *IEEE/ACM International Symposium on Microarchitecture (Micro)*, 2019.

[30] Nader Sehatbakhsh, Alireza Nazari, Alenka Zajic, and Milos Prvulovic. Spectral profiling: Observer-effect-free profiling by monitoring em emanations. In *IEEE/ACM International Symposium on Microarchitecture (Micro)*, 2016.

[31] David H Staelin. Fast folding algorithm for detection of periodic pulse trains. *Proceedings of the IEEE*, 1969.

[32] Colin Stagner, Andrew Conrad, Christopher Osterwise, Daryl G Beetner, and Steven Grant. A practical superheterodyne-receiver detector using stimulated emissions. *IEEE Transactions on Instrumentation and Measurement*, 2011.

[33] Expressif Systems. Esp8266 overview. `https://www.espressif.com/products/socs/esp8266/`.

[34] Vivek Thotla, Mohammad Tayeb Ahmad Ghasr, Maciej J Zawodniok, Sarangapani Jagannathan, and Sanjeev Agarwal. Detection of super-regenerative receivers using hurst parameter. *IEEE Transactions on Instrumentation and Measurement*, 2013.

[35] Mathy Vanhoef and Frank Piessens. Key reinstallation attacks: Forcing nonce reuse in wpa2. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[36] Ben Wild and Kannan Ramchandran. Detecting primary receivers for cognitive radio applications. In *The IEEE International Symposium on Dynamic Spectrum Access Networks (DySPAN)*, 2005.

[37] Zhenkai Zhang, Zihao Zhan, Daniel Balasubramanian, Bo Li, Peter Volgyesi, and Xenofon Koutsoukos. Leveraging em side-channel information to detect rowhammer attacks. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[38] Yue Zheng, Yi Zhang, Kun Qian, Guidong Zhang, Yunhao Liu, Chenshu Wu, and Zheng Yang. Zero-effort cross-domain gesture recognition with wi-fi. In *ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2019.

[39] Ruogu Zhou, Yongping Xiong, Guoliang Xing, Limin Sun, and Jian Ma. Zifi: wireless lan discovery via zigbee interference signatures. In *ACM International Conference on Mobile Computing and Networking (MobiCom)*, 2010.

Adapting Wireless Mesh Network Configuration from Simulation to Reality via Deep Learning based Domain Adaptation

Junyang Shi
State University of New York at Binghamton

Mo Sha
State University of New York at Binghamton

Xi Peng
University of Delaware

Abstract

Recent years have witnessed the rapid deployments of wireless mesh networks (WMNs) for industrial automation, military operations, smart energy, etc. Although WMNs work satisfactorily most of the time thanks to years of research, they are often difficult to configure as configuring a WMN is a complex process, which involves theoretical computation, simulation, and field testing, among other tasks. Simulating a WMN provides distinct advantages over experimenting on a physical network when it comes to identifying a good network configuration. Unfortunately, our study shows that the models for network configuration prediction learned from simulations cannot always help physical networks meet performance requirements because of the simulation-to-reality gap. In this paper, we employ deep learning based domain adaptation to close the gap and leverage a teacher-student neural network to transfer the network configuration knowledge learned from a simulated network to its corresponding physical network. Experimental results show that our method effectively closes the gap and increases the accuracy of predicting a good network configuration that allows the network to meet performance requirements from 30.10% to 70.24% by learning robust machine learning models from a large amount of inexpensive simulation data and a few costly field testing measurements.

1 Introduction

Recent years have witnessed rapid deployments of wireless mesh networks (WMNs) for industrial automation [38, 95], military operations [57], smart energy [80], etc. For instance, IEEE 802.15.4-based industrial WMNs, also known as wireless sensor-actuator networks (WSANs), are gaining rapid adoption in process industries over the past decade due to their advantage in lowering operating costs [51]. Battery-powered wireless modules easily and inexpensively retrofit existing sensors and actuators in industrial facilities without the need to run cables for communication and power. Industrial standard organizations such as HART [31], ISA [38], IEC [37], and ZigBee [104] are actively pushing the real-world implementations of WSANs for industrial automation. For example, more than 54,835 WSANs that implement the WirelessHART standard [95] have been deployed globally by Emerson Process Management to monitor and control industrial processes [24].

Although WMNs work satisfactorily most of the time thanks to years of research, they are often difficult to configure as configuring a WMN is a complex process, involving theoretical computation, simulation, and field testing, among other tasks. Simulating a WMN provides distinct advantages than experimenting on a physical network when it comes to identifying a good network configuration: a simulation can be set up in less time, introduce less overhead, and allow for different configurations to be tested under exactly the same conditions. Significant efforts have been made to investigate the characteristics of wireless communication in the literature. For instance, there has been a vast array of research that empirically studied the low-power wireless links with different platforms, under varying experimental conditions, assumptions, and scenarios [6]. Decades of research have gathered precious knowledge and produced a set of mathematical models that capture the characteristics of wireless links, interference, etc, and enable the development of wireless simulators, such as TOSSIM [44, 84], Cooja [17, 65], OMNeT++ [63, 89], and NS-3 [61].

However, it is still very challenging to date to set up a simulation that captures extensive uncertainties, variations, and dynamics in real-world WMN deployments. Our study shows that the models for network configuration prediction learned from simulations cannot always help physical networks meet performance requirements because of the *simulation-to-reality gap*; therefore the advantages of using simulations to reduce experimental overhead, improve flexibility, and enhance repeatability come at the expense of questionable credibility of the results. On the other hand, data collection from many WMN deployments, which include the ones in industrial facilities, is costly; therefore it is difficult to obtain sufficient information to train a good model or identify an optimal policy for network configurations by relying solely on field testing.

In this paper, we formulate the network configuration prediction into a machine learning problem, use the configurations of a WirelessHART network [95] as an example to illustrate the simulation-to-reality gap, and then employ deep learning based domain adaptation to close the gap. Specifically, this paper makes the following contributions:

- We present the simulation-to-reality gap in network configurations and show that the models for network configuration prediction learned from simulations cannot always help physical networks meet performance requirements;
- We develop a teacher-student neural network[1] that learns robust machine learning models for network configuration prediction from a large amount of inexpensive simulation data and a few costly physical measurements; to our knowledge, our work represents the first systematic study of the effectiveness of domain adaptation in closing the simulation-to-reality gap in network configurations;
- We implement our method, evaluate it using four simulators and a physical testbed, and repeat our evaluation with different network topologies under various wireless conditions. Experimental results show that our method can significantly improve the prediction accuracy and help physical networks meet performance requirements.

The remainder of our paper is organized as the following sections. Section 2 reviews the related work. Section 3 introduces the background of WirelessHART networks. Section 4 presents our problem formulation, our feature selection study, the simulation-to-reality gap, and our method that closes the gap. Section 5 shows the design of our teacher-student neural network. Section 6 evaluates our method. Section 7 concludes the paper.

2 Related Works

The current practices in network configurations rely largely on experience and rules of thumb that involve a coarse-grained analysis of network loads or dynamics during a few field trials. For example, the WirelessHART standard specifies the use of all available channels after a human operator manually blacklists noisy ones [95], and Emerson Process Management [22] recommends using a constant value (60% in general or 70% for control and high speed monitoring) as the packet reception ratio (PRR) threshold to select links for routing [23]. Unfortunately, recent studies show that such specifications are problematic, because using more channels or a fixed PRR threshold is not always desirable in WirelessHART networks [30,75,76]. In the literature, significant research efforts have been made to model the characteristics of wireless networks and optimize network configurations through mathematical techniques such as convex optimization [52], game theory [2], and meta heuristics [73]. For instance, the characteristics of low-power wireless links have been studied empirically with different platforms, under varying experimental conditions, assumptions, and scenarios [6]. Runtime adaptation methods have been developed to improve the performance of wireless sensor networks (WSNs) by adapting a few parameters in the physical and media access control (MAC) layers [20,25,70,90,105]. Those methods are not directly applicable to configure a network with many interplaying parameters.

As wireless deployments become increasingly hierarchical, heterogeneous, and complex, a breadth of recent research has reported that resorting to advanced machine learning techniques for wireless networking presents significant performance improvements compared to traditional methods. Deep learning has been used to handle a large number of network parameters and automatically uncover correlations that would otherwise have been too complex to extract by human experts [5,14,42,54,97,101] and reinforcement learning has been employed to enable network self-configurations [18,32,36,45,47,53,56,59,60,67,72,74,91,93,96,98–100,103]. The key behind the remarkable success of those data-driven methods is the capability of optimizing a huge number of free parameters [33,35] to capture extensive uncertainties, variations, and dynamics in real-world wireless deployments, which not only yield complex features, such as communication signal characteristics, channel quality, queuing state of each device, and path congestion situation, but also have many control targets, such as resource allocation, queue management, and congestion control.

However, data collection from many wireless deployments that are not easily accessible (e.g., the ones in industrial facilities) is costly; therefore it is difficult to obtain sufficient information to train a good model or identify an optimal policy for network configurations. In such scenarios, the benefits of employing learning-based methods that require much data are outweighed by the costs. Industry has consequently shown a marked reluctance to adopt them. To address this limitation, there has been increasing interest in using simulations to identify good network configurations [7,40,46,49,75,76,79,82]. Unfortunately, our study shows that a straightforward deployment of a model learned from simulations results in poor performance in a physical network due to the simulation-to-reality gap.

Domain adaptation aims to learn from one or multiple source domains and produce a model that performs well on a related target domain; the assumption is that the source and

[1]To eliminate ambiguity, we use the word "network" to denote a wireless network and use the word "neural network" to represent a deep learning model in this paper.

target domains are associated with the same label space. It has been successfully used in computer vision [69, 92], natural language processing [66], and building occupancy estimation [3, 102]. Studies have shown that domain adaptation can mitigate the harmful effects of domain discrepancy by optimizing the representation to minimize some measures of domain shift, such as maximum mean discrepancy [13] or correlation distances [27]. Compared to fine-tuning the deep learning model, which is pre-trained using simulation data, employing domain adaptation is expected to close the gap between the simulated network (source) domain and the physical network (target) domain with fewer costly physical measurements. Recent work has focused on transferring deep neural network (DNN) representations from a labeled source dataset to a target domain where labeled data is sparse or non-existent. The main strategy is to guide feature learning via minimizing the difference between the source and target feature distributions. The maximum mean discrepancy (MMD) has been successfully used for domain adaptation, which computes the norm of the difference between two domain means [29, 86]. Several methods employed an adversarial loss to minimize domain shift and learned a representation that is simultaneously discriminative of source labels while not being able to distinguish between domains [19,26]. Despite the extensive literature on domain adaptation, little work has been done to investigate whether it can be applied to close simulation-to-reality gap in network configurations.

3 Background of WirelessHART Networks

To meet the stringent reliability and real-time requirements of industrial applications, WirelessHART networks [95] made a set of specific design choices that distinguish themselves from traditional WSNs designed for best effort services [51]. A WirelessHART network is managed by the centralized network manager, a software module, which is responsible for managing the entire network that includes generating routes, scheduling all transmissions, and selecting network parameters. Network devices include a set of field devices (sensors and actuators) and multiple access points. Each network device is equipped with a half-duplex omnidirectional radio transceiver compliant with the IEEE 802.15.4 standard [1].

WirelessHART networks adopt the time-slotted channel hopping (TSCH) technique [85], which combines time-slotted medium access, channel hopping, and multi-channel communication to provide time-deterministic packet deliveries[2]. Under TSCH, time is divided into 10*ms* time slots, each of which can be used to transmit a packet and receive an acknowledgment between a pair of devices. The network uses up to 16 channels in the 2.4 GHz ISM band and

[2]Packets must be delivered along the data flow (from a sensor to an access point and then to an actuator) by the specified time deadline.

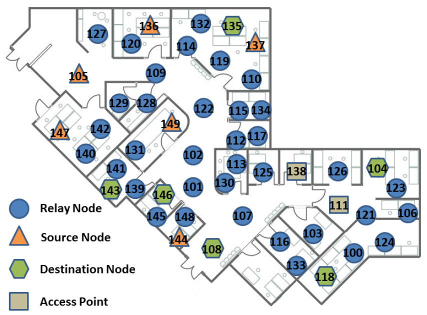


Figure 1: Device deployment on our testbed. The device ID ranges from 100 to 149.

performs channel hopping in every time slot to combat narrow band interference. WirelessHART networks support two types of routing: source routing and graph routing. Source routing provides a single directed path from each data source to its destination. Graph routing is designed to enhance network reliability by providing redundant routes between field devices and access points. A packet may be transmitted through the backup routes if the links on the primary path fail to deliver it.

4 Methodology

In this section, we first describe our experimental setup and data collection method. Then we formulate the network configuration prediction as a machine learning problem and present our feature selection study and the simulation-to-reality gap. Finally, we introduce our deep learning based domain adaptation method, which closes the gap.

4.1 Experimental Setup and Data Collection

We adopt the open-source implementation of WirelessHART networks provided by Li et al. [94] and configure six data flows on our testbed, which consists of 50 TelosB motes [81]. Figure 1 shows the device deployment on our testbed and Table 1 lists the source node (sensor), the destination node (actuator), the data generation interval (period), and the priority of each data flow. We employ the rate monotonic scheduling [48], an optimal fixed-priority policy, to generate the transmission schedule, set the data delivery deadline of each data flow to its period, and configure the devices with ID 111 and 138 to serve as two access points.

We consider three configurable network parameters, which include (i) the PRR threshold for link selection R, (ii) the number of channels used in the network C, and (iii) the number of transmission attempts scheduled for each packet A, and three network performance metrics, which include (1) the end-to-end latency L, (2) the battery lifetime B, and (3) the end-to-end reliability E. We consider $R \in \{0.7, 0.71, 0.72, ..., 0.90\}$[3], $C \in \{1,2,3,4,5,6,7,8\}$, and

[3]Emerson Process Management [22] recommends using a constant value

Table 1: Data flows.

Flow ID	Source	Destination	Period (ms)	Priority
1	147	146	500	1
2	144	143	500	2
3	105	104	500	3
4	149	118	1000	4
5	136	135	1000	5
6	137	108	1000	6

$A \in \{1,2,3\}$ as the possible parameter values, and combine them to create 744 $(31*8*3)$ network configurations. Please note that some network configurations make the network manager generate the same routes and transmission schedule. After removing all redundancy (the configurations leading to the same routes and transmission schedule), there are 88 distinct network configurations left under our experimental setup.

After deploying the data flows on the testbed, we implement the same network in the simulator[4], feed the PRR and noise traces, the routes, and the transmission schedule collected from the physical network into the simulator, and then run simulations to evaluate network performance under each network configuration. Specifically, the simulator generates simulated L, B, and E values under each network configuration (R,C,A). The network performance (L, B, and E values) is computed in every 50s. 75 network performance traces are collected under each network configuration. In total, we collect 6,600 $(88*75)$ data traces from simulations. Then, we run experiments on our testbed and measure the network performance under each network configuration. Similarly, we collect 6,600 data traces from our testbed. The data gathered from the simulated network and the physical network is denoted as $\mathcal{D}^s$ and $\mathcal{D}^p$, respectively.

4.2 Network Configuration Prediction

The primary task in network configurations is to select the configuration (the selections of parameters R, C, and A), which allows the network to meet the performance requirements (L, B, and E) specified by the application. The parameter selection should be as *accurate* as possible with *minimal data collection overhead*. We formulate the network configuration prediction task as a machine learning problem. Let $\mathbf{x} = concatenation(L,B,E)$ denote the given network performance requirements and $\mathbf{y} = concatenation(R,C,A)$ denote the configuration, which allows the network to meet performance requirements. The goal is to learn a nonlinear mapping $f_\theta(\cdot) : \mathbf{x} \rightarrow \mathbf{y}$. Based on the specific application, the user can set the performance requirements ($\mathbf{x}$). The input features in $\mathbf{x}$ are selected by the feature selection study in Section 4.3.

We use θ to denote the model parameters that are learned from data in a data-driven manner. Given the fact that the network configuration values ($\mathbf{y}$) can be discretized without losing the generality, we further restrict f_θ as a discriminative model to solve a classification problem: an application can set its performance requirements ($\mathbf{x}$), and the classifier (f_θ) will predict the network configuration ($\mathbf{y}$) to satisfy the application requirements. This data-driven learning-based model can take advantage of a large amount of data to consistently improve its performance. Experimental results (See Section 6.2) show that it significantly outperforms traditional optimization-based methods such as Response Surface Methodology (RSM) [9] and Kriging surrogate modeling approach [78]. The latter usually suffers the issues that include limited predictive power and being vulnerable to uneven data distribution [15].

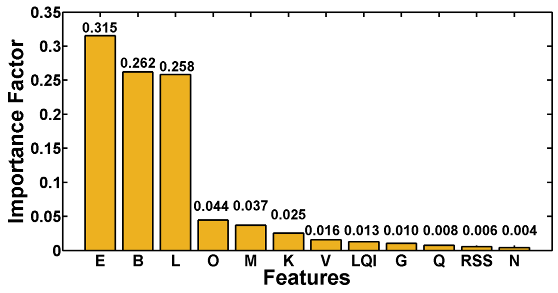


Figure 2: Importance factors of different features when using tree-based feature selection method. Under the tree-based method, the features that are selected at the top of the trees are in general more important than the features that are selected at the end nodes of the trees, as generally, the top splits lead to bigger information gains. We use the normalized importance factor generated by the tree-based method as a metric for feature selection.

4.3 Feature Selection

In addition to the features (L, B, and E) that represent performance requirements, we consider nine other features, which include the received signal strength RSS [8], the link quality indicator LQI [6], the background noise G [6], the packet delay variation O, the power consumption variation K, the network reliability variation M, the received signal strength variation V, the link quality indicator variation Q, and the background noise variation N. Using all features that are relevant to the network configuration prediction problem may not necessarily achieve the best performance but rather increases computational cost and data collection overhead. We perform a study that employs three classic feature selection methods (the tree-based method [50], the univariate feature selection method [39], and the recursive feature elimination

(0.6 in general or 0.7 for control and high speed monitoring) for R [23]. We did not consider R lower than 0.7 because of the consistent low reliability we observed.

[4]We repeat our experiments using four simulators: TOSSIM, Cooja, OMNeT++, and NS-3.

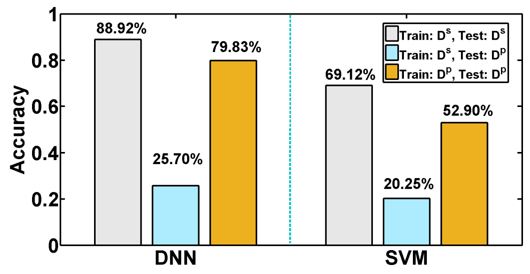


Figure 3: Modeling accuracy when model is trained and tested on different data sets ($\mathcal{D}^s$: the simulation data produced by OMNeT++ and $\mathcal{D}^p$: the physical data). The difference between the grey bar and the blue bar indicates the simulation-to-reality gap.

method [16]) to pick the most useful features. Figure 2 plots the importance factors of different features when we use the tree-based method. As Figure 2 shows, L, B, and E have much higher importance factors (0.315, 0.262, and 0.258) than the rest. Similar results are observed when we use other methods. Therefore, we use L, B, and E as the input features for WirelessHART networks. Please note that our method can accept more features for other networks.

4.4 Simulation-to-reality Gap

Our goal is to learn a classifier to predict network configurations on the physical data. However, it is a nontrivial task to learn the model from either the physical data ($\mathcal{D}^p$) or the simulation data ($\mathcal{D}^s$). Instead, we propose to use both $\mathcal{D}^p$ and $\mathcal{D}^s$ to learn the model as explained in the next section.

Using only physical data ($\mathcal{D}^p$): This would result in significant time and energy consumption due to the costly data collection process. We first leverage the physical data ($\mathcal{D}^p$) collected from the physical network to train machine learning models and explore its feasibility for our network configuration prediction problem. We employ two machine learning models, DNN and support vector machine (SVM), for classification. The input to the models is network performance requirements and the output is network configurations. We normalize the collected data ($\mathcal{D}^p$) into the $[0,1]$ range and split it randomly for training and testing. The yellow bars in Figure 3 show the modeling accuracy[5], when DNN and SVM models are used for the network configuration prediction, respectively. Both DNN and SVM models trained based on the physical data can provide high modeling accuracy when we test the models on the physical data (DNN: 79.83% and SVM: 52.90%), as the yellow bars show. This justifies the feasibility of our proposed machine learning method in Section 4.2 for the network configuration prediction and we may use the measurements collected from the physical network to train a good model. Unfortunately, relying on running experiments on a physical network to explore the configuration parameter space is impractical in many cases because running experiments on a physical network is very costly and time-consuming. The left side of Table 2 shows the modeling accuracy, data collection time, and device energy consumption when we train the DNN model with different sizes of the physical data (collected from a physical network). The modeling accuracy increases significantly from 19.39% to 79.83% with the size of the training set ($\mathcal{D}^p$) that increases from 88 traces to 3,960 traces. However, the time spent on collecting the training data ($\mathcal{D}^p$) increases from 1.22*hours* to 55.00*hours*. Moreover, the energy consumed by each field device for data collections on average increases from 310.61J to 13,974.26J, which represents 0.73% and 32.73% of its total energy capacity.

Using only simulation data ($\mathcal{D}^s$): This would result in low modeling accuracy due to the simulation-to-reality gap. The simulation data can be quickly and cheaply obtained from a simulator. As the right column of Table 2 show, the time spent on generating the simulation data varies from 27.41s to 1,231.40s and no energy is consumed by any field devices. However, a classifier that is trained based on the simulation data ($\mathcal{D}^s$) may suffer the following issue when applied on the physical data. As the grey bars in Figure 3 show, both models provide high modeling accuracy when we train based on the simulation data ($\mathcal{D}^s$) and test the models on the simulation data (DNN: 88.92% and SVM: 69.12%). However, the modeling accuracy drops rapidly when we test the models on the physical data ($\mathcal{D}^p$) as shown in blue bars (DNN: 25.70% and SVM: 20.25%). The differences on the modeling accuracy (DNN: 63.22% and SVM: 48.87%) clearly show the effect of the simulation-to-reality gap, a subtle but important discrepancy between reality and simulation that prevents the simulated experience from directly enabling effective real-world performance [12, 77]. The simulation-to-reality gap exists in network configurations because the theoretical models adopted by the simulator cannot capture all real-world performance-related factors. For example, the prerecorded noise traces and the probability-based prediction on packet reception cannot precisely capture the effects of packet failures caused by extensive uncertainties, variations, and dynamics in real-world wireless deployments (see Section 6.5). We observed similar discrepancy gaps when using Cooja, TOSSIM, OMNeT++, and NS-3. Because of the simulation-to-reality gap, the machine learning models trained based on simulation data ($\mathcal{D}^s$) for network configurations, no matter how large the data volume is, may not generalize well to a physical network.

4.5 Close the Gap by Domain Adaptation

The observations presented in Section 4.4 motivate us to explore the feasibility of using a substantial amount of inexpen-

[5]The modeling accuracy is defined as, given a set of input network performance requirements (L,B,E), the percentage of the testing set that a model can select the network configuration (R,C,A), which allows the network to meet performance requirements.

Table 2: Modeling accuracy (%), data collection time (s), and device energy consumption (J) when using the physical data ($\mathcal{D}^p$) or the simulation data ($\mathcal{D}^s$) produced by OMNeT++ for training. For comparison, our solution achieves 70.24% accuracy with only 440 data samples which are collected in 22,000s with 1,502.88J of energy (see Section 6.2).

# of Data Samples Used for Training	From a Physical Network (Train: $\mathcal{D}^p$, Test: $\mathcal{D}^p$)			From Simulations (Train: $\mathcal{D}^s$, Test: $\mathcal{D}^p$)		
	Accuracy (%)	Collection Time (s)	Energy (J)	Accuracy(%)	Collection Time (s)	Energy (J)
88	19.39	$4.40*10^3$	310.61	6.52	27.41	0
528	42.16	$2.64*10^4$	1,863.53	13.70	163.09	0
968	57.92	$4.84*10^4$	3,416.34	17.69	301.95	0
2,024	67.68	$1.01*10^5$	7,143.11	20.17	633.11	0
3,080	78.82	$1.54*10^5$	10,869.61	22.44	933.99	0
3,960	79.83	$1.98*10^5$	13,974.26	25.70	1,231.40	0

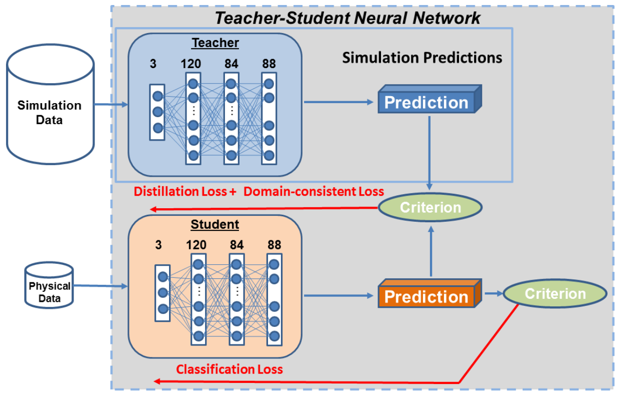


Figure 4: Our teacher-student neural network.

sive simulation data together with a small amount of costly physical data to train the model for network configuration prediction. To this end, our objective narrows down from solving a classification problem to using domain adaptation to address the domain discrepancy issue. Specifically, we first gather N^s data tuples by running simulations (source domain) and then acquire N^p data tuples by conducting experiments on the physical network (target domain). We assume $N^s \gg N^p$ due to the significant data collection overhead on the physical network (See Section 4.4). We assume that the source and target domains are characterized by different probability distribution q_1 and q_2, respectively. Our goal is to construct a deep learning model that can learn transferable features that bridge the cross-domain discrepancy and build a classifier $\mathbf{y} = f_\theta(\mathbf{x})$, which can maximize the target domain accuracy ($f_s \rightarrow f_p$) by using a small amount of physical data (N^p). The detailed design of our teacher-student neural network will be discussed in Section 5.

5 Teacher-Student Neural Network

In this section, we present our teacher-student neural network for domain adaptation. Our goal is to build a classifier that can maximize the target domain (physical network) accuracy by using a small amount of physical data (N^p) and adequate simulation data (N^s) where $N^s \gg N^p$ due to the significant data collection overhead (See Section 4.4) on the physical network. The teacher and student use independent parameters and the teacher generates the soft labels [4, 34] to transfer its knowledge to the student. Figure 4 shows our teacher-student neural network. The first stream (teacher) operates on the simulation data and the second stream (student) operates on the physical data. Classification loss, distillation loss, and domain-consistent loss are used in the training process for the student.

5.1 Teacher Neural Network

The teacher takes advantage of the large amount of simulation data for training and the training data ($\mathcal{D}^s$) consists of a total number of N^s data tuples. We follow Multilayer Perceptron (MLP) [71] to design the architecture of three layers: 120 and 84 neurons in the first two hidden layers, and 88 neurons in the output layer to represent the totally 88 distinct configuration categories. Rectified linear unit (ReLU) and softmax are employed to activate the hidden and output layers, respectively. The teacher's parameters (θ_1) are learned by minimizing the cross-entropy loss:

$$\mathcal{L}(\theta_1) = -\mathop{\mathbb{E}}_{\mathbf{x}\sim\mathcal{D}^s} \mathbf{y} log(f_{\theta_1}(\mathbf{x})), \tag{1}$$

where $\mathcal{D}^s$ denotes the training data generated from simulations, θ_1 denotes the teacher's parameters, y denotes the one-hot label, and $f_{\theta_1}(\mathbf{x})$ is the prediction made by the teacher. We use the Adam optimizer [41] with a learning rate of 0.01 to optimize the parameters of the teacher. A total number of 100 training epochs with a batch size of 128 have been used to train the neural network.

5.2 Student Neural Network

We train the student based on the N^p physical data with the help of the teacher. The student can be quickly learned using only a few shots of physical data ($N^p \ll N^s$). To achieve this, we leverage the teacher to facilitate the training of the student where knowledge is transferred from the simulation domain to the physical domain. The student shares the same architecture with the teacher but uses independent parameters. ReLU and softmax are employed to activate the hidden and output layers, respectively. The student's parameters (θ_2) are learned by minimizing the following loss:

$$\mathcal{L}(\theta_2) = \mathcal{L}_{cls} + \alpha \mathcal{L}_{dis} + \beta \mathcal{L}_{mmd} \tag{2}$$

where α, and β are weights. We empirically set $\alpha = 1$, and $\beta = 0.2$ which can provide good performance.

Classification loss $\mathcal{L}_{cls}$: This loss function allows the student to learn from the limited (N^p) physical data through employing the cross-entropy loss:

$$\mathcal{L}_{cls} = - \mathop{\mathbb{E}}_{\mathbf{x} \sim \mathcal{D}^p} \mathbf{y} log(f_{\theta_2}(\mathbf{x})), \tag{3}$$

where $\mathbf{y}$ is the one-hot label and $f_{\theta_2}(\mathbf{x})$ is the prediction made by the student.

Distillation loss $\mathcal{L}_{dis}$: This loss function allows the teacher to transfer its knowledge to the student. The generalization ability of the student can be enhanced by the loss generated by the soft labels, which carry the information of probability distribution for each class [4, 34]. To compute $\mathcal{L}_{dis}$ with soft labels, we use the following formula:

$$\mathcal{L}_{dis} = - \mathop{\mathbb{E}}_{\mathbf{x} \sim \mathcal{D}^s} \mathbf{q} log(f_{\theta_2}(\mathbf{x})), \tag{4}$$

where $f_{\theta_2}(\mathbf{x})$ is the prediction made by the student and $\mathbf{q}$ is the tempered softmax probability generated by the teacher. $\mathbf{q}$ is computed by:

$$\mathbf{q} = \frac{exp(z_i/T)}{\sum_j^k exp(z_j/T)} \tag{5}$$

where T is the temperature [34] and z_i is the pre-softmax output of the teacher. When T increases, the soft label q approaches a uniform distribution and the probability distribution generated by the softmax function becomes softer, which provides more information as to which class the teacher finds more similar to the predicted class, instead of giving a hard prediction that indicates which class is correct. We set $T = 10$ to generate soft labels for the student.

Domain-consistent loss $\mathcal{L}_{mmd}$: This loss function is employed to achieve domain-consistent representations between the source and target domains. Matching the distributions in the original input feature space is not suitable because some features may have been distorted by the domain shift. The key idea of domain-consistent regularization is to align two domains, the target (physical data) and the source (simulation data), in a latent embedding space. Our method uses the MMD [21] to achieve this goal. MMD is a hypothesis test that tests whether two samples are from the same distribution by comparing the means of the features after mapping them to a Reproducing Kernel Hilbert Space (RKHS) [68]. We calculate the loss as:

$$\mathcal{L}_{mmd} = || \mathop{\mathbb{E}}_{\mathbf{x} \sim \mathcal{D}^s} f_{\theta_1}(\mathbf{x}) - \mathop{\mathbb{E}}_{\mathbf{x} \sim \mathcal{D}^p} f_{\theta_2}(\mathbf{x}) || \tag{6}$$

where $f_{\theta_1}(\cdot)$ and $f_{\theta_2}(\cdot)$ denote the pre-softmax output of the teacher and the student, respectively. We use a learning rate of 0.01 with the stochastic gradient descent (SGD) optimizer to train the student. The momentum is set to 0.05 and the weight decay parameter is set to 0.003, which governs the regularization term of the student. A total number of 500 epochs have been trained on the student.

Table 3: Training and testing setups of different methods.

Method	Training		Testing	
	Physical Data	Simulation Data	Physical Data	Simulation Data
TPTP	√	×	√	×
TSTP	×	√	√	×
FT	√	√	√	×
CCSA	√	√	√	×
DaNN	√	√	√	×
RSM	√	√	√	×
Kriging	√	√	√	×
Ours	√	√	√	×

6 Evaluation

We perform a series of experiments to validate the efficiency of our method to identify good network configurations. We first evaluate the capability of our method to effectively improve the modeling accuracy and compare our method against seven baselines, which include five machine learning based methods: (i) Using the physical data for both training and testing (TPTP); (ii) Using the simulation data for training and the physical data for testing (TSTP) [75, 76]; (iii) Fine-tuning (FT) method [83]; (iv) CCSA: Unified deep supervised domain adaptation and generalization [58]; and (v) Domain adaptive neural network (DaNN) [28], and two non-machine learning methods: (vi) RSM method [9, 87] and (vii) Kriging method [11, 78]. Table 3 summarizes the training and testing data used by each method. All methods use L, B, and E as their input features. We then apply the network configurations selected by our method on our testbed and measure the network performance. We repeat our experiments with different network setups under various wireless conditions. Finally, we evaluate the effects of our method on closing the gap when employing different simulators and radio models.

6.1 Experimental Setup

As presented in Section 4.1, we configure six data flows on our testbed. On each data flow, sensor data is generated by

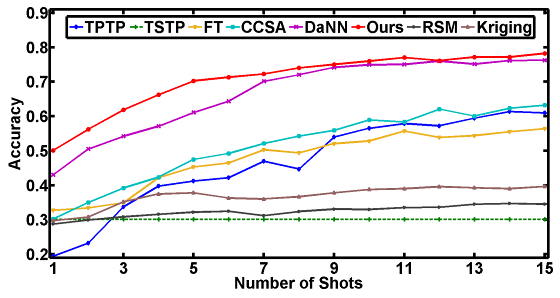


Figure 5: Modeling accuracy of our method and baselines when different number of shots of physical data are added into simulation data (3,960 data samples in total) for training. One shot includes 88 data samples (one data sample under each network configuration).

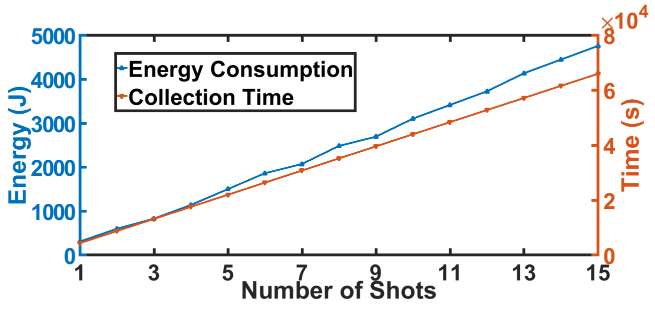


Figure 6: Time and energy consumption to collect different number of shots of data from a physical network. Using only physical data to train the model is infeasible due to unacceptable time and energy overhead.

the source node and forwarded to the access points (uplink) and then a corresponding control command is sent to the destination node (downlink). We calculate the latency, energy consumption, and reliability every 50*s*. We employ the same DNN architecture for the teacher and the student in our method with independent weights (see Section 5). Each neural network has 120 and 84 neurons in the first two hidden layers, and 88 neurons in the output layer. The weight β of MMD is 0.2 and the temperature T is 10. The learning rate is 0.01 with the SGD optimizer for the student. CCSA uses the cross-entropy loss and the semantic alignment loss between the source and target domains with the Siamese architecture. DaNN uses the standard classification loss and the MMD regularization for classification and domain adaptation. FT first uses the simulation data to train the initial model and then fine-tunes the neural network parameters to fit the target domain using a small amount of physical data. FT uses the learning rate of 0.001 to tune the parameters of the last layer in the DNN with the physical data. RSM and Kriging methods use simulation data and different amount of physical data to build RSM and Kriging models and use them to predict network configurations. Specifically, RSM is a black-box modeling technique and uses polynomial functions to approximate the model functions between the inputs and the outputs [9], while Kriging leverages spatial interpolation that uses complex mathematical formulas to estimate values at unknown points based on the values, which are already sampled [78].

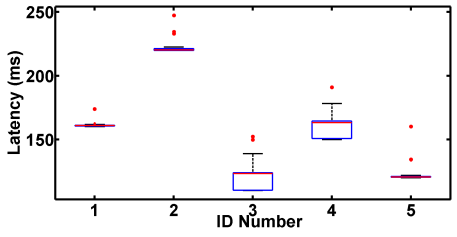
6.2 Performance of Our Method

We first evaluate the modeling accuracy of our method and compare its performance against seven baselines using the data traces presented in Section 4.1. 3,960 data samples from the simulation data are used for training under all methods except TPTP, which uses only the physical data for training. Figure 5 plots the modeling accuracy of all methods when different number of shots of physical data are added into the simulation data for training. As Figure 6 plots, collecting one shot of physical data (one data sample under each of 88 network configurations) takes 1.22 hours and consumes 310.61J of energy. Please note that TSTP uses only the simulation data for training (see Table 3) and provides the lowest accuracy (30.10%) due to the simulation-to-reality gap. The results clearly show that the model trained with the simulation data does not work well on the physical data. RSM and Kriging also provide poor performance with the maximum accuracy of 35.06% and 46.87%, respectively. Our method achieves the best performance. With only one shot of physical data (88 data samples), our method provides an accuracy of 50.12%. With four more shots of physical data, our method hits 70.24% accuracy. Using a small amount of physical data to provide a good model represents an important feature of our method because the data collection from a physical network is very time and energy consuming. As a comparison, without using the simulation data, TPTP provides only an accuracy of 19.39% and 41.21% at one shot and five shots, respectively. This highlights the importance of learning knowledge from simulations and transferring it to a physical network for network configurations.

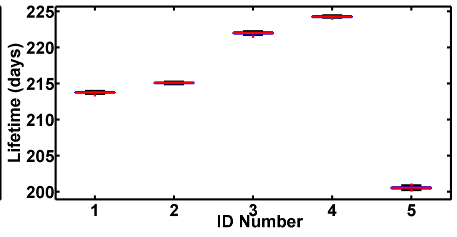
We also observe that the accuracy improves slowly from 70.24% to 78.25% when the number of shots increases from 5 to 15. However, collecting 10 more shots of physical data from a physical network takes a long time and consumes much energy. As Figure 6 plots, the collection of five shots of physical data takes 6.11*hours* and consumes 1,502.88J of energy, while collecting 15 shots take 18.33*hours* and consumes 4,758.70J of energy. The improvement on the modeling accuracy is largely shadowed by the significantly increased data collection overhead. Therefore, we use five shots in the rest of our evaluation. Figure 5 and 6 also show that only using physical data to train the model is inefficient. It takes 18.33*hours* to collect enough data from a physical network, which allows TPTP to provide an accuracy of 60.95%. By comparing the accuracy provided by our method and TPTP, we can clearly see the effectiveness of our method on reducing the data collection time for training good mod-

Table 4: Six example network configurations selected by our method and TSTP. Figure 7 and 8 show the network performance after applying the configurations selected by our method and TSTP on our testbed, respectively. Our method can meet all performance requirements. The performance requirements that TSTP fails to meet are highlighted.

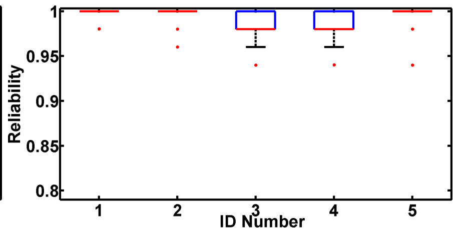
ID #	Input			Output (our method / TSTP)		
	Latency (*ms*)	Battery lifetime (*days*)	Reliability (%)	PRR threshold (%)	# of Channel	# of Tx Attempts
1	**170**	**210**	98	84 / 82	4 / 7	3 / 3
2	225	**214**	97	90 / 88	5 / 1	3 / 3
3	**130**	**220**	95	84 / 78	4 / 8	2 / 3
4	**165**	**224**	95	90 / 89	4 / 6	2 / 2
5	**130**	200	**98**	87 / 72	2 / 1	3 / 2



(a) Boxplot of latency.



(b) Boxplot of battery lifetime.



(c) Boxplot of reliability.

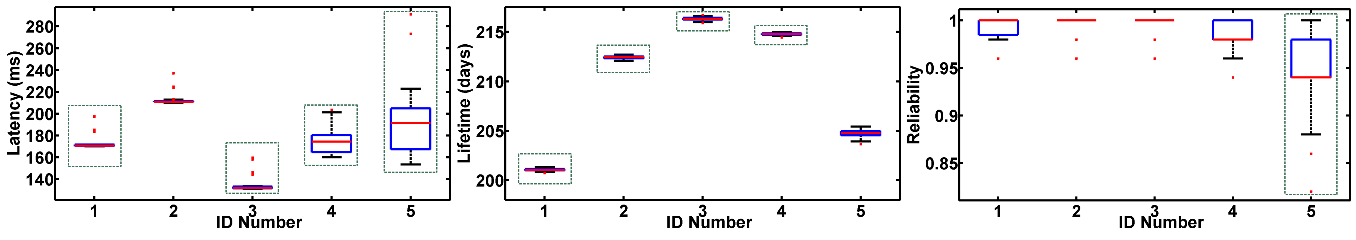
Figure 7: Network performance when employing the network configurations selected by our method (listed in Table 4). Central mark in box indicates median; bottom and top of box represent the 25th percentile (q_1) and 75th percentile (q_2); red dots indicate outliers ($x > q_2 + 1.5 * (q_2 - q_1)$ or $x < q1 - 1.5 * (q_2 - q_1)$); whiskers indicate the range that excludes outliers.

els for network configuration prediction. Our method consistently outperforms those two existing domain adaptation methods (DaNN and CCSA), which use the Siamese DNN model with different distance loss functions. For example, our method provides an accuracy of 70.24% when it uses five shots of physical data for training, while CCSA and DaNN provide 47.46% and 61.07% accuracy, respectively. The accuracy provided by FT increases from 32.73%, to 33.42%, and then to 56.40% when the number of shots increases from 1, to 2, and to 15 shots.

Our method can consistently outperform the baselines because it not only uses two different neural networks to learn two specific models for different but highly related domains with the soft labels but also employs the MMD regularization, while both DaNN and CCSA use same weights between the source and target domains for domain adaptation. Moreover, the distillation loss $\mathcal{L}_{dis}$ of our method provides a set of candidate network configurations for the student to choose and the student can quickly adapt to the target domain. The results also show that the domain-consistent loss, as a distribution distance measure, is effective for eliminating domain divergence between the source domain (simulated network) and the target domain (physical network). Our method also significantly outperforms FT. The low accuracy provided by FT shows that changing only the weight of the last layer in the DNN cannot produce a good adapted model.

We further validate the network configurations selected by our method on our testbed by examining the actual network performance. Specifically, we feed different network performance requirements to our method, employ the selected network configurations, and then measure the network performance. We repeat the experiments under each network configuration 108 times. Table 4 lists six example network configurations selected by our method and TSTP when facing different network performance requirements. Figure 7 plots the boxplots of latency, battery lifetime[6], and reliability when employing six network configurations selected by our method. As Figure 7 shows, our method always helps the network meet the network performance requirements posed by the application (listed in Table 4). For instance, the latency, battery lifetime, and reliability requirements are 170*ms*, 210*days*, and 98% in the first example ($ID = 1$). When employing the network configuration selected by our method (84% as PRR threshold, four channels, three transmission attempts for each packet), the network achieves a median latency of 161.00*ms*, a median battery lifetime of 213.76*days*, and a median reliability of 100%, which meet all given requirements. Similarly, the latency, battery lifetime, and reliability requirements are 165*ms*, 224*days*, and 95% in the fourth example ($ID = 4$). When employing the

[6]To compute the battery lifetime, we assume that each field device is powered by two Lithium Iron AA batteries with a total capacity of 42,700J. We compute the radio energy consumption based on the timestamps of radio activities and the radio's power consumption in each state according to the radio chip data sheet.



(a) Boxplot of latency. (b) Boxplot of battery lifetime. (c) Boxplot of reliability.

Figure 8: Network performance when employing the network configurations selected by TSTP (listed in Table 4). The dotted boxes highlight the network performance that fails to meet the requirements. Compared to Figure 7, our method always provides better network configurations than TSTR and help the network meet the application performance requirements.

network configuration selected by our method (90% as PRR threshold, four channels, two transmission attempts for each packet), the network achieves a median latency of 163.33*ms*, a median battery lifetime of 224.28*days*, and a median reliability of 98%, which meet all given requirements. Larger variations on latency are observed when the number of transmission attempts for each packet is small, which confirms the observations reported in our previous study [75, 76]

As a comparison, we also employ the network configurations selected by TSTP when facing the same network performance requirements. Table 4 lists the network configurations selected by TSTP and Figure 8 plots the resulting network performance. Due to the simulation-to-reality gap, the network configurations selected by TSTP cannot always meet all network performance requirements. The dotted boxes in Figure 8 highlight the network performance that fails to meet the application requirements listed in Table 4. For instance, the latency, battery lifetime, and reliability requirements are 130*ms*, 200*days*, and 98% in the fifth example ($ID = 5$). When employing the network configuration selected by TSTP (72% as PRR threshold, one channel, two transmission attempts for each packet), the network achieves a median latency of 191.40*ms*, a median battery lifetime of 204.74*days*, and a median reliability of 94.00%, which fail to meet the latency and reliability requirements.

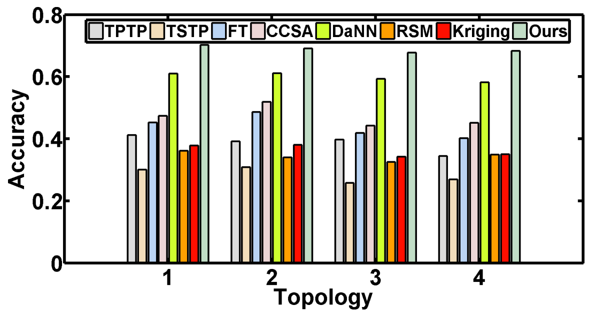


Figure 9: Accuracy comparison among different methods with different network topologies. All methods use five shots of physical data. Topology 1 is used for Figure 5.

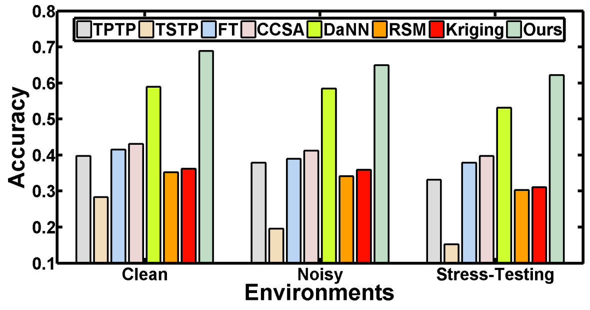


Figure 10: Accuracy comparison among different methods under different wireless conditions.

6.3 Performance with Different Network Topologies under Various Wireless Conditions

To examine the applicability of our method, we repeat our experiments with different network topologies under various wireless conditions. We first vary the number of data flows, the number of devices in the network, and the locations of source nodes, destination nodes, and access points and measure the performance of our method. Figure 9 plots the accuracy comparisons between our method and seven baselines under four example network topologies. Our method consistently provides the highest accuracy. For instance, our method achieves an accuracy of 67.09% under the third network topology, while CCSA and DaNN provide 44.23% and 59.37% accuracy, respectively. TPTP, TSTP, FT, RSM, and Kriging achieve 39.72%, 25.78%, 41.90%, 32.56%, and 34.26% accuracy, respectively. The results confirm the improvements presented in Section 6.2 and show our method can consistently outperform the state of the art.

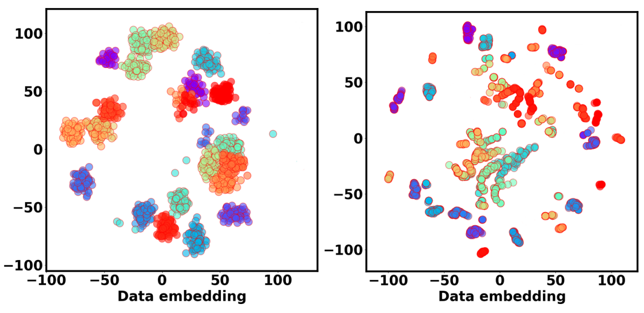
We also examine the performance of our method under different wireless conditions. We set up three jammers on our testbed (ID 116, 131, and 134 in Figure 1) and run Jamlab [10] on them to generate controlled WiFi interference with various strengths. We create three wireless conditions: a clean environment without controlled interference, a noisy environment with moderate controlled interference, and a stress-testing environment with strong controlled interference. We train the model again with different physical data under different wireless conditions. Figure 10 plots the modeling accuracy under three wireless conditions when employing our method and seven baselines. As Figure 10 shows, the accuracy provided by our method decreases from 68.89%, to 64.99%, and then to 62.20% when stronger interference is introduced. We observe similar trends when employing other methods.

This exposes a limitation of current wireless simulators, which cannot precisely simulate the effects of external interference and environmental dynamics. To better understand the physical data distribution, we visualize the data distribution of (L,B,E) collected from the physical data $(\mathcal{D}^p)$ using the t-Distributed Stochastic Neighbor Embedding (t-SNE) algorithm [88], a dimension reduction tool for data visualization. Figure 11 shows the network performance visualization provided by t-SNE where different colors stand for different network configurations. Figure 11(a) and Figure 11(b) plot the data distributions when the network operates with and without the presence of strong controlled interference, respectively. Please note that those two figures include the same amount of data points. Many data points in Figure 11(b) overlap each other. These larger variations, result from the interference, significantly increases the difficulty on transferring knowledge learned from simulations to a physical network. With the presence of interference, our method still consistently outperforms all baselines. For instance, in the stress-testing environment, our method provides an accuracy of 62.20%, while other methods provide up to 53.21% accuracy.

To illustrate the differences between physical data and simulation data, Figure 12 plots the reliability measured from the physical network and simulated by TOSSIM under four network configurations. Because of the simulation-to-reality gap, the measured reliability is different from the simulated one. More importantly, the variations of the measured reliability values are much larger than the simulated ones. Such differences highlight the important of our method, which effectively closes the gap and increases the accuracy of predicting a good network configuration that allows the network to meet performance requirements.

6.4 Effects of Different Losses

To study the effects of different losses on the performance of our method, we repeat the experiments by disabling one or two losses among the classification loss $\mathcal{L}_{cls}$, the distillation loss $\mathcal{L}_{dis}$, and the domain-consistent loss $\mathcal{L}_{mmd}$. We conduct our experiments using Topology 1 in Figure 9 in a clean environment. Figure 13 plots the accuracy when our method uses different combination of loss functions. As Figure 13 shows, our method with a single loss provides very low classification accuracy ($\mathcal{L}_{dis}$: 28.22%, $\mathcal{L}_{mmd}$: 26.81%, and $\mathcal{L}_{cls}$:41.21%). The accuracy is also very low (36.84%) when our method uses $\mathcal{L}_{dis}$ and $\mathcal{L}_{mmd}$ due to the critical need of the classification loss on the target domain. The accuracy increases to 64.60% when our method combines $\mathcal{L}_{cls}$ with $\mathcal{L}_{dis}$, because the distillation loss $\mathcal{L}_{dis}$ provides a set of candidate network configurations for the student to choose and the student can quickly adapt to the target domain by combining the knowledge distillation loss and classification loss. The accuracy further increases to 70.24% when our method uses all three losses. The results show that the domain-consistent loss, as a distribution distance measure, is effective for eliminating domain divergence between the source domain (simulated network) and the target domain (physical network).



(a) In the stress-testing environment. (b) In the clean environment.

Figure 11: Data visualization provided by t-SNE [88]. Larger variations are observed in stress-testing environment, which significantly increase the difficulty on transferring knowledge learned from simulations to a physical network.

6.5 Effects of Simulators and Radio Models

Finally, we study the effects of different simulators and radio models on the performance of our method. Unit Disk Graph Medium (UDGM) [55] and Directed Graph Radio Medium (DGRM) [55] are the two most popular radio models supported by Cooja [17, 65]. UDGM in Cooja uses the disk communication model and assumes that the receiver inside the communication range of the sender can successfully receive its packets with a constant PRR (i.e., 90%). DGRM in Cooja allows its user to specify the PRR of each link and use it together with a random number to determine whether each packet can be delivered successfully. Closest-fit pattern matching (CPM) in TOSSIM allows its user to input ambient noise traces and specify the gain value (propagation strength) between each pair of devices on every channel and then generates statistical models based on the CPM algorithm to compute the packet delivery ratio for each pair of devices [43]. We create DGRM-E by extending DGRM by allowing an user to specify different PRRs on different channels for each link, and then integrate it with TOSSIM. DISTANCE in NS-3 allow its user to specify the locations of all wireless devices and use the shadowing model to determine packet receptions [62]. OMNET++ allows its user

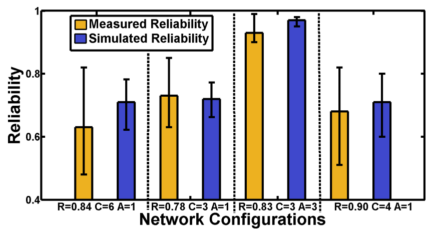


Figure 12: Reliability measured from the physical network and simulated by TOSSIM under four network configurations.

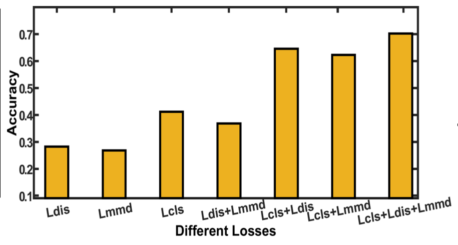


Figure 13: Accuracy when our method uses different loss functions.

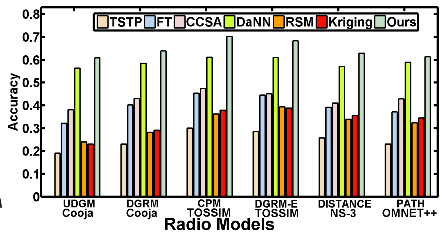


Figure 14: Accuracy comparison when using different simulators and radio models.

to specify device locations and background noise levels and uses the signal propagation model (path loss model) to compute the RSS values for packet reception prediction [64].

Figure 14 plots the accuracy of our method and our baselines when they use the simulation data generated from different simulators with various radio models. As Figure 14 shows, all methods achieve better performance when they use a more realistic model, which benefits from a smaller domain discrepancy. For instance, our method achieves 70.24% and 68.32% accuracy when it employs CPM and DEGRM-E in TOSSIM, respectively. The high accuracy results from the use of real-world noise or PRR traces in simulations. Our method provides a slightly lower accuracy (63.95%) when it uses DGRM in Cooja, which makes an unrealistic assumption that the PRRs of a link are the same on all channels. The worse performance (60.83%) appears when our method uses the simple disk model (UDGM) in Cooja. Similarly, the accuracy provided by TSTP decreases from 30.10% to 19.13% when it uses a less realistic radio model. More importantly, our method consistently provides the best performance and makes better use of more realistic simulations compared to other methods. The accuracy increases from 60.83% to 70.24% (a 9.41% increase) when our method uses CPM in TOSSIM instead of UDGM in Cooja, while the accuracy improvement offered by DaNN is 4.77% when making the same change.

The consistent low accuracy provided by TSTP shows that the simulation-to-reality gap is not tie up with a particular simulator or radio model. Although the theoretical models adopted by those simulators work satisfactorily in general, they cannot capture all real-world performance-related factors. For instance, the CPM approach in TOSSIM allows its user to input noise traces collected from a physical network and specify the gain value (propagation strength) between each pair of devices on every channel and then generates statistical models to predict packet receptions during simulations based on the CPM algorithm. Such an approach may introduce simulation inaccuracies because it has to use prerecorded noise traces and predefined gain values to simulate packet failures, and the probability-based prediction cannot precisely capture the effects of packet failures caused by extensive uncertainties, variations, and dynamics in real-world wireless deployments.

7 Conclusions

Over the past decade, WMNs have been widely used for industrial automation, military operations, smart energy, etc. Due to years of research, WMNs work satisfactorily most of the time. However, they are often difficult to configure as configuring a WMN is a complex process, involving theoretical computation, simulation, and field testing, among other tasks. Relying on field testing to identify good network configurations is impractical in many cases because running experiments on a physical network is often costly and time-consuming. Simulating the network performance under different network parameters provides distinct advantages when it comes to identifying a good network configuration, because a simulation can be set up in less time, introduce less overhead, and allow for different configurations to be tested under exactly the same condition. Unfortunately, out study shows that many network configurations identified in simulations cannot help physical networks achieve desirable performance because of the simulation-to-reality gap. To close the gap, We leverage a teacher-student deep neural network for efficient domain adaptation, which transfers network configuration knowledge learned from simulation to a physical network. Our method first uses the simulation data to learn a teacher neural network, which is then used to teach a student neural network to learn from a few shots of the physical data. Our experimental results show that our method consistently outperforms seven baselines and achieves a modeling accuracy of 70.24% with only 440 data samples collected from the physical network.

Acknowledgment

This work was supported in part by the National Science Foundation under grant CNS-2046538. The authors thank the anonymous reviewers and the shepherd Junchen Jiang at University of Chicago for their insightful comments.

References

[1] IEEE 802.15.4e WPAN Task Group. http://www.ieee802.org/15/pub/TG4e.html.

[2] E. Altman, T.Boulogne, R. El-Azouzi, T. Jiménez, and L. Wynterc. A Survey on Networking Games in Telecommunications. *Computers and Operations Research*, 33(2):286–311, 2006.

[3] Irvan B Arief-Ang, Flora D Salim, and Margaret Hamilton. DA-HOC: Semi-Supervised Domain Adaptation for Room Occupancy Prediction using CO2 Sensor Data. In *ACM International Conference on Systems for Energy-Efficient Built Environments (BuildSys)*, 2017.

[4] Taichi Asami, Ryo Masumura, Yoshikazu Yamaguchi, Hirokazu Masataki, and Yushi Aono. Domain Adaptation of DNN Acoustic Models Using Knowledge Distillation. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2017.

[5] Sachin Ashok, Sai Surya Duvvuri, Nagarajan Natarajan, Venkata N. Padmanabhan, Sundararajan Sellamanickam, and Johannes Gehrke. iBox: Internet in a Box. In *ACM Workshop on Hot Topics in Networks (HotNets)*, 2020.

[6] Nouha Baccour, Anis Koubâa, Luca Mottola, Marco Antonio Zúñiga, Habib Youssef, Carlo Alberto Boano, and Mário Alves. Radio Link Quality Estimation in Wireless Sensor Networks: A Survey. *ACM Transaction on Sensor Network*, 8(4), 2012.

[7] Mihovil Bartulovic, Junchen Jiang, Sivaraman Balakrishnan, Vyas Sekar, and Bruno Sinopoli. Biases in Data-Driven Networking, and What to Do About Them. In *ACM Workshop on Hot Topics in Networks (HotNets)*, 2017.

[8] K. Benkic, M. Malajner, P. Planinsic, and Z. Cucej. Using RSSI value for distance estimation in wireless sensor networks based on ZigBee. In *International Conference on Systems, Signals and Image Processing (IWSSIP)*, 2008.

[9] Marcos Almeida Bezerra, Ricardo Erthal Santelli, Eliane Padua Oliveira, Leonardo Silveira Villar, and Luciane Amélia Escaleira. Response Surface Methodology (RSM) as a Tool for Optimization in Analytical Chemistry. *Talanta*, 76(5):965 – 977, 2008.

[10] Carlo Alberto Boano, Thiemo Voigt, Claro Noda, Kay Romer, and Marco Zuniga. JamLab: Augmenting Sensornet Testbeds with Realistic and Controlled Interference Generation. In *ACM/IEEE International Symposium on Information Processing in Sensor Networks (IPSN)*, 2011.

[11] Gabriele Boccolini, Gustavo Hernández-Peñaloza, and Baltasar Beferull-Lozano. Wireless Sensor Network for Spectrum Cartography based on Kriging Interpolation. In *International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*, 2012.

[12] Konstantinos Bousmalis, Alex Irpan, Paul Wohlhart, Yunfei Bai, Matthew Kelcey, Mrinal Kalakrishnan, Laura Downs, Julian Ibarz, Peter Pastor Sampedro, Kurt Konolige, Sergey Levine, and Vincent Vanhoucke. Using Simulation and Domain Adaptation to Improve Efficiency of Deep Robotic Grasping. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2018.

[13] Konstantinos Bousmalis, George Trigeorgis, Nathan Silberman, Dilip Krishnan, and Dumitru Erhan. Domain Separation Networks. In *Advances in Neural Information Processing Systems (NIPS)*. Curran Associates Inc., 2016.

[14] Xianghui Cao, Lu Liu, Yu Cheng, and Xuemin Shen. Towards Energy-Efficient Wireless Networking in the Big Data Era: A Survey. *IEEE Communications Surveys & Tutorials*, 20(1):303–332, 2018.

[15] C. Caruso and F. Quarta. Interpolation Methods Comparison. *Computers and Mathematics with Applications*, 35(12):109 – 126, 1998.

[16] Xuewen Chen and Jong cheol Jeong. Enhanced recursive feature elimination. In *Sixth International Conference on Machine Learning and Applications (ICMLA)*, 2007.

[17] Source Code of Cooja. https://github.com/contiki-os/contiki/wiki/An-Introduction-to-Cooja.

[18] H. Dakdouk, E. Tarazona, R. Alami, R. Féraud, G. Z. Papadopoulos, and P. Maillée. Reinforcement Learning Techniques for Optimized Channel Hopping in IEEE 802.15.4-TSCH Networks. In *ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWIM)*, 2018.

[19] Jeff Donahue, Philipp Krähenbühl, and Trevor Darrell. Adversarial Feature Learning. *CoRR*, abs/1605.09782, 2017.

[20] Wei Dong, Chun Chen, Xue Liu, Yuan He, Yunhao Liu, Jiajun Bu, and Xianghua Xu. Dynamic Packet Length Control in Wireless Sensor Networks. *IEEE Transactions on Wireless Communications*, 13(3):1172–1181, 2014.

[21] Gintare Karolina Dziugaite, Daniel M. Roy, and Zoubin Ghahramani. Training Generative Neural Networks via Maximum Mean Discrepancy Optimization. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2015.

[22] Emerson Process Management. https://www.emerson.com/en-us/automation-solutions.

[23] System Engineering Guidelines IEC 62591 WirelessHART by Emerson Process Management. http://www2.emersonprocess.com/siteadmincenter/PM%20Central%20Web%20Documents/EMR%5fWirelessHART%5fSysEngGuide.pdf.

[24] WirelessHART Networks Deployed by Emerson Process Management. https://www.emerson.com/en-us/expertise/automation/industrial-internet-things/pervasive-sensing-solutions/wireless-technology.

[25] Songwei Fu, Yan Zhang, Yuming Jiang, Chengchen Hu, Chia-Yen Shih, and Pedro Jose Marron. Experimental Study for Multi-layer Parameter Configuration of WSN Links. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2015.

[26] Yaroslav Ganin and Victor Lempitsky. Unsupervised Domain Adaptation by Backpropagation. In *International Conference on International Conference on Machine Learning (ICML)*, 2015.

[27] Yaroslav Ganin, Evgeniya Ustinova, Hana Ajakan, Pascal Germain, Hugo Larochelle, François Laviolette, Mario March, and Victor Lempitsky. Domain-Adversarial Training of Neural Networks. *Journal of Machine Learning Research*, 17(59):1–35, 2016.

[28] Muhammad Ghifary, W. Bastiaan Kleijn, and Mengjie Zhang. Domain Adaptive Neural Networks for Object Recognition. In *Pacific Rim Conference on Artificial Intelligence Trends in Artificial Intelligence (PRICAI)*, 2014.

[29] Arthur Gretton, Alex Smola, Jiayuan Huang, Marcel Schmittfull, Karsten Borgwardt, and Bernhard Schölkopf. *Covariate Shift and Local Learning by Distribution Matching*, pages 131–160. MIT Press, 2009.

[30] Dolvara Gunatilaka, Mo Sha, and Chenyang Lu. Impacts of Channel Selection on Industrial Wireless Sensor-Actuator Networks. In *IEEE Conference on Computer Communications (INFOCOM)*, 2017.

[31] HART Foundation (Now FieldComm Group). http://www.hartcomm.org/.

[32] Hongli He, Hangguan Shan, Aiping Huang, Qiang Ye, and Weihua Zhuang. Reinforcement Learning-Based Computing and Transmission Scheduling for LTE-U-Enabled IoT. In *IEEE Global Communications Conference (GLOBECOM)*, 2018.

[33] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[34] Geoffrey Hinton, Oriol Vinyals, and Jeffrey Dean. Distilling the Knowledge in a Neural Network. In *Deep Learning and Representation Learning Workshop (NIPS)*, 2015.

[35] Gao Huang, Zhuang Liu, Kilian Q Weinberger, and Laurens van der Maaten. Densely Connected Convolutional Networks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[36] Liang Huang, Suzhi Bi, and Ying Jun Zhang. Deep Reinforcement Learning for Online Computation Offloading in Wireless Powered Mobile-Edge Computing Networks. *IEEE Transactions on Mobile Computing*, Early Access, 2020.

[37] International Electrotechnical Commission (IEC). https://www.iec.ch/.

[38] International Society of Automation (ISA). https://www.isa.org/.

[39] Alan Jovic, Karla Brkić, and Nikola Bogunović. A Review of Feature Selection Methods with Applications. In *38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 2015.

[40] Charles W. Kazer, Jo ao Sedoc, Kelvin K.W. Ng, Vincent Liu, and Lyle H. Ungar. Fast Network Simulation Through Approximation or: How Blind Men Can Describe Elephants. In *ACM Workshop on Hot Topics in Networks (HotNets)*, 2018.

[41] Diederik Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In *International Conference on Learning Representations (ICLR)*, 2014.

[42] D.Praveen Kumar, Tarachand Amgoth, and Chandra Sekhara Rao Annavarapu. Machine Learning Algorithms for Wireless Sensor Networks: A Survey. *Information Fusion*, 49:1–25, 2019.

[43] HyungJune Lee, Alberto Cerpa, and Philip Levis. Improving Wireless Simulation through Noise Modeling. In *International Conference on Information Processing in Sensor Networks (IPSN)*, 2007.

[44] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. In *International Conference on Embedded Networked Sensor Systems (Sensys)*, 2003.

[45] Feng Li, Kwok-Yan Lam, Zhengguo Sheng, Xinggan Zhang, Kanglian Zhao, and Li Wang. Q-Learning-Based Dynamic Spectrum Access in Cognitive Industrial Internet of Things. *Mobile Networks and Applications*, 23:10, 2018.

[46] Xin Li, Qiuyuan Huang, and Dapeng Wu. Distributed large-scale co-simulation for iot-aided smart grid control. *IEEE Access*, PP:1–1, 2017.

[47] Chi Harold Liu, Qiuxia Lin, and Shilin Wen. Blockchain-Enabled Data Collection and Sharing for Industrial IoT With Deep Reinforcement Learning. *IEEE Transactions on Industrial Informatics*, 15(6):3516–3526, 2019.

[48] Jane Liu. *Real-Time Systems*. Prentice Hall PTR, USA, 1st edition, 2000.

[49] Yongkang Liu, Richard Candell, Kang Lee, and Nader Moayeri. A Simulation Framework for Industrial Wireless Networks and Process Control Systems. In *IEEE World Conference on Factory Communication Systems (WFCS)*, 2016.

[50] Gilles Louppe, Louis Wehenkel, Antonio Sutera, and Pierre Geurts. Understanding Variable Importances in Forests of Randomized Trees. In *International Conference on Neural Information Processing Systems (NeurIPS)*, 2013.

[51] Chenyang Lu, Abusayeed Saifullah, Bo Li, Mo Sha, Humberto Gonzalez, Dolvara Gunatilaka, Chengjie Wu, Lanshun Nie, and Yixin Chen. Real-Time Wireless Sensor-Actuator Networks for Industrial Cyber-Physical Systems. In *Proceedings of the IEEE, Special Issue on Industrial Cyber Physical Systems (ICPSs)*, 2016.

[52] Zhi Quan Luo and Wei Yu. An Introduction to Convex Optimization for Communications and Signal Processing. *IEEE Journal on Selected Areas in Communications*, 24(8):1426–1438, 2006.

[53] Nguyen Cong Luong, Dinh Thai Hoang, Shimin Gong, Dusit Niyato, Ping Wang, Ying-Chang Liang, and Dong In Kim. Applications of Deep Reinforcement Learning in Communications and Networking: A Survey. *IEEE Communications Surveys and Tutorials*, 21(4):3133–3174, 2019.

[54] Qian Mao, Fei Hu, and Qi Hao. Deep Learning for Intelligent Wireless Networks: A Comprehensive Survey. *IEEE Communications Surveys and Tutorials*, 20(4):2595–2621, 2018.

[55] Tayyab Mehmood. Cooja Network Simulator: Exploring the Infinite Possible Ways to Compute the Performance Metrics of IoT Based Smart Devices to Understand the Working of IoT Based Compression & Routing Protocols, 2017.

[56] Fan Meng, Peng Chen, Lenan Wu, and Julian Cheng. Power Allocation in Multi-User Cellular Networks: Deep Reinforcement Learning Approaches. *IEEE Transactions on Wireless Communications*, Early Access, 2020.

[57] Meshdynamics. https://www.meshdynamics.com/index.html.

[58] Saeid Motiian, Marco Piccirilli, Donald A. Adjeroh, and Gianfranco Doretto. Unified Deep Supervised Domain Adaptation and Generalization. In *IEEE International Conference on Computer Vision (ICCV)*, 2017.

[59] Oshri Naparstek and Kobi Cohen. Deep Multi-User Reinforcement Learning for Distributed Dynamic Spectrum Access. *IEEE Transactions on Wireless Communications*, 18(11):310–323, 2019.

[60] Yasar Sinan Nasir and Dongning Guo. Multi-Agent Deep Reinforcement Learning for Dynamic Power Allocation in Wireless Networks. *IEEE Journal on Selected Areas in Communications*, 37(10):2239–2250, 2019.

[61] NS-3 Network Simulator. https://www.nsnam.org/.

[62] NS-3 Shadowing Model. https://www.nsnam.org/docs/release/3.10/doxygen/classns3_1_1_shadowing_loss_model.html.

[63] Source Code of OMNeT++. https://github.com/omnetpp/omnetpp.

[64] OMNeT++ INET Framework and Transmission Medium. https://inet.omnetpp.org/docs/users-guide/ch-transmission-medium.html.

[65] Fredrik Osterlind, Adam Dunkels, Joakim Eriksson, Niclas Finne, and Thiemo Voigt. Cross-Level Sensor Network Simulation with Cooja. In *IEEE Conference on Local Computer Networks (LCN)*, 2006.

[66] Daniel W. Otter, Julian R. Medina, and Jugal K. Kalita. A Survey of the Usages of Deep Learning for Natural Language Processing. *IEEE Transactions on Neural Networks and Learning Systems*, Early Access, 2020.

[67] Stephen S. Oyewobi, Gerhard P. Hancke, Adnan M. Abu-Mahfouz, and Adeiza J. Onumanyi. An Effective Spectrum Handoff Based on Reinforcement Learning for Target Channel Selection in the Industrial Internet of Things. *Sensors*, 19(6):1–21, 2019.

[68] Sinno Jialin Pan, Ivor W. Tsang, James T. Kwok, and Qiang Yang. Domain Adaptation via Transfer Component Analysis. *IEEE Transactions on Neural Networks*, 22(2):199–210, 2011.

[69] Vishal M Patel, Raghuraman Gopalan, Ruonan Li, and Rama Chellappa. Visual Domain Adaptation: A Survey of Recent Advances. *IEEE Signal Processing Magazine*, 32(3):53–69, 2015.

[70] Yang Peng, Zi Li, Daji Qiao, and Wensheng Zhang. I2C: A Holistic Approach to Prolong the Sensor Network Lifetime. In *IEEE International Conference on Computer Communications (INFOCOM)*, pages 2670–2678, 2013.

[71] Marius-Constantin Popescu, Valentina E. Balas, Liliana Perescu-Popescu, and Nikos Mastorakis. Multilayer Perceptron and Neural Networks. *WSEAS Transactions on Circuits and Systems*, 8(7):579–588, 2009.

[72] Xiaoyu Qiu, Luobin Liu, Wuhui Chen, Zicong Hong, and Zibin Zheng. Q-Learning-Based Dynamic Spectrum Access in Cognitive Industrial Internet of Things. *IEEE Transactions on Vehicular Technology*, 68(8):8050–8062, 2019.

[73] Mauricio G.C. Resende and Panos Pardalos. *Handbook of Optimization in Telecommunications*. Springer, 2006.

[74] Noga H. Rotman, Michael Schapira, and Aviv Tamar. Online Safety Assurance for Learning-Augmented Systems. In *ACM Workshop on Hot Topics in Networks (HotNets)*, 2020.

[75] Junyang Shi and Mo Sha. Parameter Self-Configuration and Self-Adaptation in Industrial Wireless Sensor-Actuator Networks. In *IEEE International Conference on Computer Communications (INFOCOM)*, 2019.

[76] Junyang Shi and Mo Sha. Parameter Self-Adaptation for Industrial Wireless Sensor-Actuator Networks. *ACM Transactions on Internet Technology*, 20(3), 2020.

[77] Ashish Shrivastava, Tomas Pfister, Oncel Tuzel, Josh Susskind, Wenda Wang, and Russell Webb. Learning from Simulated and Unsupervised Images through Adversarial Training. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[78] Timothy Simpson, Timothy Mauery, John Korte, and Farrokh Mistree. Kriging Models for Global Approximation in Simulation-Based Multidisciplinary Design Optimization. In *AIAA Journal*, 2001.

[79] Mariusz Slabicki, Gopika Premsankar, and Mario Di Francesco. Adaptive Configuration of LoRa Networks for Dense IoT Deployments. In *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2018.

[80] Zigbee Smart Energy. https://zigbeealliance.org/zigbee_products/smart-energy-monitor-2/.

[81] TelosB Datasheet Provided by Memsic Incorporation. http://www.memsic.com/userfiles/files/Datasheets/WSN/telosb_datasheet.pdf.

[82] Shun Tobiyama, Bo Hu, Kazunori Kamiya, and Kenji Takahashi. Large-Scale Network-Traffic-Identification Method with Domain Adaptation. In *Companion Proceedings of the Web Conference (WWW)*, 2020.

[83] Edna Chebet Too, Li Yujian, Sam Njuki, and Liu Yingchun. A Comparative Study of Fine-tuning Deep Learning Models for Plant Disease Identification. *Computers and Electronics in Agriculture*, 161:272 – 279, 2019.

[84] Source Code of TOSSIM. https://github.com/tinyos/tinyos-main/tree/master/tos/lib/tossim.

[85] IEEE 802.15.4e Time-Slotted Channel Hopping (TSCH). https://tools.ietf.org/html/rfc7554.

[86] Eric Tzeng, Judy Hoffman, Ning Zhang, Kate Saenko, and Trevor Darrell. Deep Domain Confusion: Maximizing for Domain Invariance. *ArXiv*, abs/1412.3474, 2014.

[87] K. K. Vadde, V. R. Syrotiuk, and D. C. Montgomery. Optimizing Protocol Interaction Using Response Surface Methodology. *IEEE Transactions on Mobile Computing*, 5(6):627–639, 2006.

[88] Laurens Van der Maaten and Geoffrey Hinton. Visualizing Data using t-SNE . *Journal of Machine Learning Research*, 9:2579–2605, 2008.

[89] András Varga and Rudolf Hornig. An Overview of the OMNeT++ Simulation Environment. In *International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops (ICST)*, 2008.

[90] Jiliang Wang, Zhichao Cao, Xufei Mao, and Yunhao Liu. Sleep in the Dins: Insomnia Therapy for Duty-cycled Sensor Networks. In *IEEE Conference on Computer Communications (INFOCOM)*, 2014.

[91] Jingjing Wang, Chunxiao Jiang, Kai Zhang, Xiangwang Hou, Yong Ren, and Yi Qian. Distributed Q-Learning Aided Heterogeneous Network Association for Energy-Efficient IIoT. *IEEE Transactions on Industrial Informatics*, 16(4):2756–2764, 2020.

[92] Mei Wang and Weihong Deng. Deep Visual Domain Adaptation: A Survey. *Neurocomputing*, 312(27):135–153, 2018.

[93] Shangxing Wang, Hanpeng Liu, Pedro Henrique Gomes, and Bhaskar Krishnamachari. Deep Reinforcement Learning for Dynamic Multichannel Access in Wireless Networks. *IEEE Transactions on Cognitive Communications and Networking*, 4(2):257–265, 2018.

[94] WCPS Simulator. http://wsn.cse.wustl.edu/index.php/WCPS:_Wireless_Cyber-Physical_Simulator.

[95] WirelessHART for Industrial Automation. https://fieldcommgroup.org/technologies/hart.

[96] Hansong Xu, Xing Liu, Wei Yu, David Griffith, and Nada Golmie. Reinforcement Learning-Based Control and Networking Co-Design for Industrial Internet of Things. *IEEE Journal on Selected Areas in Communications*, 38(5):885–898, 2020.

[97] Francis Y. Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and Keith Winstein. Learning in situ: a randomized experiment in video streaming. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[98] Helin Yang, Arokiaswami Alphones, Wen-De Zhong, Chen Chen, and Xianzhong Xie. Learning-Based Energy-Efficient Resource Management by Heterogeneous RF/VLC for Ultra-Reliable Low-Latency Industrial IoT Networks. *IEEE Transactions on Industrial Informatics*, 16(8):5565–5576, 2020.

[99] Kok-Lim Alvin Yau, Peter Komisarczuk, and Paul D. Teal. Reinforcement Learning for Context Awareness and Intelligence in Wireless Networks: Review, New Features and Open Issues. *IEEE Journal on Selected Areas in Communications*, 37(10):2239–2250, 2019.

[100] Yiding Yu, Taotao Wang, and Soung Chang Liew. Deep-Reinforcement Learning Multiple Access for Heterogeneous Wireless Networks. *IEEE Journal on Selected Areas in Communications*, 37(6):1277–1290, 2019.

[101] Chaoyun Zhang, Paul Patras, and Hamed Haddadi. Deep Learning in Mobile and Wireless Networking: A Survey. *IEEE Communications Surveys & Tutorials*, 21(3):2224–2287, 2019.

[102] Tianyu Zhang and Omid Ardakanian. A Domain Adaptation Technique for Fine-Grained Occupancy Estimation in Commercial Buildings. In *ACM/IEEE International Conference on Internet of Things Design and Implementation (IoTDI)*, 2019.

[103] Nan Zhao, Ying-Chang Liang, Dusit Niyato, Yiyang Pei, Minghu Wu, and Yunhao Jiang. Deep Reinforcement Learning for User Association and Resource Allocation in Heterogeneous Cellular Networks. *IEEE Transactions on Wireless Communications*, 18(11):5141–5152, 2019.

[104] Zigbee Alliance. https://zigbeealliance.org/.

[105] Marco Zimmerling, Federico Ferrari, Luca Mottola, Thiemo Voigt, and Lothar Thiele. PTunes: Runtime Parameter Adaptation for Low-Power MAC Protocols. In *International Conference on Information Processing in Sensor Networks (IPSN)*, 2012.

Practical Null Steering in Millimeter Wave Networks

Sohrab Madani*, Suraj Jog*, Jesus O. Lacruz†, Joerg Widmer†, Haitham Hassanieh*
*University of Illinois at Urbana Champaign, †IMDEA Networks

Abstract – Millimeter wave (mmWave) is playing a central role in pushing the performance and scalability of wireless networks by offering huge bandwidth and extremely high data rates. Millimeter wave radios use phased array technology to modify the antenna beam pattern and focus their power towards the transmitter or receiver. In this paper, we explore the practicality of modifying the beam pattern to suppress interference by creating nulls, i.e. directions in the beam pattern where almost no power is received. Creating nulls in practice, however, is challenging due to the fact that practical mmWave phased arrays offer very limited control in setting the parameters of the beam pattern and suffer from hardware imperfections which prevent us from nulling interference.

We introduce Nulli-Fi, the first practical mmWave null steering system. Nulli-Fi combines a novel theoretically optimal algorithm that accounts for limitations in practical phased arrays with a discrete optimization framework that overcomes hardware imperfections. Nulli-Fi also introduces a fast null steering protocol to quickly null new unforeseen interferers. We implement and extensively evaluate Nulli-Fi using commercial off-the-shelf 60 GHz mmWave radios with 16-element phased arrays transmitting IEEE 802.11ad packets [33] . Our results show that Nulli-Fi can create nulls that reduce interference by up to 18 dB even when the phased array offers only 4 bits of control. In a network with 10 links (20 nodes), Nulli-Fi's ability to null interference enables 2.68× higher total network throughput compared to recent past work.

1 Introduction

Millimeter wave (mmWave) networks introduced a major leap in data rates and scalability for 5G cellular networks, next generation wireless LANs, and IoT devices [10, 41, 46]. At the heart of millimeter wave technology are phased arrays which can focus the power of the antenna beam pattern in real-time towards the client to compensate for the large attenuation of mmWave signals. At mmWave frequencies ($\geq$ 24 GHz), phased arrays can fit many antennas into a small area due to the mm-scale wavelength of the signal [63], enabling very narrow directional beams as shown in Fig. 1. Ideally, using narrow beams would shield a mmWave device from interference outside the main direction (main lobe) of its beam. However, phased arrays suffer from side-lobe leakage as shown in Fig. 1(a). Hence, they still receive interfering signals even if these signals come from directions outside the main lobe. Past work has shown that side-lobes can lead to a significant amount of interference which can degrade the data rate and in dense networks reduce the total network throughput to half (by up to 18 Gbps) [14, 27, 44, 55, 61].

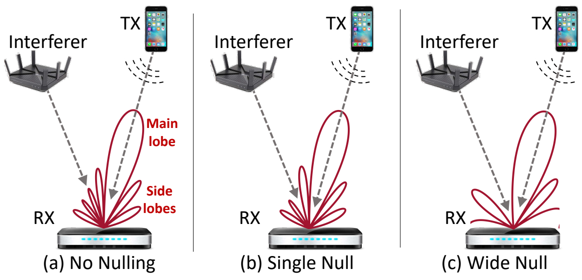


Figure 1: Directional beams in mmWave networks

To address the above problem, we leverage the fact that phased array beam patterns exhibit nulls, directions in the beam pattern where the transmitted or received power is suppressed as shown in Fig. 2(b). Thus, we can substantially reduce interference by having a null in the direction of the interferer. However, simply shifting the beam pattern to align the null with the interferer can misalign the main lobe and lead to worse performance as we show in section 6. Hence, we must create a new beam pattern to introduce a null in the direction of the interferer while preserving the alignment of the main lobe as shown in Fig. 1(b). This problem is commonly referred to as null steering.

Past mmWave systems research has mainly focused on beam alignment and steering [13, 17, 20, 40, 51, 56, 65], i.e. creating and steering the main lobe of the beam. Creating and steering nulls, however, while ensuring the main lobe is preserved is significantly harder. To better understand why, consider the phased array diagram shown in Fig. 2(a). The beam pattern of the array is created by modifying complex weights applied to each antenna element of the phased array. These complex weights alter the magnitude and phase of the signal received on each antenna. We can adjust these weights to align signals on the antennas coming from a certain direction to sum constructively creating a main lobe as shown in Fig. 2(d). In contrast, to create a null, we must ensure that the signals sum up destructively to cancel each other.

Setting the complex weights to ensure the signals from the direction of the interferer cancel each other while signals from the main lobe direction continue to sum up constructively is challenging in practice for several reasons. First, commercial mmWave phased arrays only allow us to change the phase of the complex weight but do not offer any control over the amplitude [11, 64]. While it is sufficient to rotate the phase of the signals to ensure they sum up constructively, it is hard to ensure signals cancel each other without modifying their

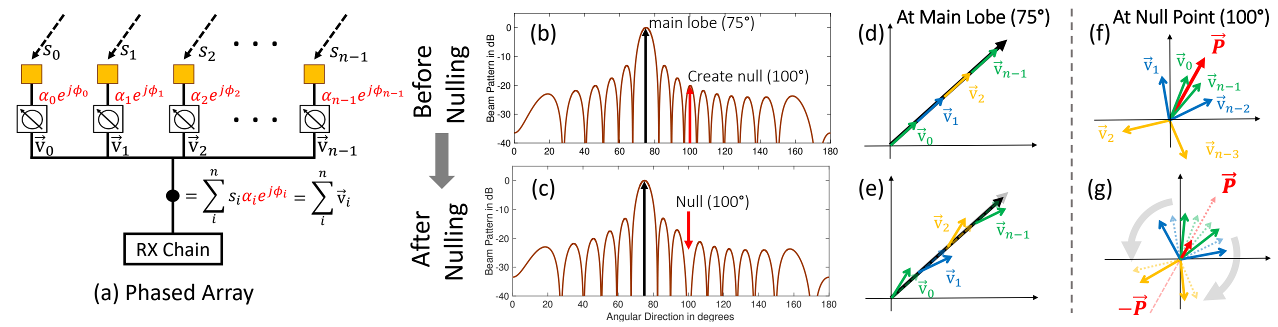


Figure 2: (a) phased arrays weight combination, (b-g) Nulli-Fi's Nulling Algorithm

amplitude. This is further complicated by the fact that phase control in practical arrays is highly quantized using at most 2 to 5 bits to control the phase shifts.[1] Moreover, practical phased arrays suffer from hardware imperfections [38] which have little impact on the main lobe but can limit the ability to null [38, 43]. For example, in an array with 8 antennas, if the phase on one antenna is off by 5°, the received signal along the main lobe degrades by only 0.004 dB whereas the interference signal along a null increases by 10 dB as we describe in more details in section 4.1.

Furthermore, unlike the main lobe which is naturally wide and, hence, can tolerate small errors in the direction of communication, nulls are narrow as shown in Fig. 2. As a result, any small error in the direction of the interferer will misalign the null and prevent us from effectively eliminating interference. To address this, we must create wider nulls rather than point nulls as shown in Fig. 1(c). In addition, in dense networks, we would need to null multiple different directions to account for multiple interferers or multipath reflections. Creating multiple nulls and wider nulls impose even more requirements that are hard to meet given the constraints and hardware imperfections of practical phased arrays.

Due to the above challenges, past work on null steering remains simulation based [5, 32, 53] and has not been implemented on practical mmWave phased arrays. Furthermore, most past work focuses on creating a single point null and none of the past work accounts for hardware imperfections.

This paper presents Nulli-Fi, the first practical mmWave null steering system that is able to null interference on commercial off-the-shelf phased arrays while preserving the main lobe. Nulli-Fi addresses the above practical challenges by combining a new theoretically optimal algorithm that accounts for limitations in practical phased arrays with a novel discrete optimization framework that overcomes hardware imperfections and enables multiple and wider nulls.

Nulli-Fi's optimal algorithm is able to create a single null within the constraints of practical phased arrays. To understand how this algorithm works, consider the example shown in Fig. 2. The goal is to have the main lobe at 75° and create a null at 100°. Each vector in Fig. 2(d)-(g) represents the signal received on a given antenna element. For signals received along 75°, Nulli-Fi sets the phase shifters to rotate the phase of each of these signals to sum up constructively as shown in Fig. 2(d). For signals received along 100°, the signals will have different phases and the vectors will sum up to some vector $\vec{P}$ as shown in Fig. 2(f). Our goal is to rotate these vectors by changing the phase on the phase shifters in order to null $\vec{P}$ while preserving the main lobe. To do so, Nulli-Fi restricts further phase-shifts on each antenna to a limited range. For example, if we restrict it to $\pm 15°$ on all antennas, the main lobe does not change by more than 0.3 dB, as shown in Fig. 2(e). Nulli-Fi then leverages the insight that the vectors are symmetric around $\vec{P}$ as shown in Fig 2(f).[2] By rotating pairs of symmetric vectors towards $-\vec{P}$, as shown in Fig 2(g), we reduce the amplitude of $\vec{P}$. We iteratively rotate the vectors until we null $\vec{P}$ or achieve the best possible reduction which we prove is optimal given the restrictions on the phase shifts.

The above algorithm provides a simple, optimal way to create nulls under limited phase control but it does not account for hardware imperfections, nor can it create nulls in more than one direction. To address this, we introduce a discrete optimization framework customized to null steering. The framework is inspired by genetic algorithms which have proven effective in discrete optimizations [19, 60]. However, genetic algorithms are very slow and can take thousands of iterations to converge [60] which prevents practical realtime null steering as we discuss in detail in section 7. Like many other optimization techniques, the initialization and stopping criterion are among the most contributing factors to the algorithm's convergence speed [48, 62]. To address this, Nulli-Fi uses the solution to its optimal algorithm. First, it initializes the optimization framework using the solution from the above algorithm which gives Nulli-Fi a significant head-start and helps it converge faster as we show in section 6. Second, since Nulli-Fi's algorithm is optimal, it can serve as a stopping criterion to the optimization framework (i.e., the algorithm knows if it has reached a reasonable solution). Combining the two methods gives Nulli-Fi a powerful framework that is both fast and is able to handle hardware imperfections.

Finally, to enable a practical system, Nulli-Fi develops a

[1]For example, 802.11ad compliant consumer-grade devices use only 2 bit phase shifters, i.e. we can set the phase only to 0°, 90°, 180°, or 270°.

[2]We prove this in lemma 4.2 to be true for any directions of the main lobe and the null for even number of antennas.

Past Work	Analog Beamforming?	Phase only?	Discrete Phase?	Implemented?	HW Imperfections?	Wide Nulls?
[49]	✗	✗	✗	✗	✗	✗
[52]	✗	✓	✗	✗	✗	✗
[6, 25, 29, 53]	✓	✗	✗	✗	✗	✓
[5, 9, 30, 58]	✓	✗	✗	✗	✗	✗
[32]	✓	✗	✓ (1° Res.)	✗	✗	✗
[12, 21, 39, 50, 54, 57]	✓	✓	✗	✗	✗	✗
[22, 23, 35]	✓	✓	✓ (6 bits)	✗	✗	✗
[8]	✓	✓	✓ (9 bits)	✓ (at 4.5 GHz)	✗	✗
[15]	✓	✓	✗	✓ (at 2.5 GHz)	✗	✗
Nulli-Fi	✓	✓	✓ (2-4 bits)	✓ (at 60 GHz)	✓	✓

Table 1: Summary of Related Work on Phased Array Nulling

fast null steering protocol that is able to quickly find the direction in which to create a null whenever a new unforeseen interferer appears. The protocol leverages the intuition that the interferer direction is more likely to be at the large side-lobes shown in Fig. 2(b). Hence, instead of searching all possible directions, Nulli-Fi starts with a large side-lobe where it creates a wide null and iterates through the side-lobes until the interferer is nulled.

We have implemented and extensively evaluated Nulli-Fi using commercial 60 GHz, 16 element phased arrays transmitting IEEE 802.11ad packets [33]. Our results show that for 4 bit phase shifters, Nulli-Fi is able to create 3° narrow nulls that suppress interference by 18 dB and 10° wide nulls that supress interference by 10.5 dB while maintaining the main lobe within 1 dB. For 2 bit phase shifters, Nulli-Fi is still able to null interference by 12.6 dB. Nulli-Fi is also able null up to 5 different directions. We further compare Nulli-Fi with past null forming algorithms and demonstrate up to 10 dB better nulling and 37× faster convergence. We also evaluate NullFi's fast null steering protocol on top of the mm-Flex platform [33] to show that Nulli-Fi can find the direction of an unknown interference and null it within 290 ns. Finally, to demonstrate the effectiveness of Nulli-Fi in dense mmWave networks, we compare Nulli-Fi to past work that leverages the directionality of mmWave radios to enable many concurrent transmissions [27]. By nulling interference from side lobes, Nulli-Fi is able to achieve 2.6× higher data rate when 10 mmWave links (20 nodes) are transmitting concurrently.

Contributions: The paper has the following contributions:

- The paper presents the first practical system that can create nulls on mmWave phased arrays.
- The paper introduces a theoretically optimal algorithm for creating nulls and a novel discrete optimization framework that account for practical challenges in mmWave systems.
- The paper develops a fast null steering protocol to deliver a practical system.
- The system is built and evaluated on real phased arrays to demonstrate significant gains in suppressing interference.
- We have open sourced implementations of our algorithms and baselines on our git repository [36].

2 Related Work

There is a significant literature on millimeter wave beam shaping and steering. Past mmWave systems research, however, has mainly focused on beam alignment, i.e. developing protocols to quickly find the best direction to align the beams of a transmitter and receiver or to switch the beam to a different path to avoid blockage [17, 20, 27, 40, 56, 65, 66]. Some works also explore the problem of beam pattern synthesis [13, 42, 51]. However, these works focus on shaping the main lobe of the beam to achieve good antenna gain along the direction of communication. In contrast, we focus on forming and steering nulls to suppress interference.

Past work on mmWave networks proposes leveraging the directionality of mmWave links to enable dense spatial reuse and maximize the number links that can transmit simultaneously [27, 28]. However, the work shows that side lobe leakage from practical mmWave phased arrays limits the ability to enable spatial reuse. In section 6, we compare with this work to show that Nulli-Fi can enable 2.43× higher throughput than [27] when 10 links are transmitting concurrently. Another work [59] mitigates interference by aligning the natural nulls in the beam pattern toward the interferer. This, however, comes at the cost misaligning the mainlobe [59]. In section 6, we show that this can reduce the SNR by up to 10 dB. In contrast, Nulli-Fi creates new nulls that suppress interference while preserving the main lobe alignment.

Previous work on null forming in phased arrays is simulation based and to the best of our knowledge has not been implemented on practical mmWave phased arrays. Most of the past work ignores many of the practical limitations. Table 1 summarizes past work. Specifically, most methods assume that it is possible to arbitrarily set the phase and amplitude of the complex weights. Others do not require amplitude control but assume phase control is continuous and can be set arbitrary. However, mmWave phased control is highly quantized offering only 2 to 5 bits to control the phase [2, 11, 47]. Two works [49, 52] assume a digital phased array, i.e. each antenna is connected to a district transmitter or receiver and the complex weights can be set arbitrary in digital. Commercial mmWave phased arrays are mostly analog and have a single digital transmitter or receiver as shown in Fig. 2(a) [2–4, 33].

The closest to our work are [22, 23, 35] which use genetic algorithms to create nulls in case of discrete phase only control with 6 bits of quantization. However, these systems are not implemented in practice, ignore hardware imperfections, take many iterations to converge and can only create a point null for which Nulli-Fi has a closed-form solution. In section 6, we implement and compare with these methods to show that even if they account for hardware imperfections, Nulli-Fi still achieves 10 *dB* better nulling with the same running time, and is 37× faster with the same performance.

Authors in [8, 15] implement nulling on custom built phased arrays. However, they operate in the sub-6 GHz frequency range where it is significantly easier to build phased arrays with flexible control. In particular, [8] works at 4.5 GHz and uses phase shifters with a 9-bit control, i.e. it is possible to set the phase at a resolution of 0.7°. They first solve the nulling problem in the continuous phase domain using gradient descent and then round off the continuous values to the 9-bit discrete space. Millimeter wave phase shifters, however, typically support 2 to 5 bits phase shifters for which the quantization error become too large. In section 6, we implement and compare with this work and show that its performance significantly degrades as the number of bits decreases. Another work [15] operates at 2.4 GHz and use deep neural networks to create the nulls. However, the DNN architecture can only output continuous values and can suffer from over-fitting.[3] In contrast, this paper presents and extensively evaluates a solution that works for highly quantized phase on practical mmWave phased array.

Some works propose changing the positions of the antennas to create nulls in the beam pattern or reduce the side lobes [7, 24, 26, 31]. However, these techniques require new custom built hardware and are only suitable only for static applications with a fixed beam pattern and null locations. Finally, there is a large body of work that proposes interference nulling using MIMO techniques at sub 6 GHz frequencies [16, 18, 34, 37, 45]. These works are complementary to Nulli-Fi as they require multiple digital transmitters or receivers to perform digital beamforming and set arbitrary complex weights in digital to null the signals.

3 Primer

In this section, we provide a primer on phased arrays as well as genetic algorithms on which we base our optimization.

1. Phased Arrays: In analog phased arrays, an array of antennas is connected to a single transmitter or receiver through a single chain. The signal on each antenna n is multiplied with a complex weight $a_n = |a_n|e^{j\alpha_n}$ as shown in Fig. 2(a). By changing these weights, we can change the beam pattern and steer the main lobe of the beam in any direction. The beam pattern along a direction ϕ can be written as:

[3]Specifically, the paper mostly provides simulation results and only shows three examples of nulls created on real hardware.

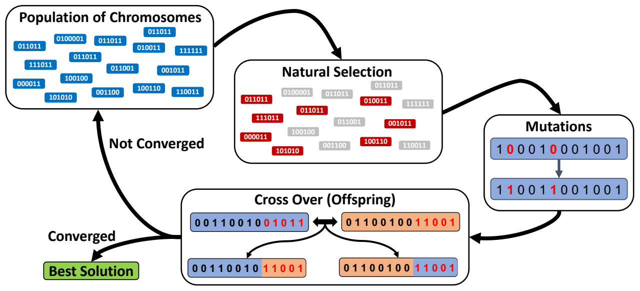


Figure 3: Overview of genetic algorithm

$$P(\phi) = \sum_{n=0}^{N-1} a_n e^{2\pi j \frac{d}{\lambda} n cos(\phi)} \quad (1)$$

where N is the number of antennas, λ is the wavelength of the signal, and d is the separation between adjacent antennas. We can steer the main lobe towards the direction ϕ by setting the complex weights to $a_n = e^{-2\pi j \frac{d}{\lambda} n cos(\phi)}$ which will cause the signals coming from direction ϕ to sum up constructively. For example, by setting $\phi = 75°$, we get the beam pattern shown in Fig. 2(b). The beam pattern exhibits natural nulls where $P(\phi) = 0$ and no signal is received along that direction. In practice, however, such perfect nulls are not possible. Hence, we define a null as a point in the beam pattern where $P(\phi)$ is extremely small (e.g. -25 dB relative to the main lobe). The deeper the null, the more effective it is at suppressing interference. Our goal is to find a setting of the complex weights to create a null along a certain angle ϕ_{null} while maintaining the amplitude level of the pattern at $\phi_{\text{main lobe}}$.

If we are able to control both amplitude and phase of the complex weights in a continuous manner, then we can easily create any beam pattern. In particular, we can transform Eq. 1 into a Fourier Transform by setting $f = -d/\lambda \cos(\phi)$. We can then construct any desired pattern and take its inverse Fourier transform to find the set of complex weights that we should use. Most practical phased arrays, however, do not support controlling the amplitude of the complex weights especially since modifying the phase is sufficient to steer the main lobe of the beam. These phased arrays use a component called a phase shifter to shift the phase of the signal on each antenna element. Hence, the problem is restricted to having $|a_n| = 1$, i.e. $a_n = e^{j\alpha_n}$. Unfortunately, the problem becomes even harder when we are limited to a quantized set of phase shifts, especially when the number of control bits used to set the phase shifter is small as the problem becomes non-convex and the search space is exponentially large. For example, for a 16 element array, and 4 bits (= 16 values) of resolution in phase-shifters, we get $16^{16} \approx 1.8\text{e}19$ possible patterns.

2. Genetic Algorithms: Genetic Algorithms (GAs) are a family of evolution-inspired algorithms designed to solve optimization problems. They are particularly useful when the search space is discrete and has many local maxima [60]. A high-level overview of the algorithm structure is depicted in Fig. 3. The algorithm starts by considering a set of *initial chromosomes* referred to as the *population*. Each chromosome represents one possible solution of the problem e.g. a setting of complex weights $(a_0, a_1, \cdots, a_{n-1})$. The first stage of the algorithm is *natural selection* where the chromosomes are ranked using a fitness function that evaluates how well each chromosome solves the problem e.g. how good of a null it creates. Fraction of chromosomes that are most fit to solve the problem are then selected and the rest are discarded. The remaining chromosomes give rise to new potentially fitter, chromosomes, which repopulate the population via *mutation* and *crossover*. In *mutation*, random bits used to represent the chromosomes are flipped to create new chromosomes whereas in *crossover*, two random parents give birth to two new chromosomes as shown in Fig. 3. Once the population reaches its original size, the fitness of the chromosomes is re-evaluated and the best chromosome is selected. The entire process keeps repeating until the algorithm converges, i.e. reaches some stopping criteria. While genetic algorithms work surprisingly well, they are completely arbitrary and do not exploit the underlying structure of the problem. As a result they take a long time to converge and can give sub-optimal results. Nulli-Fi builds on the high-level structure of such algorithms to design a new optimization framework customized to the problem of null steering.

4 Nulli-Fi

4.1 Nulling Algorithm

Assumptions: To begin, we state the set of assumptions under which we optimally solve the nulling problem. We will assume that the number of antenna elements, N, is even, and that the physical distance of adjacent antenna elements is $d \leq \lambda/2$, where λ is the wavelength. We further assume that we only have phase control over antenna elements, and before nulling, all the antenna elements are beamforming towards some direction ϕ_0, i.e. the phase shifts $\alpha_n = -2\pi nd/\lambda \cos\phi_0$ as described in Eq. 1 [4].

Preserving the Main Lobe: In order to preserve the main lobe of the beam directed towards ϕ_0, we limit any additional phase-shifts on each antenna element to $\pm\alpha^*$, i.e. $|\Delta\alpha_n| \leq \alpha^*$ for all n. We show that this limits the loss in the main lobe to at most $sin^2(\alpha^*)$. In particular, we prove the following lemma in Appendix A.5:

Lemma 4.1 *If $\alpha^* \leq 90°$, a maximum phase shift restricted to $\pm\alpha^*$ for each antenna element will result in a loss of at most $\sin^2(\alpha^*)$ in the main lobe.*

This would mean that for $\alpha^* = 15°$ (or 30° of freedom), the main lobe changes by at most 0.3 dB.

[4] In a discrete phase scenario, aligning towards any angle ϕ_0 is not possible and we must set all elements to the closest discrete value to ϕ_0.

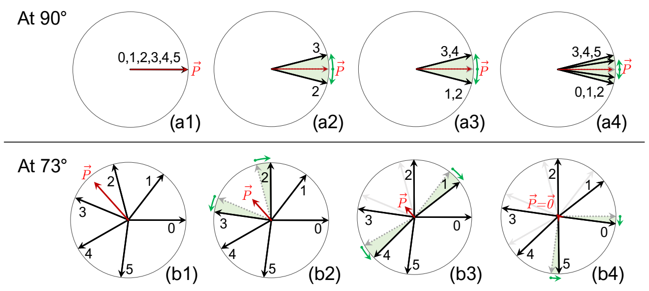


Figure 4: Example Nulli-Fi's nulling algorithm for $N = 6$ antenna elements with the main lobe at 90° an nulling at 73°.

Problem Formulation: Given the restrictions on the phase shifts to preserve the main lobe and the above assumptions, our problem becomes: *Given an angle ϕ, find a set of additional phase-shifts $\Delta\alpha_n$, such that $|P(\phi)|$ is zero (or as close to zero as possible), subject to $|\Delta\alpha_n| < \alpha^*$.*

Algorithm: Our algorithm works by representing the signal on each antenna as a vector in the complex plane. This representation is particularly useful since applying a phase shift is equivalent to rotating these vectors. Thus, our goal is to rotate these vectors to null the signal in the direction of ϕ. To better understand how this works, consider the example shown in Fig. 4. In this example, we have $N = 6$ antenna elements beamforming towards $\phi_0 = 90°$. The vectors $\vec{v_n}$ representing the signal on each element are indexed by: $0, 1, \cdots, 5$, and our goal is to create a null at $\phi = 73°$.

Initially, the vectors are aligned to sum up constructively to $\vec{P_{\phi_0}}$ along 90° as shown in Fig. 4(a1). [5] However, they are aligned differently along 73° and sum up to $\vec{P_\phi}$ as the signals come with a different phase at that direction as shown in Fig. 4(b1). To create a null along 73°, we will rotate each vector by an additional $\Delta\alpha_n$ to minimize the $\vec{P_\phi}$. The restriction $|\Delta\alpha_n| \leq \alpha^*$ will ensure that $\vec{P_{\phi_0}}$ along the main lobe is preserved as shown in Fig. 4(a2–a4). However, it will prevent us from arbitrarily rotating the vectors along 73°. To address this, we leverage the following key observation: *At any direction ϕ, all the vectors summing up to the pattern $\vec{P_\phi}$ come in pairs symmetrically located around the pattern.* For example, in Fig. 4(b1), the following pairs: $\{0, 5\}$, $\{1, 4\}$ and $\{2, 3\}$ are symmetrically located around $\vec{P_\phi}$.

The following lemma formalizes this observation. The proof of the lemma can be found in Appendix A.5.

Lemma 4.2 *At any direction ϕ, if $\Delta\alpha_n = 0$ for all n, then $\vec{v}_n$ and $\vec{v}_{N-1-n}$ are symmetrical around $\vec{P_\phi}$ for all n. That is, $\frac{1}{2}(\angle\vec{v}_n + \angle\vec{v}_{N-1-n})$ is the same as $\angle\vec{P_\phi}$ or $\angle\vec{P_\phi} + \pi$.*

Given this observation, the algorithm proceeds as follows. Choose a pair of symmetrical vectors around the pattern $\vec{P_\phi}$ and symmetrically rotate them towards $-\vec{P_\phi}$ as much as possible (i.e. until α^* degrees, or until a null is achieved). This

[5] In fact, $\vec{P} = 6e^{j0}$ but we have downscaled it by 6 for better visualization.

will reduce the beam pattern amplitude along ϕ as shown in Fig. 4 (b2) but it will *not* change its angle. This means that all the vectors remain symmetrical around the $\vec{P}_\phi$. If a null is achieved, we stop. If not, we repeat with another symmetrical pair as shown in Fig. 4 (b3,b4). Note that the same rotations are also applied at the main lobe in Fig. 4 (a2-a4). While these rotations result in a null at 73°, they cause only a 0.2 dB loss at 90°.

A pseudocode of the algorithm can be found in Alg. 1 in Appendix A.1. We also prove the following theorem regarding the optimality of our algorithm in Appendix A.5.

Theorem 4.3 *Given the constraint $|\Delta\alpha_n| < \alpha^*$, Alg. 1 gives the best nulling performance at any angle ϕ.*

It is worth noting that given the constraints, it is not always possible to achieve a perfect null i.e. $\vec{P}_\phi = \vec{0}$. In such cases, the above algorithm yields the deepest possible null. This also allows the algorithm to identify directions that can be perfectly nulled from those that cannot. In Appendix A.4, we provide further analysis and closed form solutions for the bounds of achievable nulling performance as a function of the direction of the null.

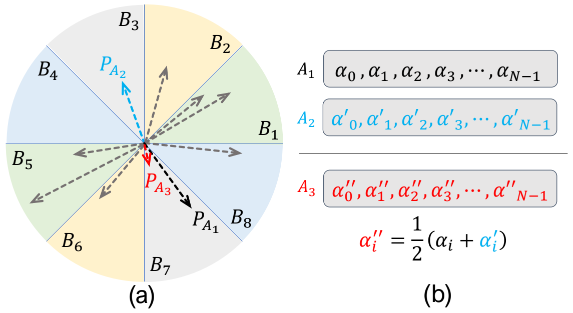
4.2 Optimization Framework

In this section, we show how to account for hardware imperfections and achieve multiple and wider nulls. We extend our definition of a null to be an interval 2β degrees wide around ϕ i.e., $[\phi-\beta,\phi+\beta]$ where the magnitude of the beam pattern is lower than a certain threshold. The input to our optimization are multiple such intervals $([\phi_i-\beta_i,\phi_i+\beta_i])$ where we wish to null interference. A pseudocode of our optimization framework can be found in Alg. 3.

Encoding: We will encode the solution i.e. the setting of the phase shifts α_n into chromosomes that form the basis of the genetic algorithm. Suppose the phase shifts are quantized using q bits, then each α_n can be represented as a bit string $(b_{n,1},\cdots,b_{n,q})$ where $b_{n,i}$ is the i^{th} most significant bit of α_n. A chromosome A can then be encoded as a concatenation of the N binary representations of the phase shifts: $A=(b_{0,1},\cdots,b_{0,q},b_{1,1},\cdots,b_{1,q},b_{N-1,1},\cdots,b_{N-1,q})$. We define P_A as the beam pattern associated with chromosome A.

***Initialization*:** While genetic algorithms generally start from a set of randomly generated chromosomes, we use the output of Alg. 1 to initialize our genetic algorithm. Specifically, for each null region $([\phi_i-\beta_i,\phi_i+\beta_i])$, we run Alg. 1 and find the optimal setting of α_n to create a null along ϕ_i. Each solution will give us a single initial chromosome. We then slightly perturb the values of the phase shifts to create a larger population of initial chromosomes. This dramatically improves the optimization's performance as we show in section 6.2.

Fitness function $F(A)$**:** This function evaluates the performance of any given chromosome A. In our problem setup, we



B_3
B_2
B_4
P_{A_2}
B_1
B_5
P_{A_3}
P_{A_1}
B_8
B_6
B_7
(a)
A_1 $\alpha_0,\alpha_1,\alpha_2,\alpha_3,\cdots,\alpha_{N-1}$
A_2 $\alpha'_0,\alpha'_1,\alpha'_2,\alpha'_3,\cdots,\alpha'_{N-1}$
A_3 $\alpha''_0,\alpha''_1,\alpha''_2,\alpha''_3,\cdots,\alpha''_{N-1}$
$\alpha''_i=\frac{1}{2}(\alpha_i+\alpha'_i)$
(b)

Figure 5: Nulli-Fi's crossover operation using buckets

define the fitness function as

$$F(A)=\min_{\substack{i=\{1,\ldots,L\}\\ \phi\in[\phi_i-\beta_i,\phi_i+\beta_i]}} -10\log_{10}\left(|P_A(\phi)|^2\right),$$

where $P_A(\phi)$ can be calculated from Eq. 1 by setting the complex weights to $e^{j\alpha_n}$. This fitness function $F(A)$ optimizes for the *worst* nulling performance in dB across all the regions we wish to null. In particular, the min point of $-10\log_{10}\left(|P_A(\phi)|^2\right)$ is the max point in $|P_A(\phi)|^2$ which is the least nulled point. Hence, the fittest chromosome, $A^*=\arg\max_A F(A)$, will give the best nulling performance across all directions since we optimized for the worst case.

***Natural Selection*:** At each iteration, we evaluate the fitness function for every chromosome and keep the ones with the best performance.In our implementation, we typically keep the top 50% of the chromosomes.

Cross-over. Recall from section 3, this operation is meant to combine two parent chromosomes A_1 and A_2, to give birth to a new, potentially fitter chromosome, A_3. Typically, the two parents A_1 and A_2 are chosen randomly. However, Nulli-Fi employs a more intelligent selection criteria. For simplicity, let us consider a single null point and use the same vector representation we used in section 4.1 to explain Nulli-Fi's cross-over operation.

To begin, we first group chromosomes into different buckets $1,\cdots,2B$. Bucket i contains all chromosomes A with $(i-1)\frac{\pi}{B}\le\angle\vec{P}_A<i\frac{\pi}{B}$. Fig. 5 (a) shows an example of these buckets for $B=4$, where buckets on the opposite sides of each other have the same color. In our cross-over operation, two parents A_1 and A_2 are then chosen at random, under the constraint that $\vec{P}_{A_1}$ and $\vec{P}_{A_2}$ are in opposing buckets (for example, B_3 and B_7). Then, a new chromosome A_3 is created by averaging the phase shifts of A_1 and A_2, as shown in Fig. 5 (b). The intuition behind this is that by taking the average phase shift of the two parents, the new chromosome will approximately have a pattern vector equal to the sum of its parents. Since the parents come from opposing buckets, the summation of their patterns will likely result in a smaller vector. This is depicted in Fig. 5 (a) where the red vector corresponding to the child chromosome A_3, is smaller than the pattern of either parent (depicted black and blue vectors). By exploiting the structure of the problem, Nulli-Fi is able to quickly generate fitter

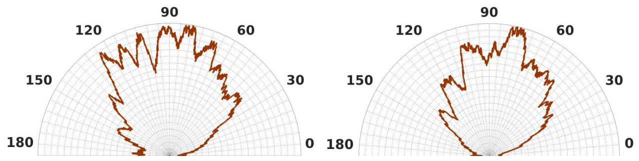


Figure 6: Non uniform radiation patterns of antenna elements

chromosomes with smaller $|P_A(\phi)|$ i.e. deeper nulls which improves the results as we show in section 6.2.

Mutation. In this step, we randomly flip bits in the parent chromosome with some probability to give rise to a new chromosome. Note that we preserve the current best chromosome A^* which remains unchanged during mutation.

Convergence. The algorithm converges once the best performing chromosome reaches a fitness threshold, or a maximum number of iterations has been reached. In the case of a single null, this threshold is directly governed by the output of Alg. 1. For multiple nulls, as one would expect, the performance usually does not reach the theoretical performance of a single null. In our implementation, we reduce the threshold by around 1 dB for every extra null region.

Preserving the main lobe. Similar to the optimal algorithm, we maintain $|\Delta\alpha_n| \leq \alpha^*$ to preserve the main lobe. This can be done by simply fixing the $q - \log_2(\pi/\alpha^*)$ most significant bits of α_n and not changing them throughout the optimization. However, in cases where q is very small, e.g. 2 bit phase shifters, such an approach does not hold. To address this, Nulli-Fi sets aside a subset of the antenna elements and does not change their phase shift throughout the entire optimization. This subset will contribute to the main lobe whereas the remaining antennas will contribute to create the null. Nulli-Fi dynamically chooses the antenna elements that are contributing the most to the main lobe to be in this subset and allows phase shifts for the ones that are contributing the least to the main lobe. A pseudocode for this process can be found in Alg. 2 in section A.1 of the appendix.

Accounting for Hardware Imperfections: There are two types of hardware imperfections: (1) phase offsets due to different wire lengths or paths that the signals traverse, and (2) non-uniform antenna element radiation patterns. In particular, the signal on each antenna incurs an additional δ_n and signals coming from a direction ϕ incur an additional attenuation of $R_n(\phi)$. Fig. 6 shows the radiation patterns of two antenna elements on our hardware setup described in section 5. As can be seen, antennas do not receive the signal uniformly across all directions. While these factors do not severely affect the quality of the main lobe, they have a more significant impact on the nulling performance of the phased array, as we show in section 6. This is because the beam pattern computed using Eq. 1 and used for evaluating the fitness function is no longer valid in practice. We can measure these imperfections using a simple calibration procedure outlined in detail in appendix A.2. Once measured we can modify Eq. 1

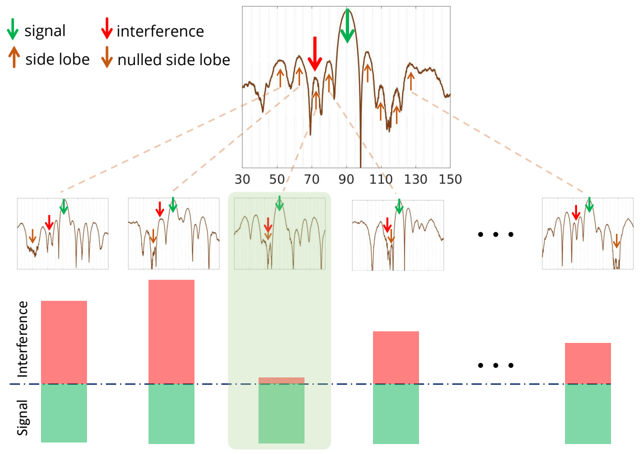


Figure 7: Illustration of Nulli-Fi's nulling alignment and interference suppression.

as follows:

$$P(\phi) = \sum_{n=0}^{N-1} a_n R_n(\phi) e^{j\delta_n} e^{-2\pi j \frac{d}{\lambda} n cos(\phi)}, \qquad (2)$$

We observe these imperfections to be stable and, hence, can be measured once. By modifying the fitness function to account for these hardware imperfections, we can generate beam patterns that achieve good nulling performance in practice.

4.3 Fast Null Steering Protocol

Now that we have a framework to form nulls at any desired direction, we need to find a practical way to align and suppress nulls in a real network. In this section, we present a simple yet fast and practical protocol to do so. The protocol finds and suppresses interferers in succession, by enforcing wide nulls at the high-level side-lobes of the pattern.

We begin with a simple example, where there is only one interferer. Consider the pattern in Fig. 7. As can be seen, the pattern has a number of significant side lobes, denoted by upwards brown arrows, which are the most likely to receive interference from other links in the network. Nulli-Fi finds these side-lobes, and computes corresponding patterns that have nulls at each side lobe as shown on the second row in Fig. 7, while keeping the main lobe. The hardware then quickly sweeps through these patterns, computing the Received Signal Strength (RSS) corresponding to each pattern. If the RSS drops for one of these patterns, it means that the interference was suppressed. This way, we can eliminate the interferer. To make the algorithm even faster, Nulli-Fi checks the SINR value at after each beam switch, and stops if the SINR is within a threshold of its original value.

Following this example in case of multiple interferers, we first suppress the one with the highest power. Once this interferer is nulled, we keep a null at its direction at all times,

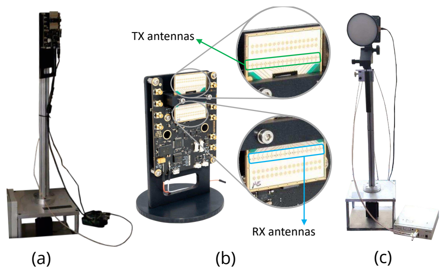


Figure 8: Hardware used in Nulli-fi.

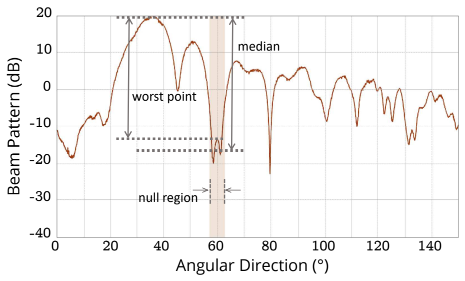


Figure 9: Defining the evaluation metrics.

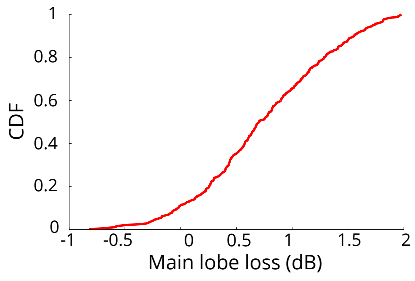


Figure 10: Main lobe loss in Nulli-Fi.

and search for the next interferer with the highest power. We repeat this process until all interferers are suppressed. We note that since our protocol only looks for interferences at high-level side lobes of the pattern, it will be much faster than a full scan, while remaining effective in improving the SINR, as we show in section 6.4.

5 Implementation

Nulli-Fi's Setup. Nulli-Fi is implemented using the off-the-shelf Sivers IMA EVK06002 platform [3], equipped with a 60 GHz 16-element linear phased array shown in Fig. 8(b). In order to measure the beam patterns, we mount the phased array radios on a steerable platform controlled through an Arduino as shown in Fig. 8(a). Our testbed also includes a 60 GHz Pasternack PEM009-KIT [1] equipped with a directional 3-degree horn antenna (Fig. 8(c)) which we use to transmit signals in order to measure the generated beam patterns. All hardware devices were connected to a machine running Ubuntu 18.04 through USRP N210 software defined radios. The center frequency in all experiments was 60.48 GHz. We run our experiments in 4 different rooms in 8 different locations, and in each location, we test 125 distinct combinations of directions of communication and interference. The EVK06002 platform offers flexible phase control for each of the antenna element weights which allows us to experiment using different number of bits. We evaluate Nulli-Fi using at most 4 bits of phase resolution, i.e. 16 distinct phase shift values per antenna element. We also show Nulli-Fi's performance for more coarse-grained control on phase, specifically 2 bit and 3 bit phase resolution. We also calibrate the array as described in Appendix A.2.

Nulli-Fi + mm-Flex Setup. We also implemented Nulli-Fi on top of the mm-Flex platform [33], to evaluate our null steering protocol. We transmitted IEEE 802.11ad control frames, where we use 10 Golay sequences to switch beam patterns. Our setup can switch between beam patterns once every 54.5 nanoseconds, during which we are able to measure RSS values corresponding to that pattern. This way, sweeping through 10 beam patterns takes less than 0.55μs. Further details of this setup can be found in Appendix A.3.

6 Results

6.1 Evaluation Metrics

We start by describing the evaluation metrics used to quantify Nulli-Fi's performance. Fig. 9 illustrates some of the metrics on an example beam pattern created by Nulli-Fi.

• *Null Width:* Since we create wide nulls, we define the null width as the region of directions nulled in the beam pattern.
• *Worst-point nulling performance:* Minimum amount of nulling in the target null region, measured as the difference between the peak of the main lobe and maximum point in the null region as shown in Fig. 9.
• *Median nulling performance:* Median amount of nulling in the target nulling region measured as the difference between the peak of the main lobe and median point in the null region.
• *SINR Gain:* Gain in Signal-to-Interfernce plus Noise Ratio before and after nulling the interferer.
• *Main Lobe Loss:* Loss in the main lobe power compared to a beam pattern without the imposed nulls.
• *Number of Nulls:* Number of different directions in which nulls are created.
• *Quantization Level:* Number of control bits used to set the phase on the phase shifters.
• *Number Iterations:* Number of iterations it takes for the optimization to converge.
• *Null Steering Latency:* Time it takes to steer the null towards an interferer once the interferer appears.

6.2 Baselines

We compare Nulli-Fi to the following baselines:

(1) Quantize-Continuous [8] – This baseline solves the problem in the continuous domain and then quantizes the phase solution to the nearest available discrete phase values.

(2) Genetic-Algorithms [22] – We compare Nulli-Fi to past work that uses genetic algorithms to create nulls.

(3) Shift-Pattern [59] – This baseline steers the main lobe a bit away from the direction of communication in order to align the natural nulls in the signal with the interferer.

(4) BounceNet [27] – This work aims to enable dense spatial reuse in mmWave networks by leveraging the directionality of mmWave beam patterns.

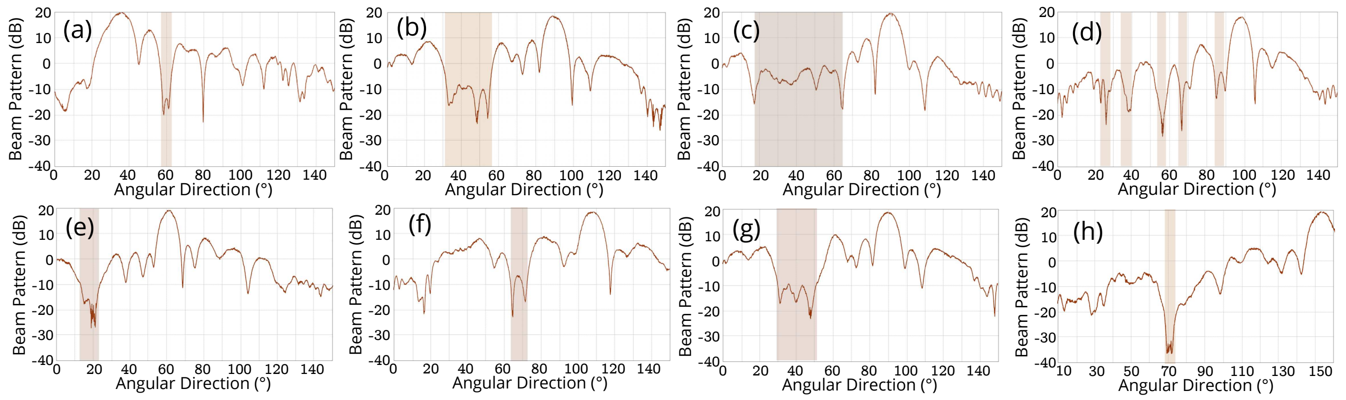
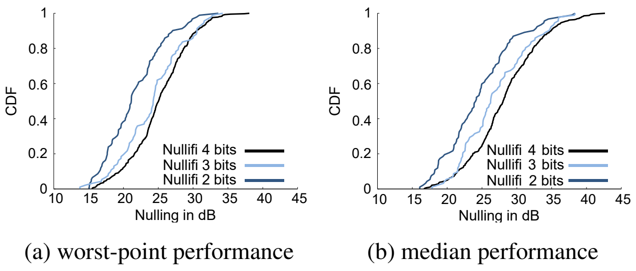


Figure 11: Examples of nulls created in hardware with different null with, number of nulls, direction of the null, and direction of the main lobe.



(a) worst-point performance (b) median performance

Figure 12: Nulli-Fi's performance using fewer bits.

6.3 Nulling Performance

We start by evaluating Nulli-Fi's ability to create nulls and compare it with past work.

1. Nulli-Fi's Nulling Performance: Fig. 11 shows a few examples of beam patterns with nulls created by Nulli-Fi on our phased array for different null directions, main lobe directions, null widths, and number of nulls using 4-bit phase shifters. As can be seen in Fig. 11(e, h), Nulli-Fi can create nulls as deep as -20 dB and -35 dB (40 dB and 55 dB below the main lobe respectively). Nulli-Fi can also create nulls that are as wide as 20° while maintaining a median nulling performance that is 20 to 25 dB below the main lobe as shown in Fig. 11(b, g). It can create up to 5 different nulls as shown in Fig. 11(d).

Fig. 12 shows a CDF of the median and worst-point nulling performance in more than 1000 experiments. Nulli-Fi's 50^{th} percentile is about 29 dB for the median nulling and 25 dB for the worst-point nulling. Since commercial 802.11ad hardware today, like laptops and tablets, comes equipped with only 2 bit phase control in the phased arrays [2, 11], we evaluate Nulli-Fi's performance using fewer bits of phase resolution. Fig. 12 also plots the CDF of the nulling performance with Nulli-Fi using 2 and 3 bits of phase control. While the nulling performance degrades with fewer bits, Nulli-Fi is still able to achieve a 24.1 dB median and 21.1 dB worst-point performance using only 2 bits, and 26.2 dB median and 24.3 dB worst-point performance using 3 bits of phase resolution.

Finally, we measure the main lobe loss suffered due to creating nulls, and plot the empirical CDF in Fig. 10. As can be seen, the median and the 90^{th} percentile values of the main lobe power loss are only 0.58 dB and 1.46 dB respectively, demonstrating Nulli-Fi's ability to preserve the main lobe while creating nulls.

2. Nulling Performance vs. Null Width: There is a natural trade-off between the width of the null created and its median (or worst-point) performance. To examine this, we evaluate Nulli-Fi's ability to create very narrow nulls like the ones shown in Fig. 11 (a, h) as well as very wide null regions like the ones shown in Fig. 11 (b, c). We run experiments where we create nulls of different widths ranging from 1° to 50°, and plot the median and worst-point performance in Fig. 13(a).

While the median nulling performance remains more than 25 dB even for nulls as wide as 50°, the worst-point performance deteriorates much more quickly. This point becomes clear when we consider the fact that the total radiated power in the beam pattern has to be conserved, implying that nulling one region will cause other regions to have an amplification in power. Therefore, while it may be possible to keep the median point in the target null region low for wide nulls, it becomes increasingly difficult to ensure that the worst-point in the target null region remains low as well.

It is worth noting that such overly wide nulls might not be needed in practice and nulls with width of 5° to 10° might be more than enough to account for inaccuracy in the estimating the direction of interferer. One could, however, use very wide null to preemptively supress less powerful interferers.

3. Nulling Performance vs. Number of Nulls: As mentioned previously, in practical networks there could be multiple sources of interference (separate signals or multipath), each occurring at a different angle. Thus, we test Nulli-Fi's ability to create l simultaneous null regions for $1 \leq l \leq 5$. We run 200 experiments for each l by randomly assigning the null regions. We constrain all null regions to be at most 10° wide, and in the case of multiple nulls, any two null regions should be at least 10° apart (otherwise, we observe that the two nulls

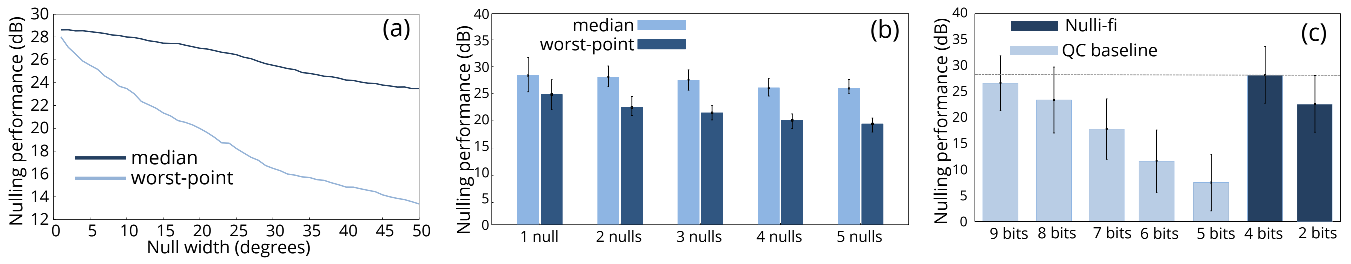


Figure 13: (a) Null performance versus width of the null. (b) Null performance versus number of nulls. (c) Nulli-Fi verus baseline as a function of number of quantization bits.

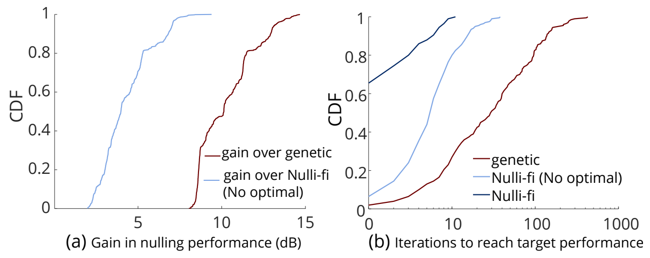


Figure 14: performance of Nulli-Fi's algorithm against baselines

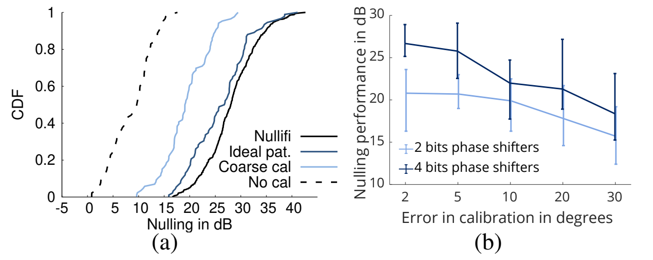


Figure 15: Importance of accounting for hardware imperfections and sensitivity to calibration errors.

merge into one wider null). Fig. 13(b) shows the median and worst-point nulling performance for different numbers of null regions. Note that for multiple null regions, we present the median and worst-point performance numbers for the poorest performing null in the beam pattern. As can be seen in Fig. 13(b), even with 5 null regions, Nulli-Fi is able to achieve 25.8 dB median and 19.1 dB worst-point performance. Fig. 11(d) shows an example of 5 null regions generated by Nulli-Fi.

4. Nulling Performance vs. Baselines: We run more than 1000 experiments for creating nulls at different angles, with different main lobe directions and null widths. Our nulling angles and main lobe directions range from the 30° to 150° region, and null widths range from 5° to 20° width. In all of the experiments, the target null region does overlap with the ±10° region around the main lobe direction. In these experiments, we focus on the performance for creating a single null. In Fig. 13(c), we compare Nulli-Fi's performance against the *Quantize-Continuous* baseline. The continuous solutions are quantized to $b = 5, 6, \cdots, 9$ bits of phase resolution, whereas Nulli-Fi is implemented on real hardware with 4 bits of phase resolution. Nulli-Fi's performance exceeds *Quantize-Continuous* even with 9 bits of phase resolution compared to Nulli-Fi's 4 bits. This shows that simply quantizing the continuous phase solution (especially quantizing to less than 7 bits) does not work for practical phased arrays.

We also compare Nulli-Fi with genetic algorithm [22] (*Genetic*) as well as with Nulli-Fi's optimization framework without initializing it with the solution of our optimal algorithm (*Nulli-Fi No optimal*). We run experiments where each algorithm is required to create single nulls at 200 different directions. We fix a target nulling performance of −20 dB and record the number of iterations required to achieve the desired nulling. Fig. 14(a) plots a CDF of the No. of iterations, showing that Nulli-Fi converges almost two orders of magnitude faster than *Genetic*. The figure also shows that Nulli-Fi's optimal algorithm enables much faster convergence and in many cases already gives a nulling performance of −20 dB. Hence, Nulli-Fi converges in a single iteration.[6] Moreover, By comparing the 99th percentile of *Genetic* and *Nulli-Fi(No optimal)*, which comprises cases where it is more difficult to create nulls, we can see that Nulli-Fi's novel crossover scheme helps in pushing the algorithm faster towards the desired nulling performance. Next, we fix the number of iterations to 10 for all three algorithms and plot the CDF of the nulling gain achieved by Nulli-Fi over each algorithm in Fig. 14(a). Nulli-Fi achieves a median gain in nulling of 10 dB over *Genetic* and 4 dB over *Nulli-Fi (No optimal)*.

5. Sensitivity to Calibration & Hardware Imperfections To show the significance of accounting for hardware imperfections, we run experiments to evaluate nulling performance using coarse and absent calibration on the phased array front-end. We also run experiments without accounting for non-uniform antenna radiation patterns discussed in Section 4. As mentioned previously, such imperfections have little effect on the location and power of the main lobe, but will lead to significant errors in null forming [43]. Fig. 15a, shows a CDF of the nulling performance. Without accounting for hardware imperfections, the median nulling is only 10 dB which is 17 dB worse than Nulli-Fi. With simple coarse calibration, the performance already improves by 7 dB. The figure also shows that while ignoring the non-uniform radiation patterns is not as severe, it still reduces the median nulling performance by 3 dB compared to Nulli-Fi. Finally, Fig. 15b shows the sensitiv-

[6]The baselines might also converge in a single iteration if the desired null happens to align with a natural null in the beam pattern.

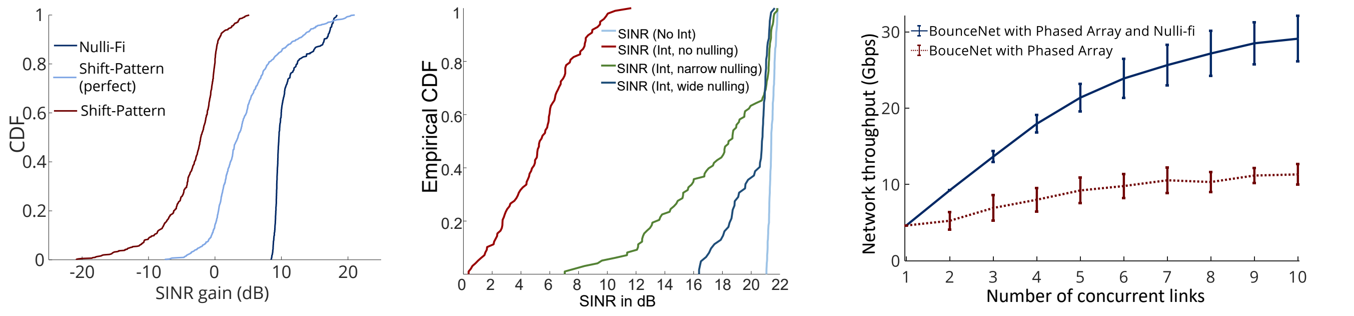


Figure 16: (a) Nulli-Fi's SINR gain over shifting the pattern baseline. (b) Nulli-Fi's ability to steer to and suppress interference. *Int* in the legend indicates the presence of interference. (c) Nulli-Fi's network throughput gains in dense networks

ity of Nulli-Fi's performance to errors in calibration. While the performance degrades as the calibration error increases, the figure shows that Nulli-Fi is robust to calibration errors less than 5° and can still null even if the calibration errors are 30°. It is worth noting that the degradation is less sharp in the case of 2 bit phase shifters. This is likely due to the fact that the phase is highly quantized and hence any calibration error is within the quantization errors.

6.4 Suppressing Interference & Improving Throughput

In this section, we present results for Nulli-Fi's ability to steer the null to suppress interference.

1. Null Steering Algorithm: Here we show the performance of Nulli-Fi's ability to suppress new, unforeseen interferences. To this end, we implemented Nulli-Fi on mm-Flex [33], and we ran close to 700 experiments with different relative locations, power levels for the interference. The experimental setup of this part was explained in section 5.

Fig. 16(b) shows the CDF of Signal to Interference plus Noise (SINR) ratio under four different conditions. As revealed by the figure, throughout all experiments, we set the SINR for when there is no interference to around 21 dB. By introducing the interference (shown by the maroon curve) whose location and power is unknown to the system, the SINR drops to as low as < 1 dB. Then, by running Nulli-Fi's null steering algorithm as described in section 4.3, Nulli-Fi chose and suppressed the side lobes one by one in order to restore the original SINR. The algorithm would stop once it reached within 1 dB of the original SINR, or it suppressed each side-lobe once (up to 10 side lobes).

We did this experiment in two regimes of narrow (2°) and wide (10°) nulls, shown by green and dark blue curves respectively. As seen in both curves, Nulli-Fi can improve the SINR by a median of 13 dB and 15 dB for narrow and wide nulls respectively [7]. We therefore see that Nulli-Fi is able to bring the SINR very close to its original value in the absence of the interferer in all cases. This shows that it is sufficient to look for interference only at the side lobes, as opposed to performing a full scan.

[7]These experiments were run in the IMDEA networks lab. We found that the SINR gains of Nulli-Fi + mm-Flex is around 3-5 dB larger than Nulli-Fi alone due to a slightly different hardware setup and a lower noise floor

We mention a trade-off between using narrow (2°) and wide (10°) nulls. We expect wide nulls to have a higher chance of capturing the interference, albeit with lower suppressing power as we showed in section 6.3. We see here that this is indeed the case: Compared to narrow nulls, wide nulls have a higher chance of capturing the interference, while narrow nulls suppress the interference better. This is also reflected by the tails of the green and the dark blue curves in Fig. 16 (b). We also compare the runtime of Nulli-Fi's algorithm against the baseline of fully scanning all angles. The numbers are reported using the fast beam switching and RSS measurement technique implemented in [33]. We see that Nulli-Fi's algorithm run on average in 290 nano-seconds, with a standard deviation of 115ns, which is more than 10× faster than a full search scheme, whose average and standard deviation for running time are 3.280 and 1.616 μs, respectively.

Finally, we compare Nulli-Fi's performance in gaining SINR with the Shift-Pattern baseline. We fix the signal and the interference power, and we move the interferer to different angles, and run 100 experiments to measure the gain in SINR. We compare Nulli-Fi with this baseline in two cases. In the first case, *Shift-Pattern (perfect)*, we assume perfect knowledge of the beam pattern, in which case Shift-Pattern chooses the best (deepest) null direction out of all direction within an interval of 10 degrees around the current pattern. We note that although this always reduces the power at the desired null location, it may lead to significant losses in the mainlobe, as we can see in Fig. 16(a). Things get even worse once we use the theoretical beam pattern to predict the optimum shifting amount (*Shift-Pattern*), which almost always results in a loss of SINR, due to inaccuracy of the theoretical beam pattern in predicting the real one. Nulli-Fi, on the other hand, always gives at least $8dB$ improvement in SINR, outperforming both versions of the baselines in almost all cases. This shows that simply shifting the pattern does not work in a practical system, since by shifting towards a null, we also shift the main lobe away from the direction of communication.

2. Throughput in Dense Networks: Fig. 16(c) demonstrates

No. of Links	Max Gain	90^{th} Perc. Gain	Median Gain	No. of Links	Max Gain	90^{th} Perc. Gain	Median Gain
1	1×	1×	1×	6	3.60×	2.83×	2.33×
2	2×	2×	1.58×	7	3.50×	2.72×	2.38×
3	3×	3×	1.8×	8	3.38×	2.94×	2.41×
4	4×	2.86×	2.12×	9	2.97×	2.77×	2.44×
5	4.16×	2.81×	2.27×	10	3.09×	2.68×	2.43×

Table 2: Gains in Total Network Data Rate from Nulli-Fi

Nulli-Fi's performance gains in dense networks. To do so, we implement and compare with BounceNet [27] which exploits the directionality of mmWave phased arrays to enable dense spatial resuse. We incorporate Nulli-Fi's nulling into BounceNet. Fig. 16(c) plots the total network data rate as the number of links in the network increases from 1 to 10. We compare Nulli-Fi against a regular phased array testbed using standard codebook-based beam patterns without interference nulling. As seen in the figure, due to significant interference in dense networks caused by side lobe leakages and multipath, a regular phased array equipped testbed can achieve only up to 11.31 Gbps network data rate for 10 links. Nulli-Fi, on the other hand, can effectively null out interference at each link and can increase the total data rate for the same phased array testbed to 29.1 Gbps, providing a gain of 2.6×.

In Table 2, we present further statistics on the gains in total data rate achieved by Nulli-Fi over a regular phased array testbed for different number of links n in the network. For each n we perform 100 different experiments by randomizing the client and AP positions. The result shows that for up to $n = 4$ communication links, Nulli-Fi can achieve the maximum possible gain of $n\times$ over the vanilla phased array testbed. Thus, in certain experiments Nulli-Fi was able to get all 4 links to communicate simultaneously by nulling out interferences, whereas the regular phased arrays were not able to exploit any spatial reuse whatsoever due to side lobe leakages and interference. Note that this gain saturates and begins to fall as the number of links increases due to increased interference. Nonetheless, Nulli-Fi is still able to achieve gains as high as 3.09× in network data rate for 10 links in the network. Table 2 also shows results for 90^{th} percentile and median gains.

7 Discussion and Limitations

In this paper, we introduced novel algorithms that significantly boost the convergence speed and improved the nulling performance compared to past work. Furthermore, the system enabled the first practical implementation of null steering by accounting for hardware restrictions, incorporating hardware imperfections and achieving wide and multiple nulls.

Importance of Convergence Speed: One might wonder, however, why having a faster algorithm is important in practical network deployments. The reason has to do with today's commercial phased array hardware. In particular, the hardware typically stores a codebook of different beam patterns in the on-board memory, and the mmWave radio beams towards different directions by reading the precomputed phase shift values from the codebook. As such, it is not possible to store precomputed beam patterns for all combinations of main-lobes and nulling directions. For instance, if we consider beam patterns with just one null, we would need to store a beam pattern corresponding to each main-lobe direction and each null direction, so a total of 180×180 beam patterns to achieve a null accuracy of 1 degree. This requirement grows exponentially with the number of nulls and would require gigabytes of memory for more than 2 nulls. Compare this to today's millimeter wave phase array that can store 16 to 256 codebooks. Hence, pre-computing and storing the beam patterns is not feasible. This is precisely why it is important to have an efficient algorithm that can converge quickly and compute the required beam patterns in real-time operation. This can allow even further optimization of the beam pattern at run-time which was not possible earlier in the codebook approach. Therefore, the speed of convergence is an important metric in evaluating the different nulling algorithms.

Limitations. We point out a few matters worth considering.

• In this paper, Nulli-Fi enables nulling the interference at the receiver. This is because it is easy for receivers to sense the direction of interference and change their beam pattern to suppress it. That said, there is an opportunity to perform nulling from the transmitter side where the transmitter creates a null in its beam pattern to suppress its own signal in direction of other receivers. This, however, would require an efficient protocol that allows the transmitter to discover the direction of those other receivers at which it is creating interference. Performing nulling from both transmitter and receive side would further improve the performance of the network. However, we leave that for future work.

• Once Nulli-Fi successfully nulls an interferer, it may not sense when it disappears. As a result, if new interferers appear, Nulli-Fi may not know whether to create more nulls or to switch the direction of the null. This can potentially be solved by periodically checking each nulled region for the presence of interference when it is not receiving packets.

• Nulli-Fi's framework is designed for phase shifters that use analog beamforming, which is common for commercial, practical phased arrays. While digital beamforming introduces a substantial overhead in terms of cost and power consumption, the in-between class of hybrid beamforming allows for more flexibility in terms of nulling. Exploring nulling in hybrid beamforming is left for future work.

Acknowledgement

We would like to thank our shepherd Bo Chen as well as the reviewers for their feedback. We would also like to thank Sepehr Madani and Junfeng Guan for their comments. We are also grateful to NSF (award numbers: 1750725, 1824320), Google, Facebook Connectivity Lab, and the Sloan Foundation for partially funding this research.

References

[1] Pasternack pem009-kit. https://www.pasternack.com/60-ghz-development-systems-category.aspx. Accessed: 2020-09-17.

[2] Qualcomm 802.11ad products to lead the wayfor multi-band wi-fi ecosystem. bit.ly/2Fy2CnM. Accessed: 2020-09-17.

[3] Sivers ima evk06002 platform. https://www.siversima.com/product/evk-06002-00/. Accessed: 2020-09-17.

[4] O. Abari, H. Hassanieh, M. Rodreguiz, and D. Katabi. Poster: A millimeter wave software defined radio platform with phased arrays. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking*, pages 419–420, 2016.

[5] O. P. Acharya, A. Patnaik, and S. N. Sinha. Null steering in failed antenna arrays. *Applied Computational Intelligence and Soft Computing*, 2011:4, 2011.

[6] K. Akdagli. Null steering of linear antenna arrays using a modified tabu search algorithm. *Progress In Electromagnetics Research*, 33:167–182, 2001.

[7] A. Alphones and V. Passoupathi. Null steering in phased arrays by positional perturbations: a genetic algorithm approach. In *Proceedings of International Symposium on Phased Array Systems and Technology*, pages 203–207. IEEE, 1996.

[8] C. Baird and G. Rassweiler. Adaptive sidelobe nulling using digitally controlled phase-shifters. *IEEE Transactions on Antennas and Propagation*, 24(5):638–649, 1976.

[9] G. C. Bower. Simulations of narrow-band phased-array null formation for the ata. *ATA Memo Series*, 37, 2001.

[10] L. Chettri and R. Bera. A comprehensive survey on internet of things (iot) toward 5g wireless systems. *IEEE Internet of Things Journal*, 7(1):16–32, 2019.

[11] N. Choubey and A. Panah. Introducing facebook's new terrestrial connectivity systems-terragraph and project aries. *Facebook Research*, 2016.

[12] D. A. Day. Fast phase-only pattern nulling for large phased array antennas. In *2009 IEEE Radar Conference*, pages 1–4. IEEE, 2009.

[13] D. De Donno, J. Palacios, and J. Widmer. Millimeter-wave beam training acceleration through low-complexity hybrid transceivers. *IEEE Transactions on Wireless Communications*, 16(6):3646–3660, 2017.

[14] F. Firyaguna, J. Kibilda, C. Galiotto, and N. Marchetti. Coverage and spectral efficiency of indoor mmwave networks with ceiling-mounted access points. In *GLOBECOM 2017-2017 IEEE Global Communications Conference*, pages 1–7. IEEE, 2017.

[15] R. Ghayoula, N. Fadlallah, A. Gharsallah, and M. Rammal. Phase-only adaptive nulling with neural networks for antenna array synthesis. *IET microwaves, antennas & propagation*, 3(1):154–163, 2009.

[16] S. Gollakota, S. D. Perli, and D. Katabi. Interference alignment and cancellation. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, pages 159–170, 2009.

[17] M. K. Haider, Y. Ghasempour, and E. W. Knightly. Search light: Tracking device mobility using indoor luminaries to adapt 60 ghz beams. In *Proceedings of the Eighteenth ACM International Symposium on Mobile Ad Hoc Networking and Computing*, pages 181–190, 2018.

[18] E. Hamed, H. Rahul, M. A. Abdelghany, and D. Katabi. Real-time distributed mimo systems. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 412–425, 2016.

[19] G. R. Harik, F. G. Lobo, and D. E. Goldberg. The compact genetic algorithm. *IEEE transactions on evolutionary computation*, 3(4):287–297, 1999.

[20] H. Hassanieh, O. Abari, M. Rodriguez, M. Abdelghany, D. Katabi, and P. Indyk. Fast millimeter wave beam alignment. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 432–445, 2018.

[21] R. L. Haupt. Adaptive nulling in monopulse antennas. *IEEE transactions on antennas and propagation*, 36(2):202–208, 1988.

[22] R. L. Haupt. Phase-only adaptive nulling with a genetic algorithm. *IEEE Transactions on Antennas and Propagation*, 45(6):1009–1015, 1997.

[23] R. L. Haupt. Adaptive nulling with weight constraints. *Progress In Electromagnetics Research*, 26:23–38, 2010.

[24] J. A. Hejres. Null steering in phased arrays by controlling the positions of selected elements. *IEEE transactions on antennas and propagation*, 52(11):2891–2895, 2004.

[25] H. M. Ibrahim. Null steering by real-weight control-a method of decoupling the weights. *IEEE transactions on antennas and propagation*, 39(11):1648–1650, 1991.

[26] T. Ismail and M. M. Dawoud. Null steering in phased arrays by controlling the element positions. *IEEE Transactions on Antennas and Propagation*, 39(11):1561–1566, 1991.

[27] S. Jog, J. Wang, J. Guan, T. Moon, H. Hassanieh, and R. R. Choudhury. Many-to-many beam alignment in millimeter wave networks. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 783–800, 2019.

[28] S. Jog, J. Wang, H. Hassanieh, and R. R. Choudhury. Enabling dense spatial reuse in mmwave networks. In *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*, pages 18–20. ACM, 2018.

[29] N. Karaboga, K. Güney, and A. Akdagli. Null steering of linear antenna arrays with use of modified touring ant colony optimization algorithm. *International Journal of RF and Microwave Computer-Aided Engineering*, 12(4):375–383, 2002.

[30] S. Karimkashi and A. A. Kishk. Antenna array synthesis using invasive weed optimization: A new optimization technique in electromagnetics. In *2009 IEEE Antennas and Propagation Society International Symposium*, pages 1–4. IEEE, 2009.

[31] M. M. Khodier and C. G. Christodoulou. Linear array geometry synthesis with minimum sidelobe level and null control using particle swarm optimization. *IEEE transactions on antennas and propagation*, 53(8):2674–2679, 2005.

[32] L. Kogan. A minimum gradient algorithm for phased-array null formation. *Radio science*, 40(2), 2005.

[33] J. O. Lacruz, D. Garcia, P. J. Mateo, J. Palacios, and J. Widmer. mm-flex: an open platform for millimeter-wave mobile full-bandwidth experimentation. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, pages 1–13, 2020.

[34] K. C.-J. Lin, S. Gollakota, and D. Katabi. Random access heterogeneous mimo networks. *ACM SIGCOMM Computer Communication Review*, 41(4):146–157, 2011.

[35] C. Lu, Y. Wu, R. Mahmoudi, M. K. Matters-Kammerer, and P. G. Baltus. A mm-wave analog adaptive array with genetic algorithm for interference mitigation. In *2012 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2373–2376. IEEE, 2012.

[36] S. Madani. Nullfi code for null steering. Git Repository, 2021. gitlab.engr.illinois.edu/smadani2/nulling-python.

[37] A. F. Molisch, V. V. Ratnam, S. Han, Z. Li, S. L. H. Nguyen, L. Li, and K. Haneda. Hybrid beamforming for massive mimo: A survey. *IEEE Communications Magazine*, 55(9):134–141, 2017.

[38] T. Moon, J. Gaun, and H. Hassanieh. Online millimeter wave phased array calibration based on channel estimation. *IEEE Design & Test*, 2020.

[39] M. Mouhamadou, P. Vaudon, and M. Rammal. Smart antenna array patterns synthesis: Null steering and multi-user beamforming by phase control. *Progress In Electromagnetics Research*, 60:95–106, 2006.

[40] T. Nitsche, A. B. Flores, E. W. Knightly, and J. Widmer. Steering with eyes closed: mm-wave beam steering without in-band measurement. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 2416–2424. IEEE, 2015.

[41] Y. Niu, Y. Li, D. Jin, L. Su, and A. V. Vasilakos. A survey of millimeter wave communications (mmwave) for 5g: opportunities and challenges. *Wireless networks*, 21(8):2657–2676, 2015.

[42] J. Palacios, D. De Donno, and J. Widmer. Lightweight and effective sector beam pattern synthesis with uniform linear antenna arrays. *IEEE Antennas and Wireless Propagation Letters*, 16:605–608, 2016.

[43] L. Poli, L. Manica, P. Rocca, E. Giaccari, and A. Massa. Tolerance analysis with phase errors in linear arrays by means of interval arithmetic. In *EuCAP 2014*. IEEE, 2014.

[44] J. Qiao, L. X. Cai, X. Shen, and J. W. Mark. Stdma-based scheduling algorithm for concurrent transmissions in directional millimeter wave networks. In *2012 IEEE International Conference on Communications (ICC)*, pages 5221–5225. IEEE, 2012.

[45] H. S. Rahul, S. Kumar, and D. Katabi. Jmb: scaling wireless capacity with user demands. *ACM SIGCOMM Computer Communication Review*, 42(4):235–246, 2012.

[46] T. S. Rappaport, S. Sun, R. Mayzus, H. Zhao, Y. Azar, K. Wang, G. N. Wong, J. K. Schulz, M. Samimi, and F. Gutierrez. Millimeter wave mobile communications for 5g cellular: It will work! *IEEE access*, 1:335–349, 2013.

[47] M. E. Rasekh, Z. Marzi, Y. Zhu, U. Madhow, and H. Zheng. Noncoherent mmwave path tracking. In *Proceedings of the 18th International Workshop on Mobile Computing Systems and Applications*, pages 13–18, 2017.

[48] M. Safe, J. Carballido, I. Ponzoni, and N. Brignole. On stopping criteria for genetic algorithms. In *Brazilian Symposium on Artificial Intelligence*, pages 405–413. Springer, 2004.

[49] R. Schreiber. Implementation of adaptive array algorithms. *IEEE transactions on acoustics, speech, and signal processing*, 34(5):1038–1045, 1986.

[50] R. Shore. Nulling a symmetric pattern location with phase-only weight control. *IEEE Transactions on Antennas and Propagation*, 32(5):530–533, 1984.

[51] G. H. Sim and J. Widmer. Finite horizon opportunistic multicast beamforming. *IEEE Transactions on Wireless Communications*, 16(3):1452–1465, 2016.

[52] S. T. Smith. Optimum phase-only adaptive nulling. *IEEE Transactions on Signal Processing*, 47(7):1835–1843, 1999.

[53] H. Steyskal. Synthesis of antenna patterns with prescribed nulls. *IEEE Transactions on Antennas and Propagation*, 30(2):273–279, 1982.

[54] H. Steyskal. Simple method for pattern nulling by phase perturbation. *IEEE Transactions on Antennas and Propagation*, 31(1):163–166, 1983.

[55] C.-S. Sum, Z. Lan, R. Funada, J. Wang, T. Baykas, M. Rahman, and H. Harada. Virtual time-slot allocation scheme for throughput enhancement in a millimeter-wave multi-gbps wpan system. *IEEE Journal on Selected Areas in Communications*, 27(8):1379–1389, 2009.

[56] S. Sur, X. Zhang, P. Ramanathan, and R. Chandra. Beamspy: enabling robust 60 ghz links under blockage. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, pages 193–206, 2016.

[57] R. Vescovo. Null synthesis by phase control for antenna arrays. *Electronics Letters*, 36(3):198–199, 2000.

[58] T. Vu. Simultaneous nulling in sum and difference patterns by amplitude control. *IEEE transactions on antennas and propagation*, 34(2):214–218, 1986.

[59] T. Wei and X. Zhang. Pose information assisted 60 ghz networks: Towards seamless coverage and mobility support. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, pages 42–55, 2017.

[60] D. Whitley. A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85, 1994.

[61] Y. Yang, H. S. Ghadikolaei, C. Fischione, M. Petrova, and K. W. Sung. Fast and reliable initial access with random beamforming for mmwave networks. *arXiv preprint arXiv:1812.00819*, 2018.

[62] D. Z. Yong Deng, Yang Liu. An improved genetic algorithm with initial population strategy for symmetric tsp. *Mathematical Problems in Engineering*, pages 1–6, 2015.

[63] J. Zhang, X. Ge, Q. Li, M. Guizani, and Y. Zhang. 5g millimeter-wave antenna array: Design and challenges. *IEEE Wireless communications*, 24(2):106–112, 2016.

[64] R. Zhao, T. Woodford, T. Wei, K. Qian, and X. Zhang. M-cube: a millimeter-wave massive mimo software radio. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, pages 1–14, 2020.

[65] A. Zhou, L. Wu, S. Xu, H. Ma, T. Wei, and X. Zhang. Following the shadow: Agile 3-d beam-steering for 60 ghz wireless networks. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 2375–2383. IEEE, 2018.

[66] A. Zhou, X. Zhang, and H. Ma. Beam-forecast: Facilitating mobile 60 ghz networks via model-driven beam steering. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2017.

A Appendix

A.1 Pseudocode

Here we present pseudo codes to Nulli-Fi's algorithms discussed in section 4.

Algorithm 1: OPTIMALNULLING($N, \phi, \phi_0, d, \lambda, \alpha^*$)

```
$\theta \leftarrow 2\pi \frac{d}{\lambda}(cos(\phi) - cos(\phi_0))$;
$v_P \leftarrow \exp(j\frac{N-1}{2}\theta)$;
for n in range($N$):
    $v_n \leftarrow \exp(jn\theta)$;
    $\Delta\alpha_n \leftarrow 0$;
for $n$ in range($\frac{N}{2}$):
    if $|\angle(v_P, v_n) - \pi| < \alpha^*$:
        $\Delta\alpha_n = \pi - \angle(v_P, v_n)$;
    $\Delta\alpha_{N-n-1} = \angle(v_P, v_n) - \pi$;
    elseif $\angle(v_P, v_n) < \pi - \alpha^*$:
        $\Delta\alpha_n = \alpha^*$;        $\Delta\alpha_{N-n-1} = -\alpha^*$;
    elseif $\angle(v_P, v_n) > \pi + \alpha^*$:
        $\Delta\alpha_n = -\alpha^*$;        $\Delta\alpha_{N-n-1} = \alpha^*$;
    if $\angle(v_P, \sum_{n=0}^{N-1} v_n \exp(j\Delta\alpha_n)) \neq 0$:
        return 0;
return $|\sum_{n=0}^{N-1} v_n \exp(j\Delta\alpha_n)|$;
```

Algorithm 2: CHOOSESUBSET(N, q)

```
S $\leftarrow \emptyset$;
$(\alpha_0, \cdots, \alpha_{N-1}) \leftarrow$ ideal phase shifts for main lobe;
for $n$ from 0 to $N-1$:
    if $\alpha_n$ is within $t$ degs of an available phase shift:
        add $n$ to $S$;
return $S$;
```

A.2 Phase Calibration

Here we explain Nulli-Fi's phase calibration in detail. In order to calibrate for the difference in the lengths of the wires coming out of each antenna element, we pick one reference antenna element i^*, and calibrate the remaining antennas with respect to this reference. Note that, the process of calibration is finding the additional phase shift one has to apply to antenna j in order to bring it in phase with the reference antenna i^*. To do so, we run a series of simple experiments as follows. We note that throughout the all of these experiments, transmitter and receiver are directly facing each other.

- First, for a fixed i, $0 \leq i \leq 15$, we turn off all antenna elements except i. We then apply phase shifts to the weight of antenna element i over time to cover all the possible phase values, and capture the received power over time. For antenna element i and its nth phase shift $\alpha_i[n]$, we denote the received signal amplitude by $a_i[n]$.
  We repeat this for all i from 0 to 15.
- We repeat the previous phase for all antennas i save for a chosen reference, i^*. Throughout these experiments, we keep element i^* turned on with a constant amplitude a_0, and for the experiment with element i and nth phase shift, we call the corresponding received amplitude $b_i[n]$. Now $b_i[n]$ is the sum of the signals received from i^* and i. If there is a α_{i,i^*} phase shift between the two elements, then we must have

$$b_i(t) = |a_i(t) + e^{\alpha_{i,i^*}} a_0|,$$

Therefore we can find α_{i,i^*} by performing a simple binary search over all possible values in $[0, 360]$ degrees. An example of the two normalized curves, before and after calibration, is shown in Figure 17.

Algorithm 3: NULLI-FI-GENETIC

```
Initialize $\mathscr{A} = \{A_1, \cdots, A_M\}$ using
OPTIMALNULLING($N, \phi, \phi_0, d, \lambda, \alpha^*$);
if $q \leq 3$:
    $S \leftarrow$ CHOOSESUBSET($N, q$);
else:
    $S \leftarrow \{1, 2, \cdots, N\}$;
Limit the adaptive elements to $S$;
while not converged:
    for i from 1 to $M$:
        $f_i \leftarrow F(A_i)$;
    sort $A_i$ according to $f_i$;
    keep $\mathscr{A} = \{A_1, \cdots, A_{\eta M}\}$ and discard others;
  while $|\mathscr{A}| < M$:
        Randomly choose two chromosomes $A_i$, $A_j$;
        perform CROSSOVER($A_i$,$A_j$);
        Randomly mutate some $A$'s with prob. $p_m$;
output $A_1$.
```

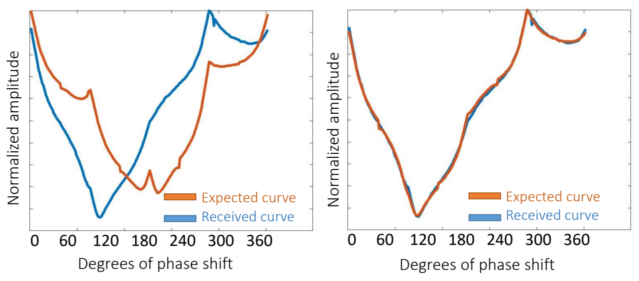


Figure 17: Expected versus measured power of two antenna elements before and after calibration.

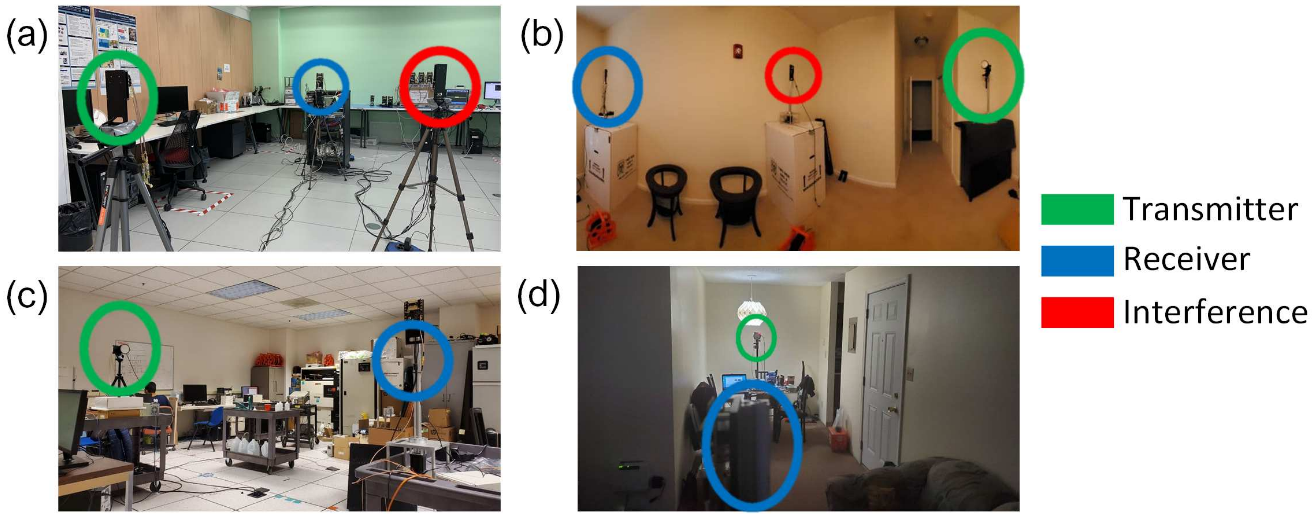


Figure 18: Different locations where we ran our experiments.

A.3 Experimental Setup

Experiment Locations. We ran our experiments in 4 different rooms shown in Fig. 18. Two locations (a, c) were inside a lab environment with many metal cabinets that contributed to multi path reflections. The other 2 locations (b, d) were different rooms inside apartments with many indoor objects. In all rooms, there were human subjects in the background during the experiment, thus constituting dynamic environments.

Nulli-Fi + mm-Flex. One transmitter/receiver pair are implemented using a single FPGA device in a full-duplex manner, i.e. transmitter and receiver functionalities are used at the same time. This pair corresponds to the transmitter and the receiver implementing Nulli-Fi.

We use a second FPGA (mounted on the same hosting chassis) which serves as the interferer. Since both FPGAs are mounted on the same chassis, long cables (5m) are used to carry the baseband signals to the corresponding transmitting antennas (the one transmitting the packets of interest and the one from the interferer). Therefore, with this setup, we are able to easily cover indoor scenarios.

Both FPGAs are managed from a control and management processor integrated in the same hosting chassis. This is used to send/receive frames to/from each baseband processor, configure ADCs/DACs, IP blocks, as well as the setup for the 60GHz Siversima RF-frontends.

A.4 Further Analysis of Nulling Performance

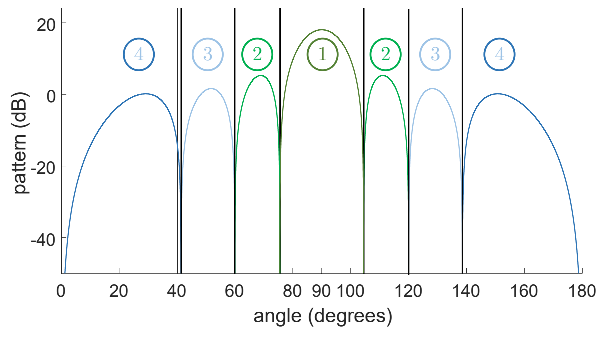
Closed Form Solutions. Alg. 1 offers a step by step solution to find nulls. It is also possible to find closed form solutions for bounds of achievable nulling performance as a function using the algorithm. This can be done by going thorough the algorithm with by keeping the symbol ϕ as opposed to setting it to a specific value. Doing so will result in explicit formulas for the angles for which perfect nulling is possible. For the angles that perfect nulling is not possible, we can find explicit formulas that determine the deepest possible nulls as a Piecewise-defined function of the angle ϕ. Different cases of this piecewise-defined function are separated by the naturally occurring nulls in the original beam pattern. An example of this for $N = 8$ antennas is shown in Fig. 19(a) where there are four cases separated by natural nulls, with each case having its own piecewise formula. For example, Theorems A.1 and A.2 show examples of closed form solutions for nulling around the main lobe as a function of number of elements N, angle of nulling ϕ, the main lobe angle ϕ_0, and the maximum phase shift allowed on each antenna α^*:

Theorem A.1 *The two closest perfect nulls to the main lobe given a maximum phase shift of α^* for each element are given by* $\phi^* = \arccos\left(\cos(\phi_0) \pm \frac{\lambda}{Nd}(1-\frac{2\alpha^*}{\pi})\right)$.

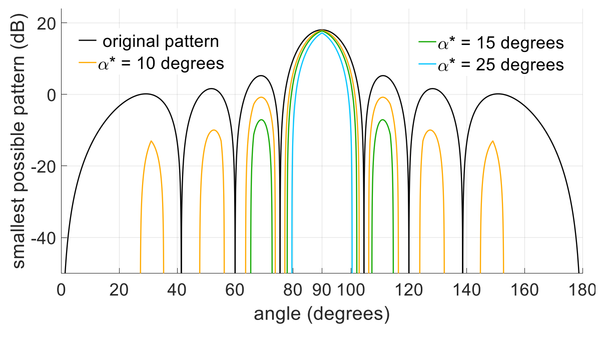
Theorem A.2 *For the area around the main lobe that perfect nulling is not possible, the deepest possible null at direction ϕ is given by* $N\cos(\frac{N}{4}\theta+\alpha^*)$, *where* $\theta = \frac{2d}{\lambda}(\cos(\phi)-\cos(\phi_0))$.

Specifically, Theorem A.1 determines the areas where perfect nulling is possible, and Theorem A.2 determines the deepest possible nulls for angles where perfect nulling cannot be achieved. For $N = 8$, these formulas correspond to the case 1 in Fig. 19(a). Following similar methods demonstrated in the proofs of these theorems in section A.5 we can find explicit formulas for other cases too.

Using the closed form formulas, we have plotted the best achievable nulling performance (i.e., the lowest possible value of the pattern P for each angle) for $N = 8$ antennas, and $\alpha^* = 10, 15$ and 25 degrees in Fig. 19(b). As revealed by



(a)



(b)

Figure 19: (a) The closed form solutions for the single null problem are piecewise-defined functions, with the cases separated by the nulls naturally occurring in the original pattern. Here there are four color-coded cases each corresponding to their own explicit formulas. For instance, Theorems A.1 and A.2 determine the best possible nulling for case 1. (b) Best achievable nulling performance for $N = 8$ elements are depicted for different angles and values of α^*. when a curve is not present at an angle, perfect null (i.e. $P = 0$) is achievable there.

the figure, there is a trade-off between how much we lose in the main lobe, and how strongly we can null different angles. For instance, while $\alpha^* = 15^\circ$ ensures a maximum main lobe loss of 0.3 dB, there are certain regions depicted by the green curve that cannot be nulled. As can be seen, for lower α^* (orange curve) there are more regions that cannot be nulled, while a higher α^* (blue) shows only an area around the main lobe that cannot be nulled. Since Alg. 1 is optimal, we believe it can help decide the degree of trade-off in different applications. This is especially useful as these curves also define as a stopping criterion for our algorithms especially when we lump in the hardware imperfections into the optimization problem. For example if we know that it is not possible to get a nulls stronger than 20 dB in the ideal case (i.e., without hardware imperfections), we can expect that with hardware imperfections the nulling performance cannot get far beyond 20 dB, as we explain in section 4.2.

Alternative Algorithms. In the walk-through example with 6 antennas in section 4.1, the final configuration of the antennas is shown in Fig. 4 (b4). As can be seen from the figure, pairs $\{0,3\}$, $\{1,4\}$ and $\{2,5\}$ are perfectly canceling each other, yielding a null. Looking at this configuration, one might wonder if we can always try and create *pairs* of opposing vectors, such that the sum of every pair is zero. However, it is possible to construct examples where this solution does not work, but Alg. 1 achieves a perfect null. In fact, for larger values of N, it is possible to construct examples in which no set of $K < N$ vectors sum to zero while the sum of all N vectors is still zero. For further information, we refer the interested reader to our git repository where we have implement and compare these algorithms.

A.5 Proofs

Proof of lemma 4.1. We align the main lobe toward some angle ϕ_0, and therefore the signal coming from that angle will sum up coherently. Specifically,

$$|P|^2 = \left|\sum_{n=0}^{N-1} e^{-2\pi j \frac{d}{\lambda} n\cos(\phi_0)} e^{j\alpha_n}\right|^2 = \left|\sum_{n=0}^{N-1} 1\right|^2 = N^2.$$

Imposing additional phase shifts $\Delta\alpha_n$ in order to enable nulling would give us:

$$\begin{aligned}|P'|^2 &= \left|\sum_{n=0}^{N-1} e^{j\Delta\alpha_n}\right|^2 \\ &= \left(\sum_{n=0}^{N-1}\cos(\Delta\alpha_n)\right)^2 + \left(\sum_{n=0}^{N-1}\sin(\Delta\alpha_n)\right)^2 \\ &\geq \left(\sum_{n=0}^{N-1}\cos(\alpha^*)\right)^2 + 0 = N^2\cos^2(\alpha^*)\end{aligned}$$

since $|\Delta\alpha_n| \leq \alpha^* \leq 90^\circ$. Hence, $|P'|^2 \geq |P|^2\cos^2(\alpha^*)$ and the loss in the main lobe power is at most $1 - cos^2(\alpha^*) = sin^2(\alpha^*)$.

Proof of lemma 4.2. Replacing θ, we get $v_n = e^{jn\theta}$. Since they are unit vectors, v_k and v_{N-1-k} are symmetric around their sum. Further, we have

$$\begin{aligned}\angle(v_k + v_{N-i-k}) &= \angle(e^{jn\theta} + e^{j(N-1-n)\theta}) \\ &= \angle\Big(e^{j\frac{N-1}{2}\theta} \times 2\cos(\frac{N-1-2n}{4})\Big) \\ &= \angle e^{j\frac{N-1}{2}}\theta + \angle\cos(\frac{N-1-2n}{4}\theta) \\ &= \frac{N-1}{2}\theta \pm \pi,\end{aligned} \tag{3}$$

where the last line follows from the fact that the phase of a real number is either 0 or π. Since the phase of these vector pair sums are the same (up to $\pm\pi$), so is the sum of all of them, P. This concludes the proof of the lemma.

Proof of Theorem 4.3. For a given nulling angle ϕ, we identify two possible cases. First, if it is not possible to create a perfect null at ϕ, and second, if it is possible to create a perfect null at ϕ.

- In the first case, Alg. 1 does not stop until all vectors $v_0, \cdots, v_{N-1}$ have rotated by $\pm\alpha^*$. In this case, following the exact same argument in the proof of Theorem 4.4, Eq. 6 hold with equality, which means that the best nulling performance is achieved.
- In the second case Alg. 1 where nulling is possible, at some point in the algorithm, $\nu = \angle(v_P, \sum_{n=0}^{N-1} v_n \exp(j\Delta\alpha_n)) \neq 0$ should at some point return π. Otherwise, it remains 0 until the end, in which case nulling should not be possible, contradicting our assumption. Therefore, a some point in the algorithm, $\nu \neq 0$, so the output of the algorithm will be 0, meaning it predicts a perfect null.

In both cases, the output of the algorithm gives the best nulling performance, proving that the Alg. 1 is optimal.

Proof of theorem A.1. For a given ϕ, assume an x-y coordinate for the complex plane, such that $\angle P(\phi) = 0$. In this coordinate, let each vector v_k have the representation (x_k, y_k). We are looking for the first possible ϕ for which there exists a set of additional phase shifts, $\Delta\alpha_k$, such that $P(\phi) = (0,0)$. In its general form, P is expressed as

$$\begin{aligned}
P(\phi) &= \sum_n v_n e^{j\Delta\alpha_n} \\
&= \Big(\sum_n \cos((n - \frac{N-1}{2})\theta + \Delta\alpha_n), \sum_n \sin((n - \frac{N-1}{2})\theta + \Delta\alpha_n)\Big) \\
&= \Big(\sum_n x_n, \sum_n y_n\Big),
\end{aligned} \tag{4}$$

where θ is defined according to section 4.1. Note that

$$\begin{aligned}
x_n^* &:= \min\{\cos((n - \frac{N-1}{2})\theta + \Delta\alpha_n) \mid -\alpha^* \leq \Delta\alpha_n \leq \alpha^*\} \\
&\in \{-1, \cos((n - \frac{N-1}{2})\theta \pm \alpha^*)\}.
\end{aligned} \tag{5}$$

Further, we can bound the absolute value of the pattern P as follows.

$$\begin{aligned}
|P(\phi)|^2 &= \big(\sum_n x_n\big)^2 + \big(\sum_n y_n\big)^2 \\
&\geq \big(\sum_n x_n^*\big)^2,
\end{aligned} \tag{6}$$

where we have bounded the second term with zero. This inequality holds as long as $\sum_n x_n^*$ is positive, which is true around the main lobe, before the first possible null.

Let us rotate each vector v_n to get x_n^* as its x component. Using lemma 1, v_n rotates by $\pm\alpha$ if and only if v_{N-1-n} is rotated by $\mp\alpha$. This means that the two vectors remain symmetrical around the x axis. Therefore, we will necessarily have $\sum_n y_n = 0$, bringing equation 6 to an equality. Hence, as long as $\sum_n x_n^* > 0$, nulling is not possible.

The first point at which nulling becomes possible can therefore be derived by finding the solution to $\sum_n x_n^* = 0$. Using equation 5 combined with lemma 1, we get

$$\sum_{n=0}^{N-1} x_n^* = 2 \sum_{n=0}^{\frac{N}{2}-1} \cos((n - \frac{N-1}{2})\theta + \alpha^*) = 0, \tag{7}$$

The solution to which is $\theta = \pm\frac{2}{N}(\pi - \alpha^*)$, or its corresponding ϕ value given in the theorem.

Proof of Theorem A.2. Using Theorem 4.3, we have to run the output of the algorithm for the assumptions in this theorem. Since nulling is not possible, the algorithm will run from 0 to $N-1$, yielding vectors $\cos(n\theta + \alpha^*)$ for $0 \leq n \leq \frac{N}{2} + 1$, and $\cos(n\theta - \alpha^*)$ for $\frac{N}{2} + 1 \leq n \leq N-1$. Summing them up, we get the result in stated in the theorem.

SyncScatter: Enabling WiFi like synchronization and range for WiFi backscatter Communication

Manideep Dunna, Miao Meng, Po-Han Wang, Chi Zhang, Patrick Mercier, Dinesh Bharadia

University of California, San Diego

Abstract

WiFi backscattering can enable direct connectivity of IoT devices with commodity WiFi hardware at low power. However, most existing work in this area has overlooked the importance of synchronization and, as a result, accepted either limited range between the transmitter and the IoT device, reduced throughput via bit repetition, or both. In this paper, we present SyncScatter, which achieves accurate synchronization with incident signals at the IoT device level while realizing maximum possible sensitivity afforded by a backscatter link budget. SyncScatter creates a novel modeling framework and derives the maximal optimal range and synchronization error that the receiver can tolerate without significant performance compromises. Next, SyncScatter builds a novel hierarchical wake-up protocol, which, together with a custom ASIC, achieves a range of 30+ meters and the peak throughput of 500Kbps, with an average power consumption of 30μW.

1 Introduction

Ubiquitous wireless network coverage is required to enable the next-generation of Internet of Things(IoT) devices. In smart homes, offices, industrial environments, and more, WiFi is by far the most ubiquitous form of connectivity. However, enabling WiFi connectivity at the IoT device level requires high power consumption - to the point where most such IoT devices must be either plugged into wall power, must use large and frequently re-charged batteries, or simply cannot afford to transmit data at high average throughput [13, 22].

Recent work has proposed using backscatter communication techniques to reduce power, which forgoes direct WiFi signal generation by instead modulating data on top of ambient WiFi transmissions generated by existing access points (APs) [10, 40]. There are three components to backscattering systems: 1) the transmitting radio which generates the excitation signal; 2) the IoT device which reflects the incoming signal by encoding its information, and 3) a receiving radio (WiFi-compatible) which receives the packets and decodes the data from the IoT device. By avoiding any signal generation or amplification directly at RF, the power consumption of backscatter communication can be low - on the order of microwatts - such that small energy harvesters or batteries can directly power the IoT devices.

Existing work on WiFi-backscattering can be broadly classified into two major categories: the first set of WiFi-backscattering is inspired by the RFID style of backscatter communication systems [8, 21], wherein a tone-generator is deployed as the excitation radio. This gives the IoT device the freedom to begin backscattering at any time it likes, along with the freedom to create any backscattered waveform like WiFi to encode its information. This tone-based backscattering approach requires additional custom hardware - the tone generator - beyond commodity WiFi hardware, and thus deployment timelines and costs can be appreciable [21].

In contrast, the second set of works [40, 42] leverage existing WiFi infrastructure for both the excitation and receiving radios, therefore requiring no additional infrastructure deployment. In this approach, an existing WiFi radio's transmission is used as excitation signal, and the IoT device modulates the underlying data in the excitation signal in a specific manner according to the IoT device's data which is then reflected back to the environment. Properly backscatter modulated signals will retain a WiFi-compatible form, and therefore can be decoded by another WiFi radio. Ideally, the IoT device should synchronize itself to the incident signal, such that it knows precisely when to perform backscatter modulation. To ensure the superimposed data (i.e., the data from the incident source and the IoT device data) is readable by a commodity WiFi receiver, this synchronization should be down to the symbol level. Unfortunately, no prior work in WiFi backscatter synchronizes down to the symbol level [20, 21, 40].

Synchronizing to the symbol level with high-accuracy is challenging, as all ISM, including WiFi transmissions, are made up of complex digital waveforms at fairly high data rates. A large literature has worked on optimizing the power consumption of the synchronization routines in WiFi transceivers, which still takes considerable power [5]. Furthermore, the power consumption of synchronization increases exponentially to make it work at lower incoming signal power. Therefore, prior WiFi backscattering work such as Hitchhike [40] employs a very simply energy-detecting synchronization scheme that consumes low power but can only synchronize to an accuracy of 2μsec at an input power of $-20dBm$. Since the symbol rate in 802.11b WiFi is 1μsec, this means Hitchhike will effectively begin backscatter modulation at a random location within a symbol, which ultimately limits its achievable distance to be no more than 6m from the transmitter. Unfortunately, this comes directly at the cost of decreased achievable throughput, increased inter-symbol interference(ISI), and ultimately reduced communication range.

In this paper, we present SyncScatter, which is the first

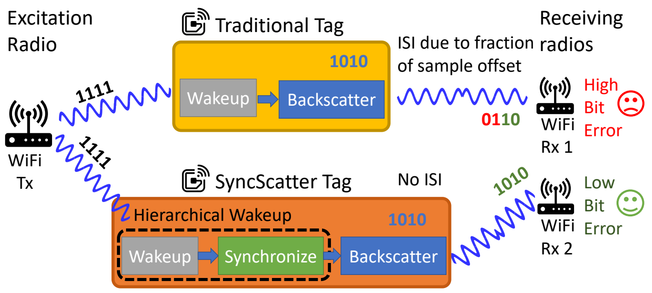


Figure 1: Shows a traditional tag form [40] which backscatter signal without accurate synchronization, leading to higher BER. In contrast the SyncScatter tag accurately synchronizes to the incoming signal.

integrated circuit-based backscattering platform that can enable symbol-level synchronization through a hierarchical wakeup and synchronization protocol, shown in Fig. 1, which works up to theoretical sensitivity levels. Such symbol synchronization enables longer range, higher-throughput, and more reliable backscatter communication than prior art for all forms of communication (not just WiFi). Here, we specifically built and prototyped SyncScatter to demonstrate the first fully-WiFi-compatible symbol-level synchronized, long-distance, extremely low-powered backscatter system. Furthermore, SyncScatter can support multiple IoT devices to co-exist without interfering with each other. SyncScatter is designed on a custom ASIC, enabling ultra-low-power consumption.

In order to bound the design space, we begin the paper by asking some fundamental questions: how accurate does the synchronization need to be? How far away from the transmitter can we work? What sensitivity level should we target at that range? How do we do all of this while keeping power low? To answer the first question, we present analysis wherein we add increasing amount of synchronization error and observe the performance (SNR vs BER), and choose point where the gains are incremental beyond it. Then, we leverage FCC specifications on maximum available transmit power along with the minimum SNR required to decode a certain ISM backscatter, accounting for loss in backscattering and the noise figure of the corresponding receiver, to derive the maximum path loss, and therefore distance, a backscatter system could potentially work at. This analysis also gives us the sensitivity level needed at the backscattering tag, which as we will show turns out to be $-35dBm$. Importantly, we discuss how improving sensitivity beyond this level is wasteful. This is the first analysis of its kind that combines path loss, synchronization accuracy, and sensitivity.

The next natural question for SyncScatter is how to achieve the required tight synchronization accuracy and desired high sensitivity while consuming microwatts of power? To keep power low, a direct envelope-detector (ED) approach is typically used; however, such a system's sensitivity typically reduces with increasing bandwidth. Since achieving a high synchronization accuracy requires high bandwidth, this poses a direct trade-off between accuracy and sensitivity. The only way to break this trade-off is to add RF amplification before the ED. However, this can cost significant power - upwards of 100s of microwatts.

To break this trade-off, our key insight is to create a two-stage, hierarchical wake-up and synchronization protocol, wherein a first stage (the wake-up receiver) is designed with single-digit microwatt power and leverages low-bandwidth energy detection to simply wake-up the tag at approximately the right time, at which point a second stage (the synchronization receiver) uses higher-power active RF amplification to enable the desired sensitivity at the desired bandwidth, but is turned on only for a short time to synchronize, and is powered down immediately post synchronization. SyncScatter creates a new protocol where two packets with controlled length are sent apriori to backscattering. The time duration of the two packets encodes the tag's identity, which results in an enable signal from the first stage wake-up receiver. The second stage turns on just before the start of the backscatter payload packet, samples the incoming signal at high bandwidth, looking for the beginning of the packet and the symbol boundary, and then promptly goes to sleep. Once symbol-level synchronization is achieved, the backscatter modulation logic reflects the incoming signal by overlaying its data in a synchronized fashion.

SyncScatter specifically builds an RF integrated circuit and hardware design for the entire hierarchical wake-up protocol, along with single-sideband backscattering circuits, which can backscatter any ISM 2.4 GHz signals. SyncScatter's WiFi transmitter and receiver are implemented using open-wrt [25] on TP-Link devices. SyncScatter is evaluated in indoor office environments to achieve the following results:

- SyncScatter achieves a sensitivity of up to -35 dBm via the custom integrated circuit, with a synchronization accuracy of 150 ns, which enables a 30+ meter link operation as measured in a regular office environment. As a result, the longer wake-up distance offered by SyncScatter allows the use of WiFi APs deployed in a typical home or office environment without requiring additional smartphones, unlike in HitchHike [40].
- SyncScatter tag enables symbol-level synchronization at very low power consumption by utilizing a hierarchical wake-up receiver with a false negative rate of 10^{-3}.
- SyncScatter tag supports backscatter communication over a wide range of the transmitter(Tx) to tag and receiver(Rx) to tag distances whose product is $\leq 169\ m^2$, i.e., 13m from Tx and 13m from Rx or 33m from Tx and 5m from Rx.
- SyncScatter supports multiple tags running concurrently and supports 802.11b waveforms, modulating at symbol level providing peak bit-rates of 500 Kbps.

2 Synchronization & Sensitivity Requirements

This section introduces a model that helps scope out requirements for the custom IC designed for WiFi backscattering. The methodology presented herein could also be used to design any ISM backscatter systems.

2.1 Synchronization Requirements for WiFi

Synchronization is at the heart of all communication systems, and must be thought through carefully, even for backscatter systems (at least, ones that do not use tone generators, like fully WiFi-compatible systems). In this sub-section, we specifically discuss the need for synchronization and establish the minimum required synchronization accuracy, which is needed for minimal performance reduction in a fully WiFi-compatible backscatter system.

To describe the synchronization problem we are solving, we will start with a brief discussion of the problem with a figure, specifically on a representative 802.11b signal in Fig. 2. To backscatter a valid 802.11b signal, the tag performs code-word translation on the ambient 802.11b packets, similar to [40]. The tag embeds its data on the ambient 802.11b WiFi packet by changing the phase of each of the 802.11b symbols in the packet. The resulting backscattered signal is a product of the incident signal and the tag's phase modulation. Therefore, changing each symbol's phase ensures that the chip sequence on each symbol retains the 11-bit barker code and backscatters a valid 802.11b signal, which is then decodable by the WiFi receiver without inter-symbol interference shown on the right.

As observed in [40], the HitchHike tag, and in fact all past work that backscatters ambient signals, do not accurately synchronize with the incoming transmitter signal and therefore applies code-word translation with incorrect boundaries shown in the Fig. 2 left. Misalignment between the backscatter symbol timing and the original symbols in the 11b packet will start to change the barker code, which hurts the signal to interference ratio and, therefore, hurts the receiver's ability to decode the backscatter packet and can result in errors.

To quantitatively understand the impact of synchronization, we emulate the backscatter system wherein we transmit the reflected packet using an SDR while intentionally adding an increasing amount of synchronization error from 0 ns to 625 ns as shown in Fig. 3a. The synchronization errors are added after the header of the transmission, as the backscattered payload is applied only after the header. The transmission is received using an off-the-shelf WiFi AP. We measure 10000 packets with backscatter payload in a total of 1 million bits to generate the SNR to BER plot. As we can observe, synchronization errors up to 150ns do not lead to significant BER degradation. However, errors beyond this can lead to 7dB or more degradation.

A natural question is how did past work resolve this issue. Due to power-related issues that we will discuss shortly, prior work such as [40] achieved synchronization accuracies of anywhere from 1 to 10 μsec, which is well beyond the symbol period. This means that the backscattered signal modulated on top of the existing WiFi packet will have a random phase offset for each packet. To combat this problem, [40] uses a

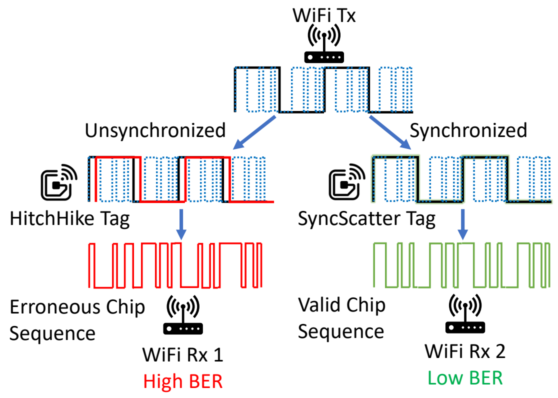
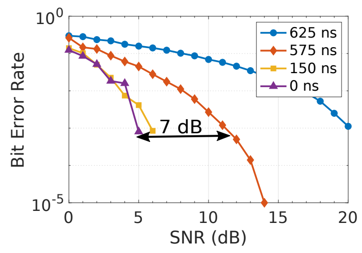


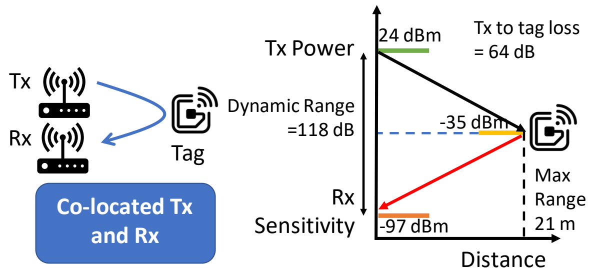
Figure 2: Impact of in-accurate synchronization, and resulting loss in signal quality at the receiver.

preamble to help the receiver find the start of backscatter data in the packet and decode the tag data. Further, to ensure proper decoding of tag data, each tag data bit is repeated multiple times(repetition coding), reducing the available throughput. Said differently, it would lose significant SNR gain due to lack of synchronization. A more severe consequence is that the CRC(cyclic redundancy check) of the packet often fails with past work. Instead, we can ensure CRC checks can be met with proper synchronization, enabling more WiFi cards as receivers while simultaneously enabling higher average channel throughput.

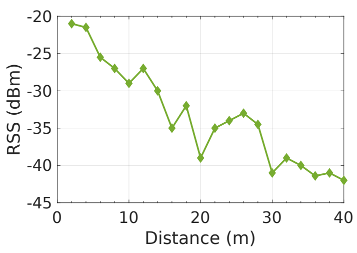
Generalization of Synchronization requirements to other wireless standards: An obvious next step is to know the synchronization requirements for backscatter systems based on different wireless standards like BLE(Bluetooth low energy) and 802.11g WiFi. In BLE transmissions, a data bit 0/1 is encoded as different frequency modulated sine tone signals. Backscatter tag modifies the frequency of sine tones present in the BLE symbols to encode backscatter data on top of BLE packets. The backscatter encoder must know the symbol boundaries to ensure that backscattering is successful. Otherwise, the backscatter data is spread on two consecutive symbols, and the receiver fails to decode the packet. Similarly, 802.11g WiFi backscatter systems encode data on top of an OFDM symbol by changing the signal phase. Hence the backscatter encoder must be aware of the OFDM symbol boundaries. To understand the synchronization delay's impact, we perform Matlab simulations of backscattering BLE and 20MHz standard OFDM signals as given in [42] and introduce synchronization(sync) delay while finding symbol boundaries. Our simulations reveal that BLE backscatter loses 4 dB SNR at 10^{-3} BER for 100 ns sync delay and can tolerate up to 50 ns sync delay(More details in Appendix 9). On the other hand, OFDM backscatter can handle up to 1000 ns sync delay and lose more than 10 dB SNR at 10^{-3} BER for 1200 ns sync delay. These observations suggest that symbol-level synchronization is essential for backscatter systems based on other protocols as well.



(a) BER-SNR curve



(b) Co-located WiFi Tx and WiFi Rx with a SyncScatter tag



(c) Rx Signal Strength with Tx-Tag distance

Figure 3: (a) BER-SNR curve for different amount of synchronization errors, **(b)**Maximum distance analysis and operating point for IoT device sensitivity assuming a co-located Tx and Rx, **(c)** Received Signal Strength (RSS) measured at the SyncScatter tag vs. distance.

2.2 Sensitivity of the backscatter tags

For an IoT device to backscatter its data, it needs to detect the incoming signal from the excitation radio. Prior work has shown very short distance backscatter from excitation radio to the tag, for example, less than 6 meters for WiFi backscattering [20, 40]. Given the flexibility in the design parameters with integrated circuit design, we would like to understand better the sensitivity for which the backscatter tag should be designed to optimize for range while minimizing power consumption.

At the outset, the above question looks ill-posed. To simplify the above question and perhaps get a more coherent answer, let us take an example of the same 11b signals transmitted at 1 Mbps. To simplify the above analysis, we would assume the worst-case mono-static backscatter communication scenario, i.e., the transmitting radio and the receiving radio are co-located. The challenge with the backscatter system is that it suffers from two-way path loss. Specifically, the transmitter to the tag suffers a path loss of $\frac{1}{d^2}$. Then the signal is re-radiated back from tag to the receiver, which suffers a multiplicative path loss of $\frac{1}{d^2}$, which implies $\frac{1}{d^4}$ path loss (or additive in dB). Said differently, it would suffer the same loss as traveling quadratic distance.

With that analysis in place, we can leverage the maximum possible dynamic range by operating at the highest possible transmit power and minimum receiver sensitivity at which we can decode the packet. For WiFi at 2.4GHz, the maximum average transmit power is 24 dBm (11b has a peak power of 30 dBm and 6 dB of PAPR), while a receiver sensitivity to decode 1 Mbps is at -97 dBm as shown in Fig. 3b. Given a tag can have ideal insertion loss of 3dB, this provides a dynamic range of 24 dBm $-(-97$ dBm$) - 3$ dB $= 118$ dB for decode-ability. Assuming a worst-case range condition where the tag communicates with a co-located transmitter and receiver AP, this suggests operation with 59 dB of one-way path loss, for a worst-case incident signal power 24 dBm -59 dB $= -35$ dBm.

To find the path loss in indoor environments, we measure the received signal strength with increasing distance between the transmitter and IoT device, up to a point where the received signal strength at the IoT device goes below the threshold of -35 dBm, providing us the range up to which we expect the backscatter to work. Fig. 3c plots the received power with increasing distance, and received power or sensitivity of -35 dBm would provide the maximum range benefit, i.e., from 15 – 21 meters, one-sided range. Note that prior systems have achieved -15 dBm sensitivity, with 6 meters working range [40].

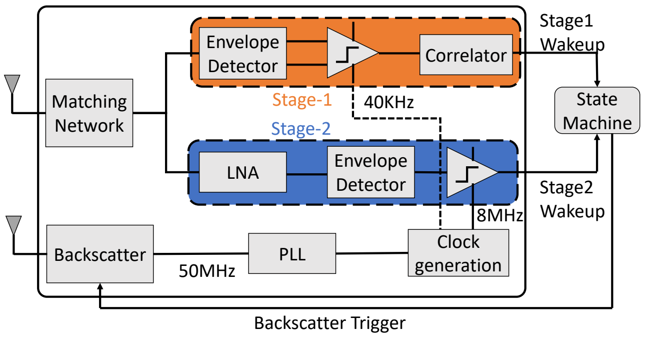


Figure 4: Design Overview of SyncScatter.

In summary, we need to achieve a sensitivity of -35 dBm beyond which the gains are incremental and concurrently achieve synchronization accuracy of 150 ns, all while keeping low power. Achieving this specification at the micro-watt level is extremely challenging.

3 SyncScatter Tag Design Overview

In this section, we describe the architecture of SyncScatter's tag and show how it can wake-up to properly-designed incident WiFi signals, perform symbol-level synchronization to said incident WiFi signals, and then perform backscatter modulation. This is accomplished by three separate subsystems: a wake-up receiver, a sync stage edge-detector, and a backscatter modulator as shown in Fig. 4.

In most prior works [40,42], the wake-up receiver and sync stage are combined into a single circuit. i.e., a single circuit is responsible for determining if an appropriate WiFi signal is incident on the tag and then indicating to the tag when to begin backscatter modulation. This is, however, a problematic approach if the tag desires low-power and sufficient sensitivity and synchronization accuracy, as these items all trade-off directly with one another. Thus, breaking wake-up functionality apart from synchronization functionality via the proposed hierarchical approach can serve to break this trade-off, enabling a low-power yet sufficiently sensitive and accurate design.

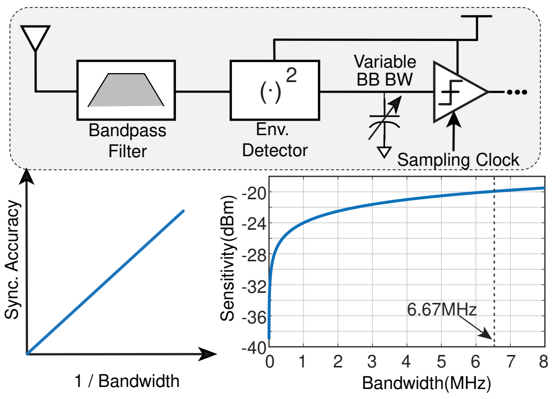


Figure 5: Analysis on Direct-Envelope Detector (ED) Structure

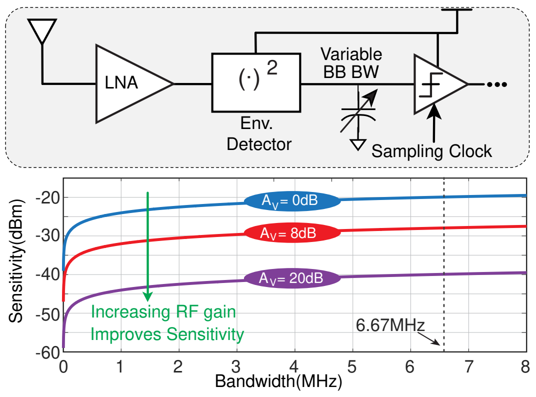


Figure 6: LNA Based ED Improvements

3.1 Sync Stage Receiver

As aforementioned, the best we can do given FCC WiFi transmitter power limitations and the best achievable WiFi receiver noise figure is to achieve a sensitivity at the tag of approximately −35 dBm. Achieving a sensitivity better than this does not improve the system's performance or range in any meaningful manner and is thus just wasteful. At the same time, we need to detect the symbol boundary of an incident WiFi packet with an accuracy of at least 150 ns. Again, doing better than this does not meaningfully improve performance. Thus, the design space here is: achieve these sensitivity and synchronization accuracy specifications while consuming as little power as possible.

To find the accurate symbol boundaries, we can use the fact that multiple symbols together constitute a packet. If we can somehow find the exact time instant at which the packet starts, we can determine the symbol boundaries by keeping track of the time elapsed from the beginning of the packet. The packet boundary is indicated by a change in the signal power received, and it can be used to find the start of the packet. Assuming we have already woken-up to a pre-specified WiFi signature (as described in the following section), we can then measure the instantaneous signal amplitude by passing the signal through an envelope detector (ED) and monitoring its envelope for a strong rising edge representing the beginning of the packet that will be backscatter modulated.

Before we present how we can perform necessary synchronization, it is first instructive to provide a brief overview of very low-power energy-detecting radio receivers. The simplest and lowest power receiver directly connects an antenna to an ED, whose output is low-pass filtered and then sampled. Fig. 5. shows a simplified block diagram of this approach. This approach's main benefit is that the ED can be passive and thus consumes zero power; this circuit's only power consumption is due to the sampler/comparator, which can be in the low single-digit microwatt regime. Note that this approach energy-detects everything at its input, and thus there is usually a filter at the input to ensure out-of-band interferers do not get demodulated.

The main design parameter in such a receiver is the baseband bandwidth, BW_{BB}, set by the combination of the effective resistance of the ED itself and its load capacitance as a simple first-order *RC* filter. The larger the baseband bandwidth, the more precise an ED will be able to detect a rising packet edge, and therefore the better the synchronization accuracy will be. To first order, this is a one-to-one trade-off, as shown in Fig. 5, assuming that the comparator is sampling at the Nyquist rate (i.e., 2× the baseband bandwidth). To achieve a synchronization accuracy of 150 ns, we would need a baseband bandwidth of at least 6.7 MHz.

However, the complication here is that the larger the baseband bandwidth, the larger the noise bandwidth becomes. As described in [18, 34], with no gain in front of the ED, the receiver's sensitivity is typically dominated by the noise of the ED itself. Interestingly, RF noise is immaterial in such a scenario because a passive ED's noise is so much larger than all downconverted RF noise. As a result, increasing the baseband bandwidth directly increases the noise, which degrades the sensitivity with a $5\log(BW_{BB})$ trade-off (where $5\log()$ instead of $10\log()$ is used to account for the squaring function of the ED). Specifically, the sensitivity of a direct-ED receiver is given by equation 1.

$$P_{sensitivity} = \frac{20}{k_{ED} * A_V^2}\sqrt{BW_{BB} * PSD_0 * SNR_{MIN}} \qquad (1)$$

where k_{ED} is the scaling factor of envelope detector, A_V is the front-end voltage gain (equals to 1 for a direct-ED receiver), PSD_0 is the output-referred baseband noise, and SNR_{MIN} is the required minimum signal-to-noise ratio. The result of this equation is plotted in Fig. 6 for representative values of k_{ED}, PSD_0, and SNR_{MIN} to be 250/V, 300 nV^2/Hz and 6 dB, respectively. Achieving a synchronization accuracy of 150ns requires a baseband bandwidth of 6.7MHz, which, as shown per these numbers, permits a sensitivity of at best −20 dBm. This is obviously unacceptable.

Since we cannot further reduce the noise floor of a passive ED, the only recourse is to either provide more voltage gain before the ED or build an active ED to reduce its noise floor. It turns out that building an active ED with sufficiently low noise is not only not easy but also only helps with a $5\log()$ benefit. On the other hand, providing voltage gain before the ED helps with a $20\log()$ benefit and is thus far more attractive.

In fact, it is possible, even at 2.4 GHz, to provide some amount of voltage gain completely passively through an impedance transformation network. For example, the π network (shown in Fig. 8a later) can take the 50 Ω antenna impedance and transform it to a 300 Ω impedance, theoretically giving $20\log(\sqrt{Z_o/Z_i}) = 8$ dB of "free" voltage gain without consuming power. As shown in Fig. 6, adding 8 dB of voltage gain improves the sensitivity by 8 dB for a net sensitivity of -28 dBm. This is still not good enough. Considering that the detector's impedance is of the order of $100k\Omega$, can we get more voltage gain if we build a better matching network? Unfortunately, the ability to do this is limited by the low-quality factor of components at high frequencies, along with parasitics - where for example, even 0.1 pF of parasitic capacitance presents as 600 Ω of impedance. Likewise, a 0.8 nH high-Q inductor from Coilcraft (0402DC-N80XR, Coilcraft, Illinois, USA) has $Q = 110$, which at 2.4 GHz is a series resistance of 25 mΩ, which limits Z_o to 300 Ω. Thus, it is very difficult to get much more than $\sim$8 dB of passive voltage gain at these frequencies.

As a result, the only remaining way to improve sensitivity is to add active RF gain. This is typically undesired by designers of backscatter tags, as the main purpose of doing backscatter modulation is to avoid having to build active circuits operating at RF since they tend to consume significant power. For example, in this work, we have built a custom RF amplifier into the integrated circuit, which provides a gain of 12dB for a power consumption of 240 μW. With the matching network, the provided gain improves the sensitivity by 20dB, which now meets the desired -40 dBm sensitivity specification (with some margin). However, while still significantly lower than the 10's to 100's of mW a typical WiFi transceiver would consume, the power consumption is still higher than desired.

We can now get to the key insight provided by SyncScatter: the high bandwidth needed for synchronization only needs to occur when we know we are about to backscatter - that is after we have already woken up. As a result, we only need to turn this RF amplifier on for a short amount of time to detect the rising edge of the packet to be backscatter modulated. We can shut-off the RF amplifier before and after this event. By duty-cycling the amplifier in this manner, we can cut down its average power consumption significantly. For example, the sync receiver needs to be turned on only for 50us throughout the 500 μs wakeup + 2000 μs data packet duration. The duty-cycled power, in this case, turns out to be $\frac{50}{2500} \times 240\mu\text{W} = 4.8\mu\text{W}$.

Again, the key insight in SyncScatter is that we can decouple the precise symbol level synchronization from the wake-up functionality, so that we can spend high power momentarily during symbol synchronization to achieve the desired bandwidth and sensitivity, while duty-cycling this down to low average power at times when we are not expecting the packet edge. Unlike other works which combine wake-up and synchronization in a single circuit, this work proposes a hierarchical method - where a low-power wake-up receiver is used to detect a WiFi compliant signature that indicates the next incident packet should be backscatter modulated, which then hierarchically turns on a sync stage receiver to provide the necessary high-bandwidth synchronization accuracy. This hierarchical approach, of course, only helps if we can design a wake-up path with low power and sufficient sensitivity, as will be described next.

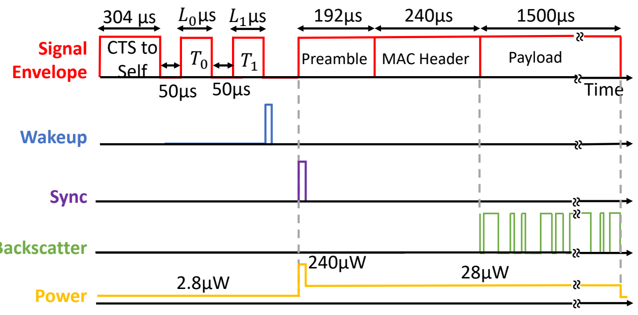


Figure 7: Timing Digram of SyncScatter.

3.2 Wake-up Receiver – First Stage

The wake-up stage's goal is to monitor the RF spectrum for a pre-specified set of packets that indicate the next packet is the one to be backscatter modulated. This should occur with the same sensitivity as the sync stage receiver, but ideally at much lower power since it must be on for potentially long durations of time while waiting to be triggered by a WiFi AP. If the wake-up stage is sufficiently low power, and the sync stage only needs to operate over a small duty-cycle, then the overall hierarchical approach can consume low average power.

The logical question is then: how can the wake-up path consume lower power and yet achieve the same sensitivity as the sync stage receiver? The answer is that the wake-up stage does not require the same amount of bandwidth since it is not being used to perform symbol-level synchronization. Reduced baseband bandwidth enables a reduction in the required amount of pre-ED RF gain to the point where no active RF amplification is needed, all while still meeting the desired -35dBm sensitivity level (with margin).

The wake-up pattern is constructed as follows. A WiFi-compatible identifier is transmitted by a WiFi AP first consisting of a CTS-to-self to temporarily reserve the channel, followed by the transmission of two packets, T0 and T1, with pre-determined lengths corresponding to the IoT tag that is supposed to be woken up. This sequence is illustrated in Fig. 7. Multiple tags can then be uniquely woken up by choosing different T0 and T1 packet lengths. At the tag level, the wake-up stage uses an 8 dB passive voltage gain network that is directly connected to an ED. The ED energy detects the entire packets and samples the energy with a comparator. Since the packet lengths are restricted to a minimum 50 μs, the required ED baseband bandwidth is 20 kHz. As shown previously in Fig. 6, with 8 dB of passive voltage gain, this results in a sensitivity of -40 dBm - which is the desired level (with margin). The comparator's output passes into a counter that counts the number of logic '1's, given by the presence of a packet (vs. a logic '0', which would occur in the inter-packet interval), at a sampling rate of 40 kHz. If the expected number of 1's and 0's occur in the right order, the wake-up stage's output triggers the sync stage receiver. Importantly, this wake-stage is achieved with a purely passive ED that consumes zero power. As a result, the only power here is that of the comparator, correlator, and clock generator (Fig. 4), which consumes only 2.8 μW during active mode.

3.3 Backscatter Communication

Once the tag has woken up and the sync stage identifies the exact packet start instant, the system starts backscattering with zero data. This ensures that the incident WiFi packet's header is backscattered to a different WiFi channel for reception by another WiFi AP without any modification using a Single side-band (SSB) modulation technique similar to [40]. While this is occurring, the tag counts the number of clock cycles until the header is complete, after which it can begin to introduce its data into the backscatter modulator. The backscatter data is XORed with the incident 11b symbol data, also known as code-word translation similar to [40]. The backscatter data is recovered at the receiving AP by XORing the received data again with the original 11b symbol data.

3.4 Putting it all Together

In the subsection, we discuss the end-to-end life-cycle of data packet exchange from an IoT device to the WiFi AP. A WiFi AP with the firmware support to transmit an excitation signal transmits a CTS-to-self packet to reserve a slot of 5 milli-second. Next, the transmitting AP transmits the two packets T0 and T1, whose lengths are a multiple of 25μsec. The tag notices a special pattern of three packets using the wake-up stage receiver by measuring the duration of CTS-to-self, T0, and T1 packets.

Downlink to a Specific Tag: Each IoT device is pre-coded with the lengths for T0 and T1 (akin to a destination address), which is the tag's identity. The finite state machine (Figure 4) at the tag continuously runs the wake-up receiver to look and match the three packet durations consuming 3 μW, with the trigger level of the reference voltage for ED set to -40 dBm. Whenever the three measured lengths match with the precoded sequences, it enables the specific IoT device to receive the downlink data.

A fixed number of bits are allocated for downlink in the finite state machine. The AP transmits the packets with varying lengths to encode the downlink data with 25 μsec granularity. The wake-up receiver at the IoT device uses the packet length to decode the downlink message. Therefore, the downlink data-rate supported is 40 Kbps. We reserve 3 bits for downlink in our implementation, which are used to set the reflection side-band upper or lower.

Uplink from the Tag: Upon completing the downlink, the tag fires up the sync receiver at the IoT device to acquire synchronization to uplink the data. The AP transmits a longer packet which we use to uplink the data. The tag synchronizes to the receiving packet with 150 ns accuracy, assuming incoming power is higher than -40 dBm. The tag starts backscattering at 50 MHz without any data, as soon as it receives a trigger from the sync receiver. Back-scattering with no-data ensures the incoming packet is reflected on channel 11, assuming transmission was on channel 1. The receiving AP on channel 11 starts receiving the packet. It successfully receives the PHY and MAC header of a total of 432 μsec. Upon completion of 432 μsec, the IoT device starts backscattering data, which is compliant to WiFi standards, as discussed in the next section. The receiving AP decodes the packets successfully, with CRC matching ensuring the packet is reported to the higher layers. The receiving AP XORs its data with the transmitted data in the cloud to recover data from the IoT device, thus connecting the IoT device to the AP.

3.5 Working with COTS WiFi

In the previous sub-section, we assumed certain capabilities for COTS WiFi. We will present how our system is compatible with using commercial WiFi transceivers. Specifically, we explain our test setup to receive the backscatter packets and decode them. We also discuss how the physical layer modulation schemes in the 802.11b protocol affect the bit-data processing at the backscatter IC and the receiver end.

Generating the wakeup pattern: The wakeup pattern is an On/Off pattern made up of WiFi compliant packets. It contains two WiFi packets separated by a DCF interframe spacing(DIFS) gap between two successive packets. The WiFi packets are broadcast packets that are of 107us duration, each separated by 50us corresponding to the DIFS gap. We note that typical 802.11b data packet sizes are of the order of few milliseconds, and 500us (CTS-to-self + wakeup pattern) duration do not add significant overhead for the backscatter communication.

Scrambling and Differential encoding: 802.11b WiFi APs randomize the data by scrambling it before transmission. At the receiver end, a de-scrambler is used to de-randomize the data and obtain the original bits. So, the backscatter data bits have to be scrambled before they are transmitted. The backscatter bits are scrambled using a 7^{th} order polynomial implemented as a feedback shift register initialized with a fixed seed [3]. Since the seed is fixed in 11b transmissions, all the tags can be programmed with this seed to facilitate data scrambling. Following the scrambling operation, the bits have to be encoded in a differential manner following the 802.11b PHY differential modulation scheme so that the receiver can decode the backscatter bits correctly.

4 Hardware and IC Design

Each SyncScatter tag is built upon a custom integrated circuit that was fabricated in TSMC's 65nm GP process. Our chip design is inspired by previous works [35, 36], which like other works HitchHike [40] lacks synchronization at the necessary sensitivity levels. Our IC design includes the proposed wake-up, sync receiver, and SSB backscatter modulator. The remainder of this section describes the chip design and operation, along with board-level integration efforts.

IoT Device: Daughterboard: The 65 nm die is directly mounted onto a custom printed circuit board, hereafter referred to as the daughterboard. The daughterboard(Figure 9b) routes all power and digital control traces to headers, which interface with a larger motherboard. Although not strictly necessary, the daughterboard is fabricated to utilize two RF antennas for ease of initial design: one for the wake-up path and one for the backscatter-path. However, we can use a simple switch to include both these paths to interact with a single antenna. The chip is clocked primarily from a 16 MHz crystal oscillator, with the oscillator circuit integrated on the chip, and the crystal soldered onto the daughterboard.

4.1 Wake-up receiver:

The wake up receiver consists of passive voltage gain directly feeding an ED and then to comparator which is sampled at 40kHz sampling rate as shown in Fig. 4.

Passive Voltage gain: At power levels of -40 to -30 dBm, the voltage seen at the antenna port are on the order of 5-20 mV. A passive voltage gain network is included to boost this according to the benefits outlined in the previous section. In this design, an *CLC*-based π matching network is employed, as shown in Fig. 8a, which provides 8 dB of voltage gain. The maximum achievable gain is limited by the quality factor, Q, of the constituent components at 2.4 GHz, along with the input impedance of the ED. In the current implementation, the employed inductor's Q is 110.

Envelope detector and comparator: Since the antenna is single-ended, it's easier to achieve high passive voltage gain with a single-ended matching network. Thus, the ED should also be single-ended. However, in general, it is better to perform baseband signal processing in a differential manner. Considering that the input impedance, output referred noise, and conversion gain are all important parameters to optimize in ED design, a multi-stage fully passive ED design is employed. In particular, a single-ended-to-differential Dickson-based topology is selected, thus acting as a pseudo-balun.

The ED's output bandwidth is controlled in part by the body bias voltage, which controls the ED's effective resistance, along with a variable capacitor. For the stage one design, a bandwidth of 200 kHz is targeted, which is sufficient to enable detection of the presence of T0 and T1 packets and their lengths, though without precise synchronization accuracy.

The pseudo-differential outputs of the stage one ED then feed into a differential comparator based on a Strong-ARM regenerative latch topology. This comparator effectively acts as a 1-bit analog-to-digital converter (ADC), and thus to extract useful timing information, it must be oversampled. As a result, it is clocked at 40 kHz. This clock is derived directly from the on-board crystal, after an on-chip division by a factor of 400. The comparison threshold voltage is tuned by externally controlling the bulk voltages of the input pair of the preamplifier implemented by a *gmC* integrator.

4.2 Sync Receiver

Once the first stage has determined that packets T0 and T1 have been transmitted, we turn on the sync receiver, and its purpose is to look for the rising edge of the subsequently transmitted packet header. It must do so with a bandwidth and sampling frequency greater than 6.67 MHz to meet the 150 *n*sec timing accuracy requirements. Since increasing the envelope detector's bandwidth will, without any other action, decrease sensitivity, this is countered by adding some active gain in front of the envelope detector.

The schematic of the sync receiver is shown in Fig. 4. Here, the stage two design shares the same antenna and passive voltage gain network as the first stage. But, instead of going directly into an envelope detector, it first goes into a low-noise amplifier (LNA). The LNA is designed to support a gain of 11 dB and an active-mode power consumption of 240 μW. If this second stage were on all of the time, this would be a significant power burden to the entire system. This is the beauty of the hierarchical wake-up feature: the LNA only needs to be turned on after the first stage wake-up receiver triggers reception of the appropriate signature. Otherwise, the LNA is nominally in a low-power sleep state. As a result, its average power consumption is extremely low - limited by the frequency of activation by stage one.

A schematic of the LNA is shown in Fig. 8c. A current reuse common source amplifier is implemented to achieve the desired gain with sufficiently low noise. Its output then feeds into a second envelope detector, optimized in a similar manner to wake-up receiver, though in this case for a bandwidth of 32 MHz to ensure a highly accurate rising edge timing. To compensate for the higher bandwidth, an active ED with decreased noise and increased conversion gain is employed in place of the passive ED in the wake-up receiver. A similar comparator as in the wake-up receiver with externally tunable reference voltage is used after the envelope detector, though in this case sampled at 8 MHz, which is sufficient to achieve a 150 ns synchronization time.

4.3 Clock generation:

The 16 MHz crystal oscillator clock is divided on chip into 8 MHz, 2 MHz and 40 kHz to be used by sampling clocks for sync receiver, PLL reference clock and sampling clock for wake up receiver, respectively. To drive the backscatter modulator, an integer-N PLL is adopted with a 2 MHz reference frequency and a dividing ratio of 25. The voltage controlled oscillator is implemented via a current starved ring oscillator with tunable current to ensure stable clock generation against PVT variations and it consumes 1.5 μW power.

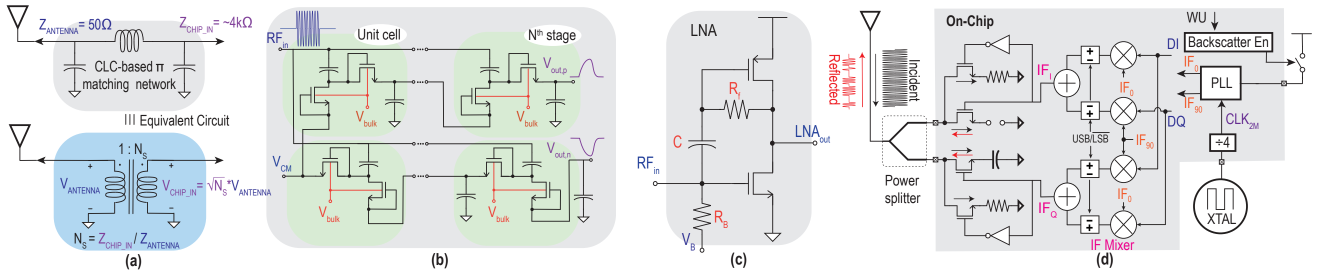


Figure 8: **(a)** CLC-based π matching network, **(b)** Schematic of passive pseudo-balun ED,**(c)** Schematic of LNA, **(d)** Transmission-line-less SSB QPSK Backscatter.

4.4 Backscatter circuit:

Single-side-band QPSK modulation is achieved by the backscatter circuit via the approach shown in Fig. 8d. Here, a power splitter/combiner breaks incident wireless power into two separate paths. The first path meets one of two possible termination conditions: 50 Ω or $Z_{L,0}$, depending on the state of $\mathrm{IF}_{\mathrm{OUT,I}}$. If it meets 50 Ω, then all of the incident RF energy is absorbed by this resistor, and no reflection is generated. However, if it meets $Z_{L,0}$, which is designed to be an open circuit in this implementation, then all of the incident RF power will be reflected back through the power combiner and to the antenna for re-radiation purposes. A similar situation occurs on the other path of the power splitter/combiner, which is terminated by either 50Ω or $Z_{L,90}$, depending on the status of $\mathrm{IF}_{\mathrm{OUT,Q}}$. If $Z_{L,90} = -j \times 50$, then all the signal is reflected, though in this case with a 90° phase shift. If I/Q mixers drive the two paths with 90° separated IF clocks, most of the energy will result in a single sideband, in the same way that a single-side-band mixer operates. Importantly, this approach only requires $Z_{L,90}$ to be a 1.2 pF capacitor - which is very easy to design on chip. This is in contrast to prior work, which required a transmission line to generate the requisite 90° phase shift [40].

4.5 Mother board

The prototype motherboard contains various voltage regulators for the chip and an ultra-low-power microcontroller for generating the data sequence with the correct timing. The voltage regulators generate all the different supply voltages required for the chip. To tune the LNA sensitivity and comparator thresholds, DACs are used to generate the variable voltages, which are controlled by the microcontroller through I2C/SPI buses. The chip's backscatter data input, wake-up signal, and synchronization signals are connected to the microcontroller(MCU) as well. The MCU turns on the sync receiver circuits once the wake-up pattern is detected, and it starts sending the data to the chip after an appropriate delay when the synchronization signal is asserted. The delay ensures the PHY header of the frame is preserved and stays valid. In a practical design, all the regulators and the microcontroller can be integrated onto the same chip, which would greatly reduce size and power consumption.

5 Evaluation

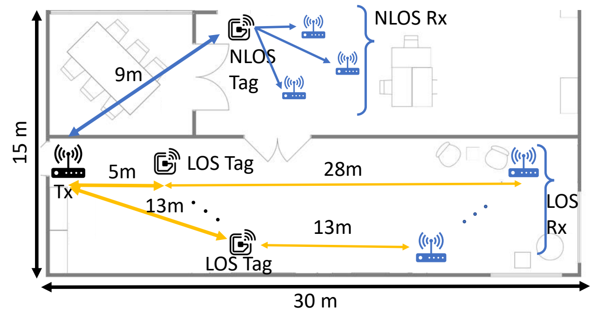
In this section, we first present micro-benchmarks to demonstrate the working of the individual modules in the SyncScatter tag. Then we evaluate the end-to-end performance of the SyncScatter in terms of Bit Error Rate (BER) and goodput by placing the tag in a large indoor environment (30m x 15m) as shown in Fig. 9a. We conduct the experiments by placing the Transmitter AP, Tag, and the receiver AP in both line of sight and non-line of sight conditions to demonstrate that SyncScatter tag is suitable for deployment in an office environment. We also conduct experiments to find SyncScatter's maximum range by co-locating the transmitter and receiver AP. To understand the impact of the co-existence of multiple tags, we perform goodput measurements by keeping two multiple tags in the same environment.

5.1 Micro-benchmarks

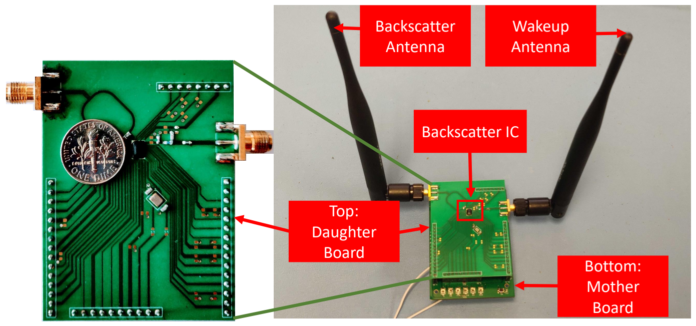
Here we verify the tag's functionality with micro-benchmarks to explain wake-up, synchronization, backscattering modules, and overall system design.

Tag Wake-up Accuracy: To test the robustness of the wake-up receiver, we measure the accuracy of wake-up with varying input power levels. To evaluate this, we conduct a wireless experiment to send a wake-up packet at different power levels. If the tag has woken up to the T0 and T1 packet, then it generates a trigger signal at the end of the wake-up packet. We send 1000 wake-up packets at varying power levels at the tag from -38 dBm to -30 dBm and monitor the number of triggers generated by the tag. The wake-up error rate is calculated as the number of successful triggers divided by the number of wake-up packets sent. Fig. 10a shows the wake-up accuracy for different power levels. SyncScatter's sensitivity is approximately -34 dBm, and for power levels above -34 dBm, the wake-up rate is very close to 1. Hence the tag responds well up to power levels of -34dBm, and SyncScatter will thus be able to wake up at a distance of up to 30 m.

Synchronization Jitter: The synchronization stage is prone to have some timing jitter in detecting the variation of power level. Here we quantify the jitter at different input power levels by doing a wireless experiment similar to the previous setup. To measure the jitter, we send an 802.11b packet from an RF signal generator and monitor the output of the synchronization stage. Then, we measure the time elapsed from

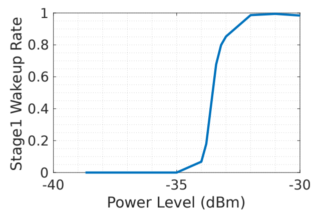


(**a**) Experiment floor plan

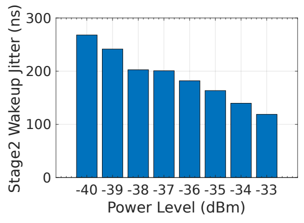


(**b**) SyncScatter tag

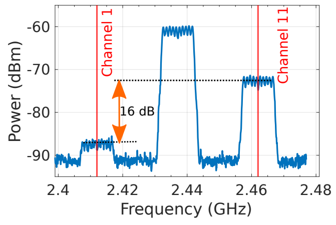
Figure 9: (**a**) The floor plan shows our deployment of the transmitter, the backscatter tag and the receiver AP at eight different locations in an office environment. (**b**) The mother board and the daughter board with the wire-bonded backscatter IC.



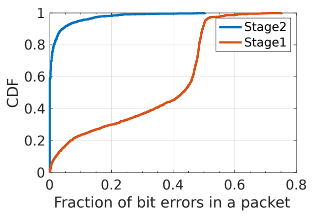
(**a**)



(**b**)



(**c**)



(**d**)

Figure 10: (**a**) Wakeup rate (**b**) Sync Jitter (**c**) SSB Backscattering (**d**) CDF of Bit error rates.

the moment the RF signal is applied and the instant at which the rising edge appears at the synchronization stage's output. We repeat this experiment by sending a thousand packets at different power levels and report the standard deviation of all the measurements at each power level. Figure 10b shows the synchronization jitter is below 150ns at -35 dBm power level and beyond, achieving desired optimal spec.

SSB Backscatter: Here we evaluate the performance of the single sideband backscatter in terms of the image rejection of SyncScatter tag. To measure the image rejection, we configure the tag to shift by 25MHz to the upper sideband and send a Wi-Fi signal on channel 6. Figure 10c shows the backscattered signal on channel 1 and channel 11. We observe that the left sideband signal i.e on channel 1, is ∼16 dB lower than the channel 11 signal.

5.2 Chip power consumption

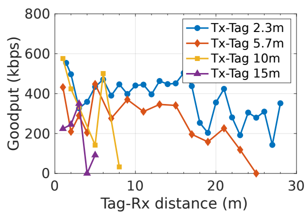
Here we evaluate the power consumption of the wake-up receiver, sync receiver, and backscatter stages of the chip. We report each stage's power consumption as the product of the supply voltage and the current drawn from the supply. The chip is powered from a 0.5v supply, and we use a source measurement unit [2] that can measure the current drawn by the circuit with a precision of 1μA. We observe that the on-board 16MHz oscillator and the wake-up receiver draw a current of 5.6 μA consuming 2.8μW. Since the chip spends most of its time in the wake-up state looking for the right time to backscatter, its average power consumption is dominated by the stage one wake-up receiver. Once the wake-up receiver detects T0 and T1 packets, then sync receiver is turned on, which draws 480 μA current and thus consumes 240μW, but only for an average of 50μs, to account for 40kHz sampling rate error. For a nominal wake-up duration of 500μs and data packet duration of 2ms, the duty-cycled power of the sync receiver is $\frac{50}{2500} \times 240\mu W = 4.8\ \mu W$. Thus, the sync receiver is controlled by a power switch and duty-cycled to conserve power. When the circuit is actually backscattering, the chip draws 56μA current consuming 28μW of power. The power consumption of the backscatter stage is dominated by the PLL to generate the 50MHz clock. Since the backscatter stage is powered on only during the data packet duration, the duty-cycled power turns out to be $\frac{2000}{2500} \times 28\mu W = 22.4\mu W$. So, the total average power from the three stages throughout the communication duration is 2.8 + 4.8 + 22.4 = 30μW, which is in the same range of Hitchhike [40] tag's power consumption.

5.3 End-to-End Results

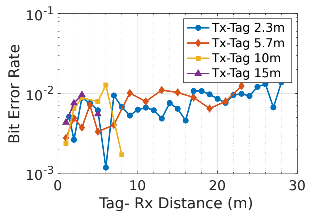
Here we use WiFi radios as both the transmitter and the receiver. The transmitter transmits 802.11b signals at a peak power of +30dBm on WiFi channel 1, and the tag is configured to backscatter to channel 11. The receiver AP is set to monitor channel 11 using Wireshark [4], a packet capture software that allows us to collect the backscatter packets and compute bit error rate (BER) and the goodput.

5.3.1 Impact of synchronization stage on BER

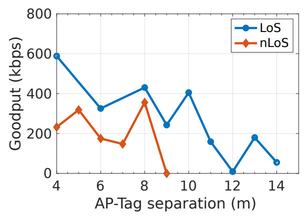
To understand the impact of synchronization on BER, we keep the transmitter, SyncScatter tag, and the receiver AP at 20m away from each other and capture the backscattered packets on channel 11. We perform this experiment in two ways, both with and without using the synchronization stage.



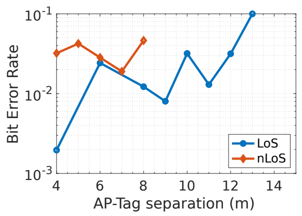
(a) Tx-Rx separated Goodput



(b) Tx-Rx separated BER

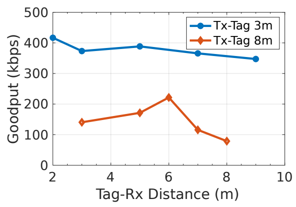


(c) Tx-Rx co-located Goodput

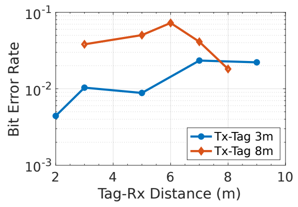


(d) Tx-Rx co-located BER

Figure 11: Goodput and BER performance with various Tx-tag and tag-Rx distances for separated case and co-located case.

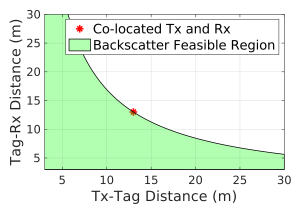


(a) NLOS Goodput

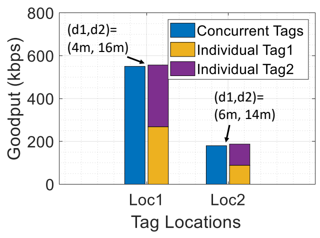


(b) NLOS BER

Figure 12: Goodput and BER performance with various Tag-Rx distances for the NLOS case.



(a) Range Plot



(b) Multi-tag Plot

Figure 13: Backscatter range plots.

We transmit 1000 data packets and plot the CDF of the bit error rates of the captured packets. As shown in Figure 10d, when the synchronization receiver is not used, more than 80 percent of the packets have BER higher than 0.1. But when the synchronization receiver is used to find the packet start, 70 percent of the packets have BER less than 10^{-3}, which is a substantial improvement over the other case.

5.3.2 Goodput/BER for a Single Tag

Here we compute the Goodput and the bit error rates by placing the transmitter, tag, and the AP in different Line of Sight(LoS) as well as non-line of sight(nLoS) configurations. We calculate goodput by taking into consideration the MAC and PHY layer overheads which account for approximately 40 percent of the packet data.

Line of Sight: We perform LoS experiments by placing the Tx, tag, and the Rx in a long corridor. We vary both the Tx to Tag distance and Tag to Rx distance while measuring the performance of SyncScatter tag. Figure 11a shows that when Tx to Tag distance is 2.3m, SyncScatter tag gives an average goodput of 450kbps at up to 30m from the tag. As expected, when we increase the transmitter to tag separation, we observe that the range of backscatter tag decreases due to higher path loss. For example, when tag and transmitter are separated by 10m, SyncScatter tag provides 400kbps goodput only until 9m tag to receiver separation. As shown in Figure 11b, the BER remains to be 10^{-2} over the whole range. We note that although the BER remains low over a wide range of Tx to tag and tag to Rx separations, there is a drop in the goodput values at certain locations. This could be attributed to the multipath nature of wireless channel causing deep fades at certain locations, impacting the number of successful packets decoded which in turn influences the goodput.

Non Line of Sight: In a real-world deployment, true line of sight conditions do not necessarily occur. So, we test our tag setup in a non-line of sight(nLoS) condition where the transmitter and tag are blocked by a wall between them. We vary the transmitter to tag separation and compute the bit error rate and goodput for various tag to receiver separations as plotted in Figure12a and Figure12b. Measurements show that the SyncScatter tag is able to be wake-up successfully even when blocked by a wall and has a backscatter range of 9m. Blockage due to the wall attenuates the signal, resulting in the throughput and BER being worse when compared to the line of sight case. When the Tx to tag separation is 3m, the tag offers an average goodput of 400kbps, and it falls to 100kbps when the Tx to tag separation is increased to 8m.

5.3.3 Co-located Tx and Rx set up

We also evaluate SyncScatter by placing the Transmitter and Receiver at the same location in both LoS and nLoS conditions. In LoS case, the backscatter range is limited to 13m which is smaller than the 30m range observed in the previous experiments where the Tx and Rx are at different separations from the tag. When the Tx and Rx are co-located, the backscatter signal suffers from the path loss twice due to the back and forth travel from the AP to the tag and back to the same location, impacting the maximum range. This argument also supports our observation that the rate of decrease in the goodput with separation is high as shown in Fig.11d and Fig. 11c. In the nLoS scenario, the tag offers an average throughput of 200kbps, with the maximum backscatter range reducing to 8m because of the wall blockage between the Tx and the tag.

5.3.4 Range of the SyncScatter Tag

Now we describe the maximum range until which SyncScatter tag works. In the co-located experimental setup, we observed that the backscatter packet decoding is successful up to 13m AP to tag Line of sight separation. Since the backscatter communication range is determined by the product of

Tx-tag (d_1), Tag-Rx separations(d_2). Any tag location satisfying the relation $d_1 \times d_2 \leq 13 \times 13 = 169m^2$ and $d_1 \leq$ maximum wake-up distance should be within the backscatter range. Fig. 13a shows the feasible backscatter range, which goes up to 30m in either direction.

5.3.5 Co-existence of Multiple Tags

Finally, we find out if multiple SyncScatter tags can exist together, specifically we want to know if the two tags wake-up simultaneously and cause interference to each other. In this experiment, we place the Tx and Receiver separated by 20m. The two tags are placed in line of sight(LoS) with the Tx and Rx and approximately at the same location. Fig. 13b shows the network goodput when both the tags are present and compare it against goodput when only one tag is present. We note that the concurrent goodput is slightly less than the averaged goodput of individual tags.

6 Related Work

SyncScatter is related to prior backscatter based networking with ambient signals [13, 22] that provides low-cost and energy-efficient communication [24, 31]. However, none of the past literature has shown the ability to synchronize at extremely low power with incoming WiFi signals. Furthermore, SyncScatter for the first time analyses the fundamental range limitation for WiFi backscattering, and provides a necessary synchronization specification to maximize the range. Using the analysis and a custom IC hardware design, SyncScatter improves the range, data-rate, and scales to multiple tags over existing literature, even prior IC implementations [35]. SyncScatter is related to the following topics:

Tone based WiFi backscatter communication: SyncScatter synchronizes with ambient commodity WiFi signals, therefore do not require deployment of new tone generators. In contrast, traditional backscatter system like RFID [12, 15, 16, 27, 29, 33, 39, 41] require reader device which act as both tone-generator and receiver. Recent work has shown the ability to backscatter WiFi from an incoming tone signal [19–21, 28, 32], which requires an excitation radio that can generate sine tone. For example, [19] uses a Bluetooth radio to generate a tone, but with the standards-limiting the maximum transmit power, the backscatter range is limited to smaller distances. In contrast, since SyncScatter leverages existing WiFi infrastructure for both excitation and receiving radio, it simplifies the deployment while improving range.

Backscatter communication with Ambient signals: Recently, there have been multiple works on backscattering ambient signals, wherein the excitation radio and receiving radio use existing infrastructure like WiFi [21, 40]. In general, the infrastructure-based backscatter that leverages WiFi [11, 20, 23, 38], LTE [13], Bluetooth [14, 19], ZigBee [22, 43], LoRa [17, 26, 28] or even visible light signal [37] can all benefit from SyncScatter's hierarchical protocol to improve the transmitter-to-tag range and throughput. For example, recent work which backscatters LTE signals [13] tag uses a preamble while backscattering due to the lack of synchronization with incident signals. In such scenarios, the hierarchical wake-up receiver can enable synchronization and improve the tag's performance.

Prior WiFi-based backscatter works focus on the core-principle of codeword translation: changing one OFDM symbol to another valid OFDM symbol [42] and [6] extends backscatter communication to leverage features of the MAC layer. Multi-hop backscatter [45] builds a mesh network of tags, and a few other works [9, 23, 44, 46] propose to leverage spatial multiplexing on the backscatter tag. However, none of the existing works provide extended coverage due to their fundamental inability to synchronize with the ambient signals.

7 Discussion and Future Work

SyncScatter presents a first integrated circuit to achieve synchronized backscatter communication with ambient signals.

802.11b and Beyond: Although 802.11b is an old technology, most modern APs come with dual-band radios supporting both 2.4GHz and 5 GHz radios. The recent 802.11ax standard is designed to be backward compatible with 11b/g devices and hence backscattering on 802.11b signals is still very applicable. Moreover, all the symbol-based backscatter systems such as FreeRider [42] which use 802.11g signals, requires synchronization making our design suitable for modern WiFi standards as well.

Adapting WakeUp Receivers to different protocols: Our hierarchical wake-up receiver design can be extended to backscatter systems based on BLE, OFDM, and LoRA protocols with some minor modifications. We observe that BLE and LoRA signals have a constant signal envelope similar to 11b signals, and they would work with our design. However, since LoRA networks demand long-range, the wake-up receiver has to be designed with more sensitivity at the expense of more power consumption on the tag. In the case of OFDM signals, the wake-up packets have to be engineered to have a small peak to average power ratio (PAPR) so that they appear like a constant envelope signal to the envelope detectors.

Integrated Power Management and Energy Harvesting: In the current implementation, we design the integrated circuit without integrated power management solutions, leading to higher power consumption. In future work, we would integrate the power management circuits such as voltage regulators, power switches within the integrated circuit itself. We note that our chip can be powered using a rechargeable coin cell such as CR2032 [1] and can have a continuous operation for more than 3 years. We can further enhance its life by exploring RF energy harvesting [7, 30] techniques to replenish the battery and integrate them onto the fabricated chip.

8 Acknowledgements

We thank anonymous reviewers and our shepherd, Prof. Haitham Hassanieh, and members of WCSNG, UCSD for their insightful feedback. This work was supported in part by the National Science Foundation under Grant No. 1923902.

References

[1] Cr2032 coin cell battery. https://data.energizer.com/pdfs/cr2032.pdf.

[2] Keithley source measurement unit. http://www.farnell.com/datasheets/2238744.pdf?_ga=2.15412083.450713776.1499842582-1530823304.1499842582.

[3] Vocal technologies: 802.11b white paper. https://www.vocal.com/wp-content/uploads/2012/05/802.11b_wp1pdf.pdf.

[4] Wireshark - packet captture and analysis tool. https://www.wireshark.org/.

[5] Ieee standard for information technology - telecommunications and information exchange between systems - local and metropolitan networks - specific requirements - part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications: Higher speed physical layer (phy) extension in the 2.4 ghz band. *IEEE Std 802.11b-1999*, pages 1–96, 2000.

[6] A. Abedi, M. H. Mazaheri, O. Abari, and T. Brecht. Witag: Rethinking backscatter communication for wifi networks. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, pages 148–154, 2018.

[7] F. Alneyadi, M. Alkaabi, S. Alketbi, S. Hajraf, and R. Ramzan. 2.4 ghz wlan rf energy harvester for passive indoor sensor nodes. In *2014 IEEE International Conference on Semiconductor Electronics (ICSE2014)*, pages 471–474. IEEE, 2014.

[8] R. Barnett, G. Balachandran, S. Lazar, B. Kramer, G. Konnail, S. Rajasekhar, and V. Drobny. A passive uhf rfid transponder for epc gen 2 with-14dbm sensitivity in 0.13 μm cmos. In *2007 IEEE international solid-state circuits conference. digest of technical papers*, pages 582–623. IEEE, 2007.

[9] D. Bharadia, K. R. Joshi, and S. Katti. Full duplex backscatter. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, pages 1–7, 2013.

[10] D. Bharadia, K. R. Joshi, M. Kotaru, and S. Katti. Backfi: High throughput wifi backscatter. *ACM SIGCOMM Computer Communication Review*, 45(4):283–296, 2015.

[11] D. Bharadia, K. R. Joshi, M. Kotaru, and S. Katti. Backfi: High throughput wifi backscatter. *ACM SIGCOMM Computer Communication Review*, 45(4):283–296, 2015.

[12] M. Buettner, B. Greenstein, and D. Wetherall. Dewdrop: an energy-aware runtime for computational rfid. In *Proc. USENIX NSDI*, pages 197–210, 2011.

[13] Z. Chi, X. Liu, W. Wang, Y. Yao, and T. Zhu. Leveraging ambient lte traffic for ubiquitous passive communication. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 172–185, 2020.

[14] J. F. Ensworth and M. S. Reynolds. Every smart phone is a backscatter reader: Modulated backscatter compatibility with bluetooth 4.0 low energy (ble) devices. In *2015 IEEE international conference on RFID (RFID)*, pages 78–85. IEEE, 2015.

[15] J. Gummeson, S. S. Clark, K. Fu, and D. Ganesan. On the limits of effective hybrid micro-energy harvesting on mobile crfid sensors. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 195–208, 2010.

[16] J. Gummeson, P. Zhang, and D. Ganesan. Flit: A bulk transmission protocol for rfid-scale sensors. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 71–84, 2012.

[17] M. Hessar, A. Najafi, and S. Gollakota. Netscatter: Enabling large-scale backscatter networks. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 271–284, 2019.

[18] X. Huang, G. Dolmans, H. d. Groot, and J. R. Long. Noise and sensitivity in rf envelope detection receivers. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 60(10):637–641, 2013.

[19] V. Iyer, V. Talla, B. Kellogg, S. Gollakota, and J. Smith. Inter-technology backscatter: Towards internet connectivity for implanted devices. In *SIGCOMM*, 2016.

[20] B. Kellogg, A. Parks, S. Gollakota, J. R. Smith, and D. Wetherall. Wi-fi backscatter: Internet connectivity for rf-powered devices. In *ACM SIGCOMM Computer Communication Review*, 2014.

[21] B. Kellogg, V. Talla, S. Gollakota, and J. R. Smith. Passive wi-fi: Bringing low power to wi-fi transmissions. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, page 151–164, USA, 2016. USENIX Association.

[22] Y. Li, Z. Chi, X. Liu, and T. Zhu. Passive-zigbee: enabling zigbee communication in iot networks with 1000x+ less power consumption. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, pages 159–171, 2018.

[23] X. Liu, Z. Chi, W. Wang, Y. Yao, and T. Zhu. Vmscatter: A versatile {MIMO} backscatter. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 895–909, 2020.

[24] S. Naderiparizi, M. Hessar, V. Talla, S. Gollakota, and J. R. Smith. Towards battery-free {HD} video streaming. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 233–247, 2018.

[25] Openwrt. Openwrt project. https://openwrt.org/.

[26] Y. Peng, L. Shangguan, Y. Hu, Y. Qian, X. Lin, X. Chen, D. Fang, and K. Jamieson. Plora: A passive long-range data network from ambient lora transmissions. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 147–160, 2018.

[27] A. P. Sample, D. J. Yeager, P. S. Powledge, and J. R. Smith. Design of a passively-powered, programmable sensing platform for uhf rfid systems. In *2007 IEEE international Conference on RFID*, pages 149–156. IEEE, 2007.

[28] V. Talla, M. Hessar, B. Kellogg, A. Najafi, J. R. Smith, and S. Gollakota. Lora backscatter: Enabling the vision of ubiquitous connectivity. *IMWUT*, 2017.

[29] S. J. Thomas and M. S. Reynolds. A 96 mbit/sec, 15.5 pj/bit 16-qam modulator for uhf backscatter communication. In *2012 IEEE International Conference on RFID (RFID)*, pages 185–190. IEEE, 2012.

[30] V. H. Tran, A. Misra, J. Xiong, and N. Hirunima. Can wifi beamforming support an energy-harvesting wearable? In *Proceedings of the Fifth ACM International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems*, ENSsys'17, page 14–20, New York, NY, USA, 2017. Association for Computing Machinery.

[31] D. Vasisht, G. Zhang, O. Abari, H.-M. Lu, J. Flanz, and D. Katabi. In-body backscatter communication and localization. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 132–146, 2018.

[32] A. Wang, V. Iyer, V. Talla, J. R. Smith, and S. Gollakota. Fm backscatter: Enabling connected cities and smart fabrics. In *NSDI*, pages 243–258, 2017.

[33] J. Wang, H. Hassanieh, D. Katabi, and P. Indyk. Efficient and reliable low-power backscatter networks. *ACM SIGCOMM Computer Communication Review*, 42(4):61–72, 2012.

[34] P.-H. P. Wang, H. Jiang, L. Gao, P. Sen, Y.-H. Kim, G. M. Rebeiz, P. P. Mercier, and D. A. Hall. A 6.1-nw wake-up receiver achieving -80.5 dbm sensitivity via a passive pseudo-balun envelope detector. *IEEE Solid-State Circuits Letters*, 1(5):134–137, 2018.

[35] P.-H. P. Wang, C. Zhang, H. Yang, D. Bharadia, and P. P. Mercier. 20.1 a 28μw iot tag that can communicate with commodity wifi transceivers via a single-side-band qpsk backscatter communication technique. In *2020 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 312–314. IEEE, 2020.

[36] P.-H. P. Wang, C. Zhang, H. Yang, M. Dunna, D. Bharadia, and P. P. Mercier. A low-power backscatter modulation system communicating across tens of meters with standards-compliant wi-fi transceivers. *IEEE Journal of Solid-State Circuits*, 55(11):2959–2969, 2020.

[37] X. Xu, Y. Shen, J. Yang, C. Xu, G. Shen, G. Chen, and Y. Ni. Passivevlc: Enabling practical visible light backscatter communication for battery-free iot applications. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, pages 180–192, 2017.

[38] Z. Yang, Q. Huang, and Q. Zhang. Nicscatter: Backscatter as a covert channel in mobile devices. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, pages 356–367, 2017.

[39] H. Zhang, J. Gummeson, B. Ransford, and K. Fu. Moo: A batteryless computational rfid and sensing platform. *University of Massachusetts Computer Science Technical Report UM-CS-2011-020*, 2011.

[40] P. Zhang, D. Bharadia, K. Joshi, and S. Katti. Hitchhike: Practical backscatter using commodity wifi. In *SenSys*, 2016.

[41] P. Zhang, P. Hu, V. Pasikanti, and D. Ganesan. Ekhonet: High speed ultra low-power backscatter for next generation sensors. In *Proceedings of the 20th annual international conference on Mobile computing and networking*, pages 557–568, 2014.

[42] P. Zhang, C. Josephson, D. Bharadia, and S. Katti. Freerider: Backscatter communication using commodity radios. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, pages 389–401, 2017.

[43] P. Zhang, M. Rostami, P. Hu, and D. Ganesan. Enabling practical backscatter communication for on-body sensors. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 370–383, 2016.

[44] J. Zhao, W. Gong, and J. Liu. Spatial stream backscatter using commodity wifi. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, pages 191–203, 2018.

[45] J. Zhao, W. Gong, and J. Liu. X-tandem: Towards multi-hop backscatter communication with commodity wifi. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, pages 497–511, 2018.

[46] R. Zhao, F. Zhu, Y. Feng, S. Peng, X. Tian, H. Yu, and X. Wang. Ofdma-enabled wi-fi backscatter. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–15, 2019.

9 Appendix

9.1 BLE synchronization requirement

To find the synchronization requirement for BLE(Bluetooth low energy) backscatter communication, we perform Matlab simulations for different synchronization delays. BLE signals encode the data bits using two different sine tones located 250kHz on either side of the center frequency, where each symbol in the BLE packet occupies a duration of $1\mu s$. To encode tag data on a BLE packet, the frequency of the sine tone present in a BLE symbol is modified by the backscatter tag. In our simulation, we consider incident data packets that are modulated using FSK(frequency shift keying) technique. The tag's modulation signal is also generated from the tag bits using the FSK technique. To incorporate the synchronization delay while backscattering, we delay the tag modulation by a fixed duration (given by the synchronization delay) beginning from the packet start instant. The resulting backscattered signal is also an FSK signal and is decoded by an FSK receiver. Fig 14 plots the BER vs SNR curve for different values of synchronization delay. As can be seen, the BLE backscatter loses 4 dB of SNR at 10^{-3} BER due to a synchronization delay of 100 ns. For 150ns synchronization delay, it loses more than 8 dB SNR. To not degrade the SNR, BLE backscatter requires a synchronization accuracy better than 100 ns which is equal to $\frac{1}{10}$ th of the symbol duration.

9.2 OFDM Synchronization requirement

To simulate OFDM backscatter synchronization requirements, here we consider the standard 20MHz OFDM signals. A WiFi packet contains many OFDM symbols, with each symbol containing 64 sub-carriers. Each OFDM symbol occupies 3.2 us duration and is preceded by a cyclic prefix of 0.8us duration. To encode data onto a WiFi packet, the backscatter tag changes the phase of an OFDM symbol. For instance, to convey tag data 1, the tag induces an additional phase of π radians on all the sub-carriers. Similarly, to encode bit 0, the phase of the sub-carriers is left unchanged. In this way, a sequence of tag bits is encoded on a WiFi packet by modifying the phase of each OFDM symbol in the packet. At the receiver, these phases are extracted to decode the tag data. If the backscatter tag is not aware of the OFDM symbol boundary, the tag bits will not be appropriately encoded on the WiFi packet, and the WiFi packet suffers from loss of SNR. The BER vs SNR curves for different synchronization delays are plotted in Fig 15. We note that the OFDM backscatter is able to tolerate a large synchronization delay of up to 1000ns. The primary reason the tolerance is very large is that every OFDM symbol has a cyclic prefix of length 800ns discarded while decoding the packet at the receiver. So, if the signal that is part of a cyclic prefix is corrupted, it will not impact the bit error rate. Another reason is that the same tag bit is encoded on every sub-carrier of the OFDM symbol, implying if the majority of the sub-carriers are decoded correctly, the BER would not be impacted much. From Fig 15, we notice that the BER floors, although the SNR increases and it loses more than 10 dB SNR to achieve 10^{-3} BER. This suggests synchronization delay plays a major role in reducing the bit error rate.

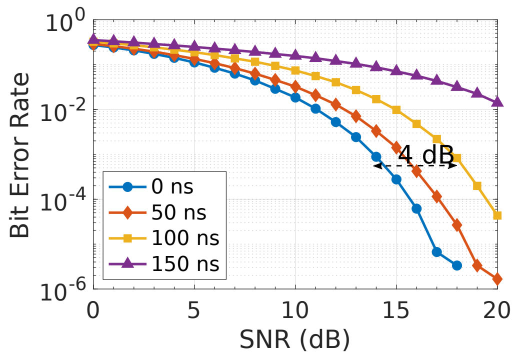


Figure 14: BER vs SNR curve for different synchronization delays in BLE backscatter

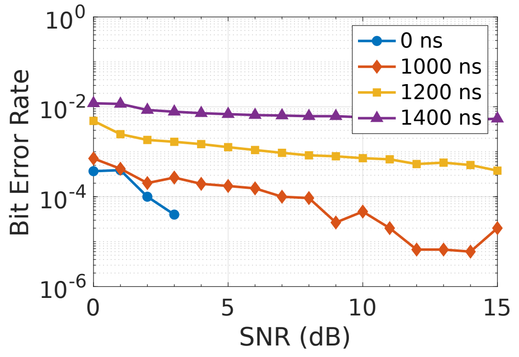


Figure 15: BER vs SNR curve for different synchronization delays in OFDM backscatter

Verification and Redesign of OFDM Backscatter

Xin Liu[1]*, Zicheng Chi[2]*, Wei Wang[1], Yao Yao[1], Pei Hao[1], and Ting Zhu[1]†
[1] *University of Maryland, Baltimore County*
[2] *Cleveland State University*

Abstract

Orthogonal frequency-division multiplexing (OFDM) has been widely used in WiFi, LTE, and adopted in 5G. Recently, researchers have proposed multiple OFDM-based WiFi backscatter systems [33, 35, 36] that use the same underlying design principle (i.e., codeword translation) at the OFDM symbol-level to transmit the tag data. However, since the phase error correction in WiFi receivers can eliminate the phase offset created by a tag, the codeword translation requires specific WiFi receivers that can disable the phase error correction. As a result, phase error is introduced into the decoding procedure of the codeword translation, which significantly increases the tag data decoding error. To address this issue, we designed a novel OFDM backscatter called TScatter, which uses high-granularity sample-level modulation to avoid the phase offset created by a tag being eliminated by phase error correction. Moreover, by taking advantage of the phase error correction, our system is able to work in more dynamic environments. Our design also has two advantages: much lower BER and higher throughput. We conducted extensive evaluations under different scenarios. The experimental results show that TScatter has i) three to four orders of magnitude lower BER when its throughput is similar to the latest OFDM backscatter system MOXcatter [36]; or ii) more than 212 times higher throughput when its BER is similar to MOXcatter. Our design is generic and has the potential to be applied to backscatter other OFDM signals (e.g., LTE and 5G).

1 Introduction

By reflecting ambient signals, backscatter systems conduct passive communication which can provide low power consumption, low cost, and ubiquitous connectivity for Internet of Things (IoT) devices to support various applications (e.g., smart buildings and smart health) [4]. To achieve ubiquitous connectivity and leverage the advantages of the ambient signals, researchers have developed various backscatter systems that reflect ambient signals from TV or frequency modulation

*Both authors contributed equally to the paper.
†Ting Zhu is the corresponding author.

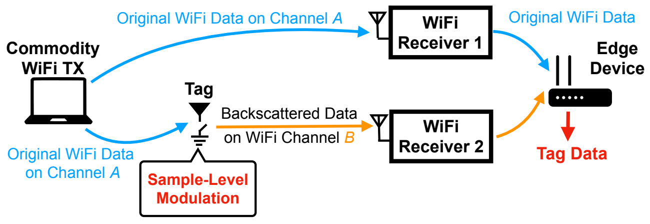


Figure 1: A general architecture of OFDM-based WiFi backscatter systems [33, 35, 36]. Different from existing systems that modulate the tag data at the symbol-level, our backscatter system uses the sample-level modulation (highlighted in a red color).

(FM) radio towers, LoRa, or WiFi [7, 12, 16, 27]. Therefore, these backscatter systems are complimentary to each other and have their own unique advantages in terms of energy efficiency, throughput, deployment cost, etc.

In this paper, we mainly focus on the design of the WiFi backscatter due to i) pervasively available WiFi signals inside buildings; ii) numerous WiFi devices; and iii) abundant applications supported by WiFi. All of these can significantly increase the adoption of our developed techniques. The pioneer work on WiFi backscatter [11] achieved up to 1 kbps throughput and 2.1 meters range. Follow up works improved the performance of WiFi backscatter by using customized full-duplex WiFi access points [6] or standard 802.11 b/g/n WiFi devices [33, 35, 36]. Since the majority of the WiFi signals in buildings are using an advanced modulation scheme – orthogonal frequency-division multiplexing (OFDM), researchers recently proposed a general architecture of OFDM-based WiFi backscatter systems (shown in Fig. 1) to leverage productive WiFi data communication from the surrounding WiFi devices. This architecture has demonstrated to be effective in supporting the smart offices application, in which WiFi receivers can be connected by Ethernet backhaul to decode the tag data [33, 35, 36].

However, these systems [33, 35, 36] require specific WiFi receivers that can disable the phase error correction. This is because the phase error correction can eliminate the phase offset created by the tag, and cause incorrect tag data decoding.

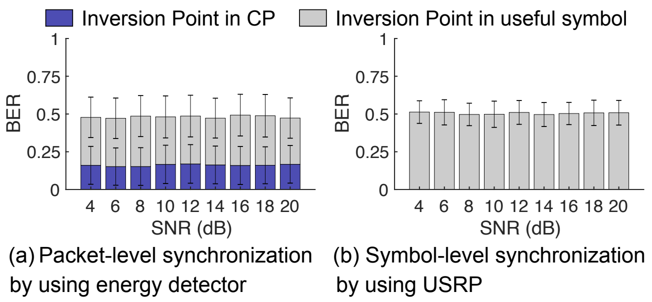


Figure 2: High BER identified in experiments with a total number of 128,000 data points transmitted

On the other hand, the phase error correction is very important for the WiFi demodulation because phase errors in the WiFi systems may be dynamically changing due to the changes of environment (e.g., temperature of the oscillators and object movement). Therefore, WiFi protocols always arrange a certain number of pilot subcarriers in each OFDM symbol to track and correct phase errors. Without the phase error correction, the WiFi demodulation error will increase. Similarly, disabling the phase error correction will also introduce the phase error into the demodulation procedure of the codeword translation, which will increase the tag decoding error.

To demonstrate the limitation of these OFDM-based WiFi backscatter systems, we rebuilt these systems using USRPs that ran the standard 802.11g stack which contains the phase error correction. We identified the high bit error rate (BER) which happens even when the signal to noise ratio (SNR) is high (shown in Fig. 2). By using an energy detector (which is a low-power component to measure the signal strength in the air) as introduced in previous systems, the tag is synchronized with the WiFi sender at the packet-level. From the result (Fig. 2(a)), we can observe that the total BER is around 50% even with a high SNR. Since a tag embeds the tag data by inverting the phase of the incoming signal, the inversion point can settle anywhere in a symbol with packet-level synchronization. After conducting a thorough analysis, we found that one portion of the inversion points (in blue) settle in cyclic prefixes (used to prevent inter symbol interference), which causes decoding failures because the cyclic prefix is removed at the receiver. The other portion of the inversion points (in gray) settle in useful symbols (used to carry real WiFi data), however, this portion will also cause the bit error. We also conducted symbol-level synchronization and found that the blue part can be eliminated with a precise synchronization (results shown in Fig. 2(b)). However, the BER is still very high across different SNRs.

To address this issue, we conducted comprehensive studies and designed a new OFDM backscatter system called TScatter to achieve higher reliability (lower BER) and higher throughput. We propose a sample-level modulation scheme, in which the phase offsets on pilot subcarriers are very different from that on the data subcarriers. Thus, the phase offsets on the data subcarriers cannot be eliminated by the phase error correction. Therefore, we can extract the tag data while the phase error correction is present. Moreover, since the phase error correction is present, our system is able to work in more dynamic environments and achieve much lower BER.

Our sample-level backscatter system has a lot of benefits. On one hand, the system provides high reliability (low BER) allowing tags to work in various scenarios, such as non-line-of-sight or underground. On the other hand, our TScatter can be configured in high throughput mode with Mbps level throughput to support IoT edge computing applications such as ubiquitous surveillance in smart buildings. Further more, high throughput provides high energy efficiency (bit/joule). Given the same amount of harvested energy, our backscatter system can transmit more data than existing ambient WiFi backscatter systems. Moreover, higher energy efficiency also provides another benefit – given the same amount of data to be transmitted, our backscatter system needs much less energy. Therefore, our backscatter system can be deployed in places that are further away from the wireless energy transferring sources than the latest WiFi backscatter systems [33, 35, 36].

The main contributions of this paper are as follows:

• We verified existing OFDM-based WiFi backscatter systems and redesigned the system with a novel sample-level modulation technique, which can retrieve tag data from the process of phase error correction in modern OFDM-based wireless communication system. Potentially, this technique can be applied to LTE and 5G which also use OFDM modulation schemes.

• To demodulate the high granularity modulated data, we built demodulation models that capture the WiFi demodulation procedure and derive a minimization function to estimate the tag data. We further enhanced our modulation and demodulation design to support different WiFi modulation schemes (e.g., BPSK, QPSK, 16QAM, and 64QAM). Our evaluation results demonstrate the effectiveness of our design.

• We built a hardware prototype of our proposed backscatter system that contains a low power FPGA and a simple RF switch. We also designed a tag IC for estimating the power consumption. Our empirical results show that TScatter has i) three to four orders of magnitude lower BER when its throughput is similar to the latest OFDM backscatter system MOXcatter [36]; or ii) more than 212 times higher throughput when its BER is similar to MOXcatter.

2 Background of Existing OFDM Backscatter

To fully reveal the impact of phase error correction on the codeword translation technique, it is necessary to first understand how the technique works. Generally, current OFDM backscatter techniques mainly consist of three parts: (i) the excitation OFDM signal generation on the sender side; (ii) the phase modulation on the tag side; and (iii) the codeword decoding on the receiver side.

The Sender: To produce the OFDM symbol, the sender conducts an Inverse Discrete Fourier Transform (IDFT) on its subcarriers in the frequency domain. The output samples of

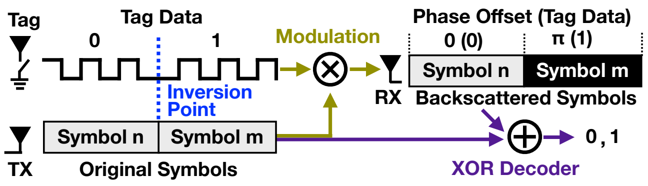


Figure 3: Overview of the existing OFDM backscatter technique

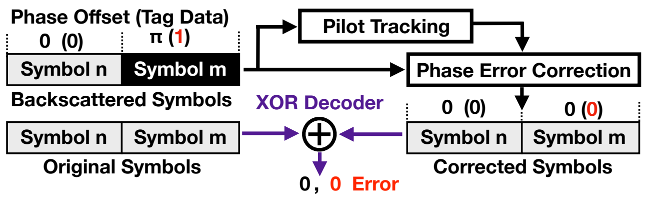


Figure 4: The phase offset π is eliminated by phase error correction, which causes tag data 1 to be mistakenly decoded as tag data 0.

the corresponding IDFT will form a single OFDM symbol $\mathcal{S}(t)$ in the time domain, which can be represented as:

$$\mathcal{S}(t) \; = \; \text{IDFT} \; [\; \mathcal{X}_k \;] \tag{1}$$

where $\{\mathcal{X}_k\}$ are the subcarriers. Multiple symbols are concatenated to create the final excitation OFDM signal.

The Tag: Existing OFDM-based WiFi backscatter systems allow a tag to convey information by inverting the phase of OFDM symbols in the time domain [33, 35, 36]. An example is shown in Fig. 3, the backscatter tag uses zero phase offset to transmit data zero and a phase inversion to transmit data one. Then, the backscattered symbol $\mathcal{B}(t)$ is given by:

$$\mathcal{B}(t) \; = \; \begin{cases} \mathcal{S}(t)e^{j0} & \text{Tag data 0} \\ \mathcal{S}(t)e^{j\pi} & \text{Tag data 1} \end{cases} \tag{2}$$

To improve the data rate, the tag can create additional phase offsets to covey more information, which is shown below:

$$\mathcal{B}(t) \; = \; \begin{cases} \mathcal{S}(t)e^{j0} & \text{Tag data 00} \\ \mathcal{S}(t)e^{j\frac{\pi}{2}} & \text{Tag data 01} \\ \mathcal{S}(t)e^{j\pi} & \text{Tag data 10} \\ \mathcal{S}(t)e^{j\frac{3\pi}{2}} & \text{Tag data 11} \end{cases} \tag{3}$$

We note that to achieve above codeword translations (Eqn. 2 and 3), the tag needs to synchronize the tag data with the symbol. Formally, we define the point that concatenates two different tag data as the tag data **inversion point**. As shown in Fig. 3, the tag data inversion point (from 0 to 1) is aligned with the beginning of the OFDM symbol m.

The Receiver: A Discrete Fourier Transform (DFT) is performed to convert the backscattered symbols to the frequency-domain backscattered subcarriers. Because of the linearity of the DFT, the operation of the phase change in the time domain corresponds to the frequency multiplication. Therefore, the backscattered subcarriers can be represented as:

$$\mathcal{X}_k e^{j\delta} \; = \; \text{DFT} \; [\; \mathcal{B}(t) \;] \tag{4}$$

where δ is the phase offset. Then, we can decode the tag data by computing the XOR operation of backscattered subcarriers and original subcarriers:

$$\mathcal{X}_k e^{j\delta} \oplus \mathcal{X}_k \; = \; \begin{cases} \text{Tag data 0} & \delta = 0 \\ \text{Tag data 1} & \delta = \pi \end{cases} \tag{5}$$

3 Why Existing OFDM Backscatter Systems Disable Phase Error Correction?

The underlying assumption in the codeword translation is that backscatter systems can be free of the effect of the phase error. However, we point out that failure to consider the effect of phase error will significantly increase the BER of OFDM backscatter systems. Therefore, in this section, we first investigate influences of the phase error correction on codeword translation. Then, we extensively analyze all the scenarios that will affect the BER of the codeword translation. At last, we outline the desired properties to reduce the BER of OFDM backscatter systems.

3.1 Tag Data is Eliminated

In real-world scenarios, due to the changes of environment (e.g., temperature of the oscillators and object movement), the OFDM receiver is required to perform the phase error correction to eliminate the phase error on the signal. This process also eliminates the phase offset created by the tag.

Specifically, the phase error correction is to use available pilot subcarriers $\{\mathcal{X}_p\}$ to track the phase error. Since the phase error can be simply modeled as a constant multiplicative component across a symbol (e.g., $e^{j\phi}$ [25]), the backscattered subcarrier in Eqn. 4 should be $\mathcal{X}_k e^{j(\delta+\phi)}$ while the backscattered pilot subcarrier should be $\mathcal{X}_p e^{j(\delta+\phi)}$. Then, the phase error $\mathcal{H}$ can be computed by comparing backscattered pilot subcarriers and original pilot subcarriers:

$$\mathcal{H} \; = \; \frac{\sum \mathcal{X}_p e^{j(\delta+\phi)}}{\sum \mathcal{X}_p} \; = \; e^{j(\delta+\phi)} \tag{6}$$

Finally, the backscattered subcarriers can be corrected as:

$$\mathcal{X}_k e^{j(\delta+\phi)} \cdot \mathcal{H}^{-1} \; = \; \mathcal{X}_k \tag{7}$$

The key observation from the above expression is that the phase error $e^{j\phi}$, as well as the phase offsets $e^{j\delta}$ from the codeword translation, are eliminated from subcarriers. In other words, as shown in Fig. 4, if the tag transmits arbitrary data, the XOR decoder may never output data 1. This is because the codeword translation makes the pilot subcarriers have the same phase offset as other subcarriers, which makes it feasible to eliminate phase offsets by the phase error correction. As a result, since phase offsets are eliminated, tag data 1 will be mistakenly decoded as tag data 0 (i.e., zero phase offset in Eqn. 2). The effect of phase error correction will become more severe when the tag increases its data rate. As shown in Eqn. 3, the tag uses four phase offsets to double the data rate. However, due to the phase error correction, the receiver may not extract these phases offsets correctly. In this case, the tag data 01, 10, 11 will be mistakenly decoded as tag data 00.

The above analysis mainly focuses on the scenario that the inversion point is perfectly aligned with the beginning of the

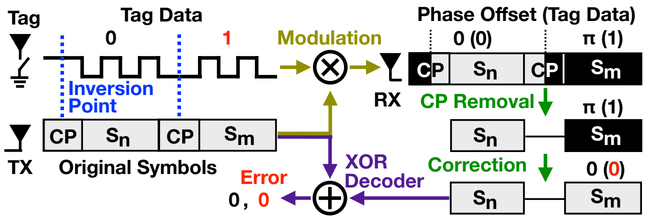


Figure 5: When the inversion point is located in the CP, the phase offset π representing tag data 1 can still be eliminated.

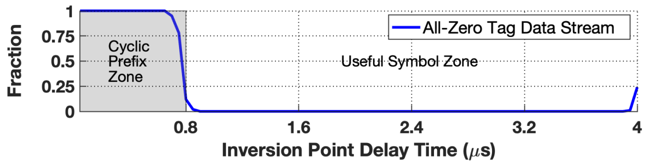


Figure 6: Experiments demonstrate that if the inversion point is within the cyclic prefix zone, the decoded tag data are all zeros regardless whatever data is sent by the backscatter tag.

symbol. In following sections, we will show that even when the inversion point is not perfectly aligned with the beginning of the symbol (i.e., the inversion point is in the cyclic prefix or the inversion point is in the useful symbol), the receiver may still face high bit error rate.

3.2 Inversion Point in Cyclic Prefix

The XOR decoder cannot extract the tag data correctly when the tag data inversion point is located in the cyclic prefix (CP). Because the OFDM receiver will first remove CP to prevent intersymbol interference introduced by the multipath effect, the inversion point in the CP is removed as well. As shown in Fig. 5, after the CP removal, although the useful symbol S_m still has the phase offset π, the phase error correction will eliminate the phase offset, which causes the decoding error.

We conduct an experiment to show the effect of the inversion point in the cyclic prefix. In this experiment, an 802.11g OFDM-based WiFi physical layer is implemented on the USRP B210 [1]. In order to precisely control the synchronization between the OFDM symbol and the tag data, the tag is connected to the USRP using two wires: one for common ground and another for signal. When the USRP transmits the WiFi signal to the air, it will also transmit the starting signal through the wire to tell the tag to embed tag data in the WiFi signal. By modifying the delay after the starting signal, we can synchronize the tag data inversion point with the different timings in the OFDM symbol duration ($4\mu s$). The original tag data is an arbitrary data stream. However, as shown in Fig. 6, when the inversion points are within the cyclic prefix zone (i.e., the inversion point delay is less than $0.7\ \mu s$), the decoded tag data are all zeros, because the phase offsets of the tag data 1 are eliminated. When the inversion points are within the useful symbol zone, the decoded tag data is not all zeros. However, as we will discuss in Section 3.3, it may still have a high BER. When the inversion point delay is between $0.7\ \mu s$

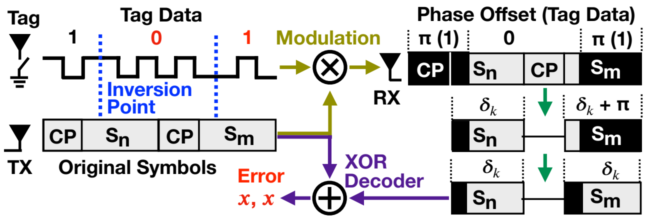


Figure 7: When the inversion point is located in the useful symbol, the decoding result is the same and thus causes the decoding error.

and $0.8\ \mu s$, although it is theoretically within the cyclic prefix zone, the decoded tag data is not all zeros. This is because of the multipath effect in the real-world, so the inversion point may be moved to the useful symbol zone.

3.3 Inversion Point in Useful Symbol

In this section, we analyze the scenario that the inversion point is located in the useful symbol. As shown in Fig. 7, since the phase offsets introduced in the symbols of S_n and S_m are opposite, the subcarriers of S_n and S_m have the opposite phase offsets δ_k and $\delta_k + \pi$ after the CP removal, where k is the subcarrier index. Obviously, the subcarriers of S_m have a common phase offset π, which can be eliminated by the phase error correction. Therefore, the final phase offsets on S_n and S_m are almost the same, which causes the same decoding result and corresponding errors. We note that δ_k on each subcarrier is different and determined by the inversion point position. Hence, the decoding result x might be 1 or 0.

3.4 Desired Properties and Our Solution

From the analysis in Sec. 3.2 and 3.3, the OFDM backscatter should satisfy the following two properties to avoid the decoding error caused by the phase error correction:

1. *The tag data inversion point should fall into the useful symbol.*
2. *The tag data should be represented by phase offsets that cannot be eliminated.*

Since the backscatter tag is a low power device, we cannot use a high power circuit to conduct precise synchronization between the inversion point and the useful symbol. To overcome this challenge, we explore a high granularity modulation scheme, sample-level modulation. By doing this, the phase offsets on the pilot subcarriers are different from that on the data subcarriers. Thus, the phase offsets on the data subcarriers cannot be eliminated by the phase error correction.

To satisfy the second property, we utilize orthogonal coding technique for reliable data transfer. Specifically, to reduce the bit error rate and achieve high reliability, we propose orthogonal pseudo-random noise (PN) sequences to represent the tag data. With the help of the sample-level modulation, each data bit in the PN sequence can represent a phase offset. As a result, the sequences of phase offsets are also orthogonal and the tag data can avoid being eliminated.

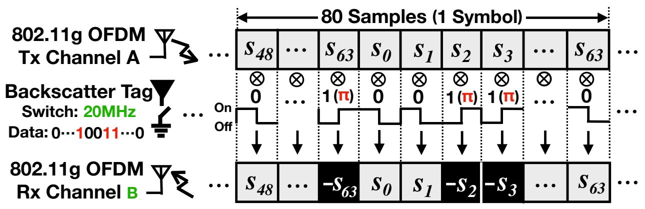


Figure 8: When a switching cycle is equal to a sample duration, the switching action can appropriately change the phase of the sample.

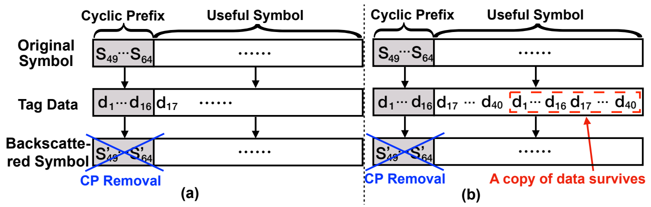


Figure 9: In (a), the tag data embedded in the CP are lost due to the CP removal. In (b), the tag embeds 40 bits of data twice in the symbol. A copy of data can survive after the CP removal.

By satisfying the two properties, compared to prior works [33, 35, 36], our backscatter system can take advantage of the phase error correction on the demodulation side and is able to achieve much lower BER. On the modulation side, since multiple tag data bits can be transmitted within one symbol by using our sample-level modulation, more robust coding methods (such as orthogonal code or convolutional code) than codeword translation can be utilized to further improve the BER.

4 Sample-Level Modulation

Our sample-level modulation has two design goals: i) embedding the tag data in the OFDM signal at the sample-level, and ii) minimizing the interference from the original band channel. In order to achieve the first design goal, we propose to use the tag to change the phase of any sample in the OFDM symbol. To achieve the second design goal, the tag needs to shift the center frequency of the backscattered samples to the adjacent band channel. Fig. 8 shows how TScatter leverages the switching action to change the phase of the sample. For the sake of simplicity, we first assume that the on and off edges of switching actions are aligned with the samples. We discuss the situation that the switching actions and the samples are not edge-aligned in Sec. 6.1.

A fundamental basis for our sample-level modulation is that the OFDM symbol can be divided into the sample-level, whose duration is much shorter than the symbol-level. When 802.11g OFDM generates a symbol in the transmitter, the IDFT converts the 64 subcarriers into a 64-sample sequence. The samples can be represented by rewriting Eqn. 1 with the definition of IDFT:

$$\mathcal{S}_n \;=\; \text{IDFT}\;[\,\mathcal{X}_k\,] \;=\; \frac{1}{64}\sum_{k=0}^{63}\mathcal{X}_k e^{j2\pi kn/64} \tag{8}$$

Where, $n \in \{0,...,63\}$, $\mathcal{S}_n$ denotes the n'th sample and X_k denotes the k'th subcarrier. To prevent inter-symbol interference, the last 16 samples $[\mathcal{S}_{48},\ldots,\mathcal{S}_{63}]$ are replicated in front of the 64 samples. Thus, a symbol consists of a total of 80 samples.

When the tag toggles its RF switch to backscatter the OFDM signals, it essentially uses square waves to modulate the phases of the samples. We use θ_n to represent the initial phase of the n'th square wave $\mathcal{W}_n$. Then, $\mathcal{W}_n$ can be represented using a Fourier series as follows [9]:

$$\mathcal{W}_n(f_s,\theta_n) = 0.5 + \frac{2}{\pi}\sum_{m=1,3,5,...}^{\infty}\frac{1}{m}cos(2\pi m f_s t + \theta_n) \tag{9}$$

Where, f_s is the toggling frequency of the switch.

Since each symbol is 4 μs long and contains 80 samples, when the tag toggles its switch at $f_s = 20MHz$, the switching cycle ($0.05\ \mu s = \frac{1}{20MHz}$) will be equal to the sample duration ($0.05\ \mu s = \frac{4\ \mu s}{80}$). Therefore, the switching action can appropriately modulate the phase of the sample. The n'th backscattered sample can be calculated by multiplying the n'th sample and the 1st harmonic of the n'th square wave:

$$\begin{aligned}\mathcal{S}_n\mathcal{W}_n^{m=1} &= \frac{2}{\pi}\mathcal{S}_n cos(2\pi f_s t + \theta_n)\\ &= \frac{1}{\pi}\{\mathcal{S}_n e^{j\theta_n}e^{j2\pi f_s t} + \mathcal{S}_n e^{-j\theta_n}e^{-j2\pi f_s t}\}\end{aligned} \tag{10}$$

Summary: From Eqn. 10, one can observe that our two design goals are achieved. First, there are two backscattered samples $\mathcal{S}_n e^{j\theta_n}$ and $\mathcal{S}_n e^{-j\theta_n}$. They are formed by changing the phase of the original sample $\mathcal{S}_n$ by θ_n and $-\theta_n$, respectively. If we change θ_n according to the tag data, the tag data is embedded into the sample. For example, if the tag wants to use the 4-phase scheme (0, $\pi/2$, π and $3\pi/2$) to transmit the tag data, we can define the tag data '00' as $\theta_n = 0$, '01' as $\theta_n = \pi/2$, '10' as $\theta_n = \pi$ and '11' as $\theta_n = 3\pi/2$. Second, the two backscattered samples are multiplied by $e^{j2\pi f_s t}$ and $e^{-j2\pi f_s t}$, respectively, which means the backscattered samples are shifted by $f_s = \pm 20MHz$ into the adjacent band channels and thus isolated from the original band channel. Therefore, a OFDM receiver can obtain the backscattered samples without the interference from the original band channel [34]. For these two sidebands, one is desired and the other is unwanted and wasted. The unwanted sideband can be easily eliminated by making the signal have a negative copy on the unwanted sideband as introduced in HitchHike [32].

5 Tag Coding Scheme

In this section, we discuss the coding methods on the tag, which aim to avoid the tag data inversion point being removed by the cyclic prefix removal and the phase offset being eliminated by the phase error correction, and make TScatter achieve highly reliable backscatter communication.

5.1 Surviving from Cyclic Prefix

Since the receiver removes the cyclic prefix (i.e., the first 16 samples of each symbol) directly prior to the procedures in the OFDM module, a portion of the tag data may be removed as well because the tag may embed this data in the cyclic

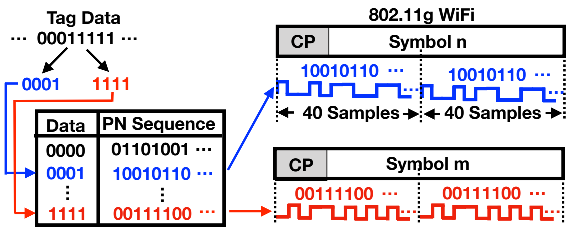


Figure 10: TScatter uses PN sequences to represent the tag data.

prefix. Fig. 9(a) shows an example that the tag data d_1 to d_{16} are lost due to the CP removal.

A naive solution is to utilize an envelope detector to detect the start of the cyclic prefix and embed the tag data in the useful symbol. However, it is unlikely to obtain an accurate CP detection using a packet-level detection on the envelope detector. Instead, TScatter simply embeds 40 bits of tag data twice on 80 samples in a symbol (shown in Fig. 9(b)). By doing this, there still exits a 40-sample sequence which stores a copy of 40 bits of tag data after the CP removal. Hence, the tag does not have to rely on the synchronization accuracy of the envelope detector to avoid the cyclic prefix. This approach can be easily implemented on the tag without extra energy cost or computing resources.

At the receiver side, to decode the 40 bits of tag data in each symbol, we need to know which sample in the symbol is the starting sample of the 40-sample sequence. To find the starting sample, we designed a backscatter header (shown in Appendix B) with a duration of two symbols. The backscatter header is predefined as a flag sequence. Even though a portion of the flag sequence is removed due to the CP removal, there still exits a backscattered symbol whose samples are modulated only by the flag sequence. We add a sliding-window-based algorithm to the decoding algorithm (detailed in Sec. 6.3) to search which part of the flag sequence is the best matching decoding result of this symbol . Then we can obtain the position of the starting sample. When the decoding result matches the two-symbol long predefined backscatter trailer, it signifies the end of the tag data.

5.2 Coding Scheme for Low BER

To avoid the phase offsets being eliminated, TScatter uses predefined nearly orthogonal pseudo-random noise (PN) sequences to represent the tag data (Appendix A lists the PN sequence table). As shown in Fig. 10, the tag data is divided into multiple groups. Each group will be spread to a special 40-bit long PN sequence. Then each bit in the PN sequence is transmitted by using the sample-level modulation. With the help of the sample-level modulation, each bit in the PN sequence can represent a phase offset. As a result, the sequences of phase offsets are orthogonal to each other and the tag data can avoid being eliminated. As described in Sec. 5.1, to avoid being removed by the CP removal, each sequence is transmitted twice. To reduce the coding complexity on the tag, we implant a lookup table of the sequences to map the tag data to the corresponding PN sequence.

Using orthogonal PN sequences to represent the tag data has three benefits. First, since the sequences are orthogonal to each other, the effective signal-to-noise ratio (SNR) of backscattered signals is improved at the receiver side. Second, since the sequences are predefined, the receiver can solve the decoding algorithm by directly finding which sequence is the best-match rather than estimating the sequence, which lowers the computational complexity (detailed in Sec. 6.3). Third, TScatter can be used in lower order modulation schemes of OFDM, such as BPSK or QPSK.

5.3 Coding Scheme for High Throughput

The coding scheme for high throughput is a combination of convolutional code and predefined data when the OFDM sender utilizes higher order modulation schemes, such as 64QAM or 16QAM. The convolutional code is a rate 2/3 feedforward encoder. The predefined data is to increase the estimation accuracy by constricting the estimation result to a smaller range of values during the decoding process (detailed in Sec. 6.3). For 64QAM, in the 40-sample sequence, we use 32 samples to embed the convolutional coded tag data. The remaining 8 samples are modulated by the predefined data. For 16QAM, we use 18 samples to embed the convolutional coded tag data and 22 samples to embed the predefined data.

6 Decoding of Tag Data

One benefit of our sample-level modulation is that it does not require an accurate synchronization with the OFDM sender because the phase offset caused by a lack of accurate synchronization can be corrected by the phase error correction. In this section, we first analyze the phase offset, then describe the OFDM demodulation model and mathematically demonstrate how the phase offset is corrected by the OFDM receiver. In the end, we describe how to decode the tag data.

6.1 Phase Offset Analysis

The phase offset can be characterized by three factors:

• *Envelope Detection Delay.* The tag leverages an envelope detector to identify the WiFi transmission. However, due to an uncertain delayed response, the detector cannot accurately identify when the sample starts. Then it always occurs that the switching actions and the samples are not edge-aligned. We can consider a time delay τ that exists between them and causes a phase offset $\beta = 2\pi f_s \tau$.

• *Wireless Channel.* The wireless channel introduces a phase offset of ϕ to the samples.

• *Frequency Offset.* The frequency offset causes two phenomenas: carrier frequency offset (CFO) and sampling frequency offset (SFO). In Appendix B and C , we demonstrate the CFO and SFO can be estimated and corrected.

Thus, a backscattered sample can be expressed as follows:

$$\mathcal{R}_n = e^{j(\beta+\phi)} e^{j\theta_n} \mathcal{S}_n \qquad (11)$$

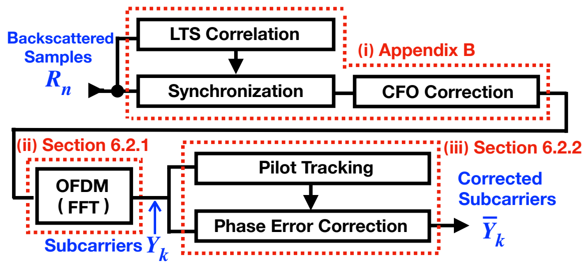


Figure 11: The demodulation process in 802.11g WiFi receiver from the backscattered samples to the corrected subcarriers.

In Sec. 6.2.2, we demonstrate that the phase offset caused by the envelope detection delay and wireless channel can be eliminated and do not affect the values of data subcarriers.

6.2 Demodulation Model

The objective of TScatter's demodulation is to decode the tag data on OFDM subcarriers. To achieve this objective, we need to understand how the OFDM receiver converts the backscattered samples to the subcarriers. To do so, we divide the workflow of an 802.11g receiver into 3 parts (shown in Fig. 11), and build models for these parts.

6.2.1 Modeling OFDM

The 80 backscattered samples of each symbol first go through the CP removal module, where the first 16 samples are removed. Then the remaining 64 samples are applied to the DFT to generate the complex values of OFDM subcarriers. The subcarrier's value can be derived as:

$$Y_k \;=\; \text{DFT}\;[\;\mathcal{R}_n\;] \;=\; e^{j(\beta+\phi)}\sum_{n=0}^{63} e^{j\theta_n}\mathcal{S}_n e^{-j2\pi nk/64} \tag{12}$$

Where, $k \in \{0,\dots,63\}$ and Y_k is the complex value of the k'th OFDM subcarrier. Eqn. 12 shows that the tag data is transfered from the backscattered samples in time domain to the subcarriers in frequency domain by OFDM.

6.2.2 Modeling Phase Error Correction

While the tag data is transfered into the subcarriers, the phase offset $\beta+\phi$ is also transfered into the subcarriers (shown in Eqn. 12), which may increase the tag data decoding error. We had an important observation that the OFDM receiver can leverage pilot subcarriers to eliminate the phase offset. In this section, we mathematically demonstrate the procedure of phase error correction.

The phase error estimation is performed by calculating the rotating phase of the four pilot subcarriers. In the sender, the four pilot subcarriers are $\{\mathcal{X}_{11},\mathcal{X}_{25},\mathcal{X}_{39},\mathcal{X}_{53}\}$ and their initial values are $\{1,1,1,-1\}$ [2]. In the receiver, we use $\{Y_{11},Y_{25},Y_{39},Y_{53}\}$ (shown in in Eqn. 12) to represent the four pilot subcarriers. The rotating phase of the pilot subcarriers can be estimated as:

$$\begin{aligned}\Psi &= \angle(Y_{11}+Y_{25}+Y_{39}-Y_{53})-\angle(\mathcal{X}_{11}+\mathcal{X}_{25}+\mathcal{X}_{39}-\mathcal{X}_{53})\\ &= \beta+\phi+\angle\Gamma \end{aligned}\tag{13}$$

Where,
$$\Gamma = \sum_{n=0}^{63} e^{j\theta_n}\mathcal{S}_n\Big(\sum_{k=11,25,39} e^{-j2\pi nk/64} - e^{-j2\pi n\frac{53}{64}}\Big) \tag{14}$$

From Eqn. 13 and 14, we observe that a new phase offset $\angle\Gamma$ is introduced which is caused by the backscatter modulation on the pilot subcarriers. Eqn. 13 also illustrates that although the backscatter modulation change the pilot subcarriers, the phase offset $\beta+\phi$ is isolated from the tag data.

Since 48 out of the 64 subcarriers are used to carry the payload data ($k \in \{6,\dots,58\}$ and $\notin \{11,25,32,39,53\}$), the OFDM receiver multiplies 48 data subcarriers by $e^{-j\Psi}$ to correct the phase offset:

$$\begin{aligned}\overline{Y}_k &= e^{-j\Psi}\,Y_k \\ &= e^{-j\angle\Gamma}\,\cancel{e^{-j(\beta+\phi)}}\,\cancel{e^{j(\beta+\phi)}}\sum_{n=0}^{63} e^{j\theta_n}\mathcal{S}_n e^{-j2\pi nk/64}\end{aligned}\tag{15}$$

Where,
$$e^{-j\angle\Gamma} = \frac{Re\{\Gamma\} - j*Im\{\Gamma\}}{\sqrt{Re\{\Gamma\}^2+Im\{\Gamma\}^2}} \tag{16}$$

Eqn. 15 demonstrates that the phase offset $\beta+\phi$ is eliminated and does not affect the tag data decoding. On the other hand, Eqn. 15 also demonstrates that the values of data subcarriers were affected twice: The first one happens during the backscatter modulation stage, when the values of the data subcarriers were changed according to the tag data; the second one happens during the phase error correction stage, when the values of data subcarriers were corrected by the phase offset calculated from pilot subcarriers. Although Γ exists in Eqn. 15, Eqn. 14 shows that the only unknown value in Γ is the tag data. Therefore, we can leverage Eqn. 15 to decode the tag data without being affected by the phase offset $\beta+\phi$.

6.3 Decoding Tag Data

Eqn. 15 shows that the phase change on each subcarrier is different and determined by all 64 backscattered samples. If the tag data on any backscattered sample change, the phase change on each subcarrier changes. Therefore, we must use all subcarriers' information to decode each tag data.

We observe that the inputs of Eqn. 15 are the tag data θ_n and the original OFDM sample $\mathcal{S}_n$, while the outputs of Eqn. 15 are the corrected data subcarrier $\overline{Y}_k$. Hence, if we obtain the values of $\mathcal{S}_n$ and $\overline{Y}_k$, we can decode θ_n.

The values of original OFDM sample $\mathcal{S}_n$ can be calculated by using Eqn. 8. However, the challenge is that we cannot obtain the real value of $\overline{Y}_k$. Normally, OFDM backscatter uses the coded data linear transform technique in [33, 35, 36] to track the phase change on the subcarriers. If we follow the same technique to get the subcarrier values, we find that the subcarrier values deviate from $\overline{Y}_k$. This is because the OFDM receiver maps $\overline{Y}_k$ to the nearest QAM points (we assume $\overrightarrow{Y}_k$).

Since $\overrightarrow{Y}_k$ are the nearest QAM points of $\overline{Y}_k$, we can estimate the tag data θ_n by calculating the minimum Euclidean distance between the corrected data subcarrier values $\overline{Y}_k$ and their mapped points $\overrightarrow{Y}_k$:

$$\begin{bmatrix} \tilde{\theta}_0 \\ \vdots \\ \tilde{\theta}_{63} \end{bmatrix} = \underset{[\theta_0,\dots,\theta_{63}]}{\arg\min} \sum_{\substack{k=6 \\ k\neq 11,25,32,39,53}}^{58} ||\overrightarrow{Y}_k - \overline{Y}_k||_2 \tag{17}$$

$$= \underset{[\theta_0,\dots,\theta_{63}]}{\arg\min} \sum_{\substack{k=6 \\ k\neq 11,25,32,39,53}}^{58} ||\overrightarrow{Y}_k - e^{-j\angle\Gamma} \sum_{n=0}^{63} e^{j\theta_n} S_n e^{-j2\pi nk/64}||_2$$

Where, k are the indexes of the data subcarriers, $\overrightarrow{Y}_k$ are provided by the receiver and Γ can be represented using Eqn. 14. The only unknown value in Eqn. 17 is the tag data θ_n. Therefore, we can leverage Eqn. 17 to decode the tag data.

Since each tag data θ_n has limited values (i.e., either 0 or π), Eqn. 17 is a constrained linear least-squares problem. We note that the left side of Eqn. 17 is 64 unknown values $(\tilde{\theta}_0,\dots,\tilde{\theta}_{63})$, while we only have 48 data subcarrier values $\overrightarrow{Y}_k$ ($k \in \{6,...,58\}$ and $\notin \{11,25,32,39,53\}$) on the right side. That means Eqn. 17 is not full row-rank. Mathematically, to determine a unique solution using Eqn. 17, we need to minimize the number of unknown values on left side to 48. A simple way to do this is that the number of unknown tag data in each symbol is no more than 48 and the remaining tag data are predefined. For example, we predefine $e^{j\theta_{48,...,63}} = e^{j0} = 1$ and calculate the unknown data $e^{j\theta_{0,...,47}}$. Therefore, the left side (the unknown tag data) and the right side (the data subcarrier values) have the same size, and Eqn. 17 is now full row-rank. In fact, Eqn. 17 illustrates that the maximum number of samples used to store the unknown tag data in each symbol is 48. This is because we rely on the data subcarriers' values to calculate the unknown tag data and each symbol provides only 48 data subcarriers. The coding scheme in Sec. 5.1 requires that the number of the tag data embedded in each symbol is no more than 40, which is also less than 48.

Eqn. 17 is solved by using the matrix decomposition, whose computational complexity is $O(n^3)$ [3], where n is the number of the unknown tag data. When n gets to the maximum value (i.e., 48), there are $n^3 = 110592$ floating-point operations. A low power edge computing platform Jetson Nano [21] (only cost \$99) implemented on an ARM A57 processor with 472G floating-point operations per second, can solve the problem in $\frac{110592}{472*10^9} = 0.25\mu s$, which is less than a symbol duration $4\mu s$.

7 Lite Version of TScatter

In prior works [33, 35, 36], the codeword translation works only when the phase error correction is disabled. In this section, we demonstrate that TScatter can also use the codeword translation at the symbol level without disabling the phase error correction and introduce the lite version of TScatter.

Prior works use the symbol-level modulation (i.e., phase change once per symbol) to realize the codeword translation. Based on our observation from Sec. 3.3, when the inversion point settles in the useful symbol, the corrected subcarriers are different from the original ones. Since the symbol-level modulation is essentially a special case of the sample-level modulation (i.e., multiple phase changes per symbol) by setting the number of phase changes as 1 within one symbol, it is possible to use the codeword translation to achieve the basic idea of TScatter without disabling the phase error correction. We call this approach Lite TScatter because its design principle is derived from our sample-level modulation by setting all the tag data values inside a symbol to be all-one or all-zero.

Specifically, the whole procedure of Lite TScatter is as follows: before the tag transmits the data, it first transmits a long all-one (or all- zero) sequence to reflect the preamble and several symbols. Since there is no phase change in the long sequence, the corrected subcarriers are the same as the original ones. Then, the tag transmits the first data. The first data is predefined, and its value should be different from that of the long sequence. For example, if the long sequence is all-one, the first data should be defined as zero. Since there is a phase change between the long sequence and the first data, we can find the first symbol which is different from the original one. Next, the tag can transmit arbitrary data. If the corrected subcarriers are the same as the original ones, it implies there is no phase change and the tag data should be equal to the last one. If the corrected subcarriers are different from the original ones, it implies there is a phase change and the data should be different from the last one. Therefore, we can decode the tag data one by one. By modifying the codeword translation decoder in Eqn. 5, the decoding function of Lite TScatter is as follows:

$$\text{Tag data } d_n = \begin{cases} d_{n-1} & \mathcal{X}_k e^{j\delta_k} \oplus \mathcal{X}_k = 0 \\ 1-d_{n-1} & \mathcal{X}_k e^{j\delta_k} \oplus \mathcal{X}_k \neq 0 \\ \text{Predefined} & n=1 \end{cases} \tag{18}$$

Eqn. 18 provides a lite version of TScatter, which enables codeword translation work with WiFi protocols that enable the phase error corrections. However, Lite TScatter's throughput is relatively low because the tag data is modulated at the symbol level instead of the sample level.

8 Evaluation

In this section, we present the implementation of TScatter and show the experimental results of our extensive evaluation.

8.1 Implementation

TScatter (shown in Fig. 12) is implemented on an open-source backscatter platform [29], which mainly consists of two components: a Microsemi AGLN250 low power FPGA and an ADG902 RF reflective switch. To compare different levels of

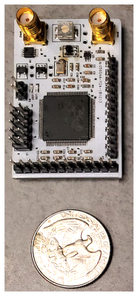


Figure 12: TScatter Tag

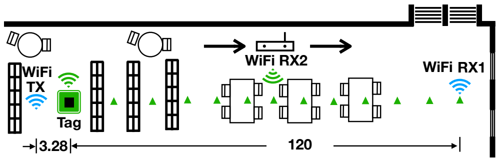


Figure 13: Library (in feet)

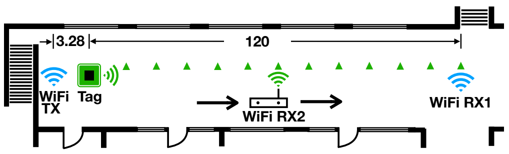


Figure 14: Hallway (in feet)

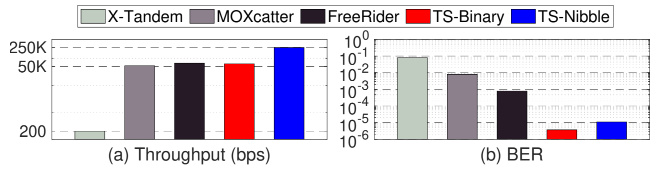


Figure 15: Comparing with state-of-the-art with similar throughput

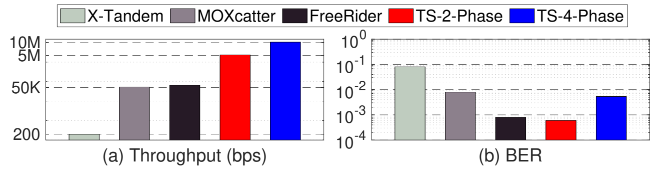


Figure 16: Comparing with state-of-the-art with similar BER

coding and modulation schemes, we implemented both the coding schemes of **binary** and **nibble** (i.e., 4-bit) (shown in Appendix A) and the modulation schemes of **2-phase** and **4-phase** (introduced in Sec. 4) in the FPGA.

To conduct fair comparisons, the distance between the WiFi transmitter and the backscatter tag is 1 meter (i.e., 3.28 feet), which is the same as the one in FreeRider [33]. The WiFi transmitter is implemented on a ThinkPad T420s laptop which transmits the IEEE 802.11g compatible signals. The WiFi transmitter utilizes 64QAM as its modulation scheme by default. In Sec. 8.6, we evaluate TScatter's performance with different WiFi modulation schemes.

To extensively evaluate TScatter's performance, the experiments were conducted in three different scenarios: **Library** (Fig. 13), **Hallway** (Fig. 14), and **Stadium**. The WiFi receivers are implemented using DELL XPS 9550 laptop and USRP-B210 with 802.11g PHY layer. WiFi receiver 1 is used to obtain the original WiFi information. WiFi receiver 2 listens for the backscattered WiFi signal. To assess the system performance, we use the following two metrics:

Throughput: The throughput is defined as correctly demodulated data bits at PHY layer.

Bit error rate (BER): The BER is defined as the number of bit errors divided by the total number of transmitted bits.

8.2 Comparing with State-of-the-art

We first compare the performance of TScatter with the reported best results produced by recent OFDM-based WiFi backscatter systems (i.e., X-Tandem [35], MOXcatter [36] and FreeRider [33]). Fig. 15 shows BER results when TScatter and state-of-the-art solutions have similar level of throughput. As we can observe from this figure, the BERs of TScatter are much lower than that of the state-of-the-art solutions. Fig. 16(a) shows the comparison of throughput. We can observe that, with 4-phase modulation scheme, TScatter achieves 10.61 Mbps. The main reason TScatter achieves orders of magnitude better performance is that it takes advantage of the phase error correction. State-of-the-art solutions require specific commodity NICs that can disable the phase error correction. However, the phase error correction is very important for the OFDM demodulation because the phase error may be dynamically changing due to the changes of environment (e.g., temperature of the oscillators and object movement). Without the phase error correction, the OFDM demodulation error will increase. Similarly, disabling phase error correction will also introduce phase error into the decoding procedure of the codeword translation, which will increase the tag data decoding error. In contrast, TScatter takes the phase error correction into consideration. On the demodulation side, TScatter builds the demodulation model from the backscattered samples to the corrected subcarriers that can not only correct the phase error due to the dynamic environments but also decode the tag data by estimating the minimal Euclidean distance rather than by using XOR. On the modulation side, TScatter leverages the sample-level modulation to apply more robust coding methods to further improve the BER or throughput. As a result, the BERs of TScatter are around $\mathbf{10^2}$, $\mathbf{10^3}$ and $\mathbf{10^4}$ **times** as low as that of FreeRider, MOXcatter and X-Tandem, respectively. The throughputs of TScatter are 53,050, 212, and 176 times higher than X-Tandem, MOXcatter, and FreeRider, respectively.

8.3 Library

To understand how TScatter works in a multipath rich environment, we conducted the experiments in a library (shown in Fig. 13). Since there are a lot of shelves, tables, and chairs in the library, these obstacles not only block the direct line-of-sight transmission path but create more multipath effects as well. In this setting, the backscatter tag and WiFi transmitter are deployed on the shelves. We move the WiFi receiver from shelves to tables to vary the multipath environment.

Fig. 17 and Fig. 18 show the throughput and BER over different communication distances, respectively. As we can observe from these figures, the throughput of TScatter is similar over the change of distances while the BER increases slowly. When the communication distance reaches 120 feet, the average BERs of TScatter are still around 10^{-3} and 10^{-4} for Nibble and Binary, respectively. Since the multipath effect will become more severe as the distance increases in the library scenario, we can conclude that TScatter is resistant to the wireless channel interference. Because the demodulation process (described in Sec. 6.1) takes the phase error intro-

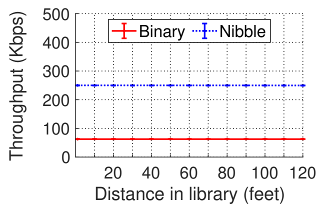


Figure 17: Throughput

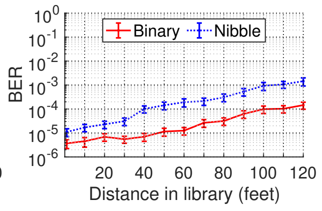


Figure 18: BER

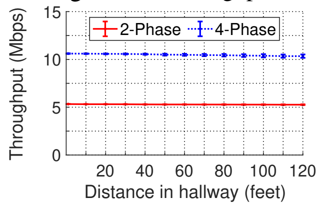


Figure 19: Throughput

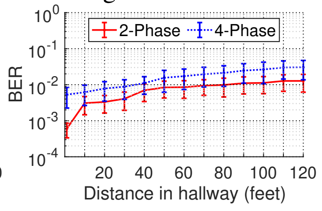


Figure 20: BER

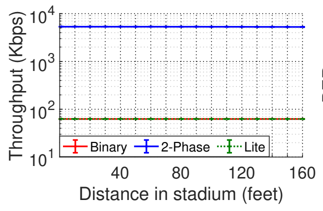


Figure 21: Throughput

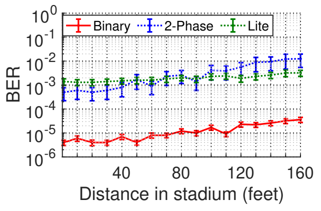


Figure 22: BER

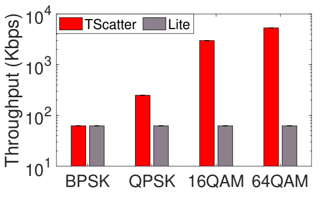


Figure 23: Throughput v.s. WiFi OFDM modulation scheme

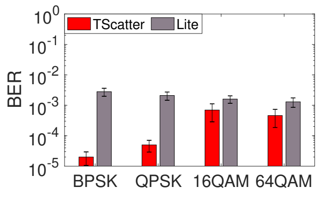


Figure 24: BER v.s. WiFi OFDM modulation scheme

duced by the wireless channel into consideration, TScatter shows great advantages in this scenario.

8.4 Hallway

In this section, we evaluate TScatter in the hallway scenario. As shown in Fig. 14, the hallway is on the second floor of an academic building. This scenario is selected to reflect TScatter's real-world performance with less environmental impact because the hallway is relatively spacious. In this scenario, the backscatter tag and the WiFi receiver were kept in line-of-sight. The distance between the tag and the receiver is changing from one foot to 120 feet.

Fig. 19 shows the throughput of our TScatter system using different modulation schemes (i.e., 2-phase and 4-phase). Overall, TScatter can achieve a maximum of 10.61 Mbps and the throughput is stable over different communication distances because it has a high granularity modulation scheme. Fig. 20 shows the BER under different communication distances. We can observe that the BER for the 2-phase modulation scheme is very low. The average BER is lower than 1% when the distance increases to 70 feet. Even when the distance increases to 120 feet, the average BER is still around 1%. This is because our demodulation model captures the impact of the phase error correction on the received subcarriers.

8.5 Outdoor Stadium

To understand the performance of TScatter with different modes (i.e., low BER, high throughput, and lite version) in the same scenario, we conduct experiments in an outdoor stadium. The distance between the tag and the receiver is changing from one foot to 160 feet.

Fig. 21 shows the throughput over different communication distances. The high throughput mode (2-phase) outperforms the low BER mode (binary) and Lite TScatter because it has a high granularity modulation scheme. When the receiver is 160 feet away from the tag, the throughput of binary, 2-phase, and Lite TScatter are around 62.5 Kbps, 5.26 Mbps and 62.3 Kbps, respectively. Fig. 22 shows the BERs at different distances. The BER of low BER mode and Lite TScatter is more stable than that of high throughput mode despite the fact that the signal strength degrades across distance. Specifically, the BER of low BER mode is much less than that of the other two modes. Therefore, the low BER mode has the significant advantage on providing reliable backscatter communications. When the distance reaches 160 feet, the average BERs of binary, 2-phase, and Lite TScatter are still around 10^{-5}, 10^{-2} and 10^{-3}, respectively.

8.6 Impact of OFDM Modulation Scheme

In Sec. 5.2 and 5.3, we introduced how TScatter deals with lower and higher level WiFi OFDM modulation schemes respectively. In this section, we compare sample-level modulation TScatter's performance (in terms of throughput and BER with low BER or high throughput coding scheme) with Lite TScatter (which uses symbol level modulation) under different WiFi OFDM modulation schemes (i.e., BPSK, QPSK, 16QAM, and 64QAM). Fig. 23 shows the throughput result. We observe that the throughput of Lite TScatter is almost the same under different OFDM modulation schemes because one bit of data is embedded in per symbol no matter what the OFDM modulation scheme is. For TScatter, the throughput increases while the OFDM modulation scheme changes from the lowest level (i.e., BPSK) to the highest level (i.e., 64QAM). At 64QAM, the throughput reaches 5.33 Mbps. The reason of this growing trend is that higher level OFDM modulation schemes can provide higher resolution to decode the tag data, which can accommodate more tag data per symbol.

Fig. 24 shows the result of BER versus OFDM modulation scheme. We can observe that TScatter's BER is lower than Lite TScatter's across all kinds of modulation schemes even when the OFDM transmitter utilizes BPSK. This is because the demodulation model captures the impact of the phase error correction on the received subcarriers. Moreover, TScatter leverages the sample-level modulation to apply more robust coding methods to further improve the BER.

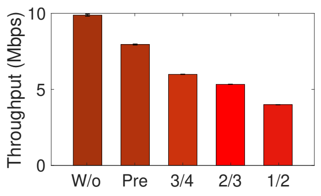


Figure 25: Throughput v.s. Tag Coding Scheme

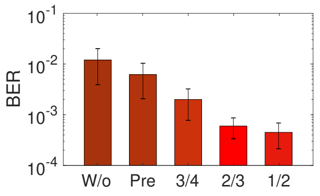


Figure 26: BER v.s. Tag Coding Scheme

8.7 Impact of Tag Coding Scheme

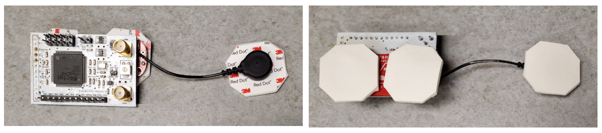
In Sec. 5.3, we have described TScatter's coding schemes for high throughput. Fig. 25 and 26 show TScatter's performance with different code rates. "W/o" denotes the results without using coding scheme. "Pre" denotes the results that TScatter only uses predefined data without coding. "3/4", "2/3", and "1/2" denote the results that TScatter uses both predefined data and coding with a code rate at $3/4$, $2/3$, and $1/2$, respectively. Since the experiments show similar trends, we show the results with the 2-phase modulation scheme in the hallway scenario. The WiFi transmitter utilizes 64QAM.

When no coding scheme is utilized, the whole 40-sample sequence is used to store the unknown tag data (described in Sec. 5.1). Consequently, TScatter has the highest throughput (shown in Fig. 25) as well as the highest BER (shown in Fig. 26) among all coding schemes. The reason of the high BER is that the corrected data subcarrier values deviate from their mapping points, which cause estimation errors in Eqn. 17. To reduce the estimation errors, we replace 8 unknown tag data with the predefined data to constrict the estimation result to a smaller range of values. Fig. 26 shows that the BER decreases to below 1%. To achieve a more reliable communication, we combine the predefined data and the convolutional code to generate the tag data. When the code rate is up to 2/3, the BER is lower than 0.1% and the throughput is as high as 5.33 Mbps, which provides a reliable high throughput communication.

8.8 Energy Consumption Analysis

We designed an integrated circuit for TScatter's digital processing module and conducted a simulation using Cadence Spectre for TSMC 65nm process. In the power consumption simulation, multiple factors are considered including Vdd (1.6V $\sim$ 2V), temperature (25°C), system clock frequency, and power mode.

The IC design consists of four modules: clock, energy detector, control module and RF transistor. TScatter first detects the WiFi packet by using the energy detector. When the WiFi packet is detected, the clock synthesizes a 20MHz frequency for the control module. Then the control module reads the data from the sensor, latches and codes the sensor data, and toggles the RF transistor according to the coded data. Since the power consumption of the energy detector and the RF transistor is very low (around $0.3\mu W$), we mainly introduce the implementation details of the clock and the control module:



(a) Front (b) Back

Figure 27: EMG connected TScatter tag

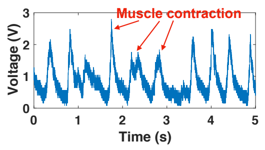
Clock. The bottleneck of the power consumption of the TScatter system is the clock. We use a ring oscillator to synthesize a 20MHz frequency. The frequency accuracy is sensitive to temperature change. We add a thermistor to design a temperature compensation circuit to compensate for the frequency drift. Moreover, we add a trimming pad to correct the frequency. The simulation result shows that the power consumption of the ring oscillator is $23.7\mu W$.

Control module. The control module contains a cache circuit to store the sensor data, a code circuit to code the sensor data, and an inverter to toggle the RF transistor as a fan-out. The cache is constructed by the basic cache cell. The cache cell is composed of one D latch and one transmission gate. Through connecting the output of D latch to the input of the transmission gate, a cache cell with simultaneous read and write capability is created. The reason we use the latches rather than flash to store the sensor data is that the power consumption of latches is very low even when they are working at 20MHz. The code circuit contains a lookup table for low BER mode, which is composed of 32 PN sequences (shown in Appendix B), and a convolutional encoder for high throughput mode, which is composed of 7 latches and 3 XOR gates. The lookup table is also constructed by basic logic gates. The inverter is minimum sized. The simulation result shows that the power consumption of the control module is around $6.2\mu W$.

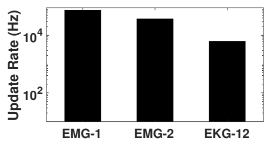
From the above analysis, the overall power consumption of TScatter is **30.2μW**, which is similar to that of other OFDM backscatter systems. This is mainly because the most power-hungry part of the backscatter tag is the clock. No matter the sample-level modulation or the symbol-level modulation, they all need to shift the incoming WiFi signal to the adjacent band channel to minimize the interference from the original band channel, which can be done by toggling the RF switch at a specific frequency. The minimum value of the frequency equals a WiFi channel bandwidth, i.e., 20MHz. Consequently, to toggle the RF switch at 20MHz, the logic control modules of the symbol-level modulation and the sample-level modulation are all operating at 20MHz. Although TScatter performs the convolutional coding in high throughput mode, the encoder which is constructed by 7 D latches and 3 XOR gates has a very low power consumption even when it works at 20MHz. Therefore, the energy consumptions of TScatter and other OFDM backscatter systems are similar.

9 Application

In this section, we show that biometric measurements (such as EMG and EKG) can be wirelessly transmitted in real-time by using TScatter. Fig. 27(a) (front side) and Fig. 27(b)



(a) EMG data



(b) Update Rate (log scale)

Figure 28: (a) shows the EMG data while the system is attached to arm muscle and the participate is doing dumbbell lifting. Each peak corresponds to a muscle contraction. (b) shows the average update rate when the system is connected with 1-channel EMG (sleeping monitoring), 2-channel EMG (fitness monitoring), and 12-channel EKG.

(back side) show a prototype that an EMG sensor that is connected to the TScatter tag. By using foam electrodes, the tag and sensor can be attached to a muscle to record the electrical activity produced by the muscle. Fig. 28(a) shows one set of data while the system is monitoring arm activities. In this figure, each peak corresponds to a muscle contraction. Fig. 28(b) shows the average update rate under different configurations: i) EMG-1, 1-channel EMG is attached to a leg muscle for monitoring sleep abnormalities [5]; ii) EMG-2, 2-channel EMG is attached to upper and lower arm for fitness monitoring; and iii) EKG-12 , 12-channel EKG is generated to test how TScatter works with multi-channel EKG. With a typical sampling rate at 1 KHz and 24-bit resolution, the required bit rate for 12-channel EKG is 288 Kbps. With different modulation schemes, our TScatter can provide around 5 ~10 Mbps throughput at the physical layer. Even after deducting the performance loss due to the wireless signal attenuation caused by the human body and upper layer overheads (e.g., ACKs), TScatter's throughput should be sufficient for 12-channel EKG measurements.

10 Related Work

Recently, backscatter has played an attractive role in the passive communication field because of its significant performance on power consumption. Researchers have proposed various kinds of novel techniques [8, 10, 17–20, 26, 30, 31] to support various applications. For example, ReMix [26] enables backscattering of deep tissue devices by overcoming interference from the surface of a human body and localizing the in-body backscatter device. Living IoT [8] enables smart farming applications by placing backscatter devices on live insects. NICscatter [31] introduces a backscatter communication method on commercial Wi-Fi NICs that enables malware to covertly convey information. PAB [10] enables long-term ocean applications by backscattering acoustic signals in underwater environments.

To leverage existing infrastructures, such as TV band [16, 22], FM radio [27], LoRa [7, 23, 24], Bluetooth [9], ZigBee [13], and WiFi [6, 11, 12, 32–37], researchers have also explored various solutions. Specifically, in the WiFi backscatter field, Passive WiFi [12] demonstrates for the first time that the backscatter can generate 802.11b transmissions. Interscatter [9] shows that Bluetooth transmissions can be used to create 802.11b transmissions using backscatter communication. HitchHike [32] allows a tag to backscatter existing 802.11b transmissions from a commodity WiFi transmitter. To further support more complex WiFi structures, such as OFDM, WiFi Backscatter [11] conveys information by modulating the CSI and RSSI measurements at the OFDM receivers. BackFi [6] uses customized full-duplex devices to clean out the effect during backscattering OFDM signals. FS-Backscatter [34] minimizes the interference from the original band by shifting the backscattered OFDM signals to an adjacent band. FreeRider [33] proposes the OFDM-based codeword translation technique, which piggybacks the tag data by changing the phase of backscattered OFDM symbols. Built on top of FreeRider, MOXcatter [36] and X-Tandem [35] realize a MIMO OFDM backscatter system and a multi-hop OFDM backscatter system, respectively. However, FreeRider, MOXcatter, and X-Tandem require specific WiFi receivers that can disable the phase error correction.

Different from the above backscatter systems, TScatter is the first work that conducts sample-level coding and subcarrier-level decoding based on OFDM waveforms. More importantly, it is able to achieve orders of magnitude lower bit error rate or much higher throughput than existing approaches.

11 Discussion & Conclusion

Although our system was tested using USRPs, we believe that our Lite TScatter can potentially be deployed on commodity WiFi devices with little modification because it is a natural extension of FreeRider and MOXcatter. Based on the description of prior works and WiFi specification [2, 14, 33, 35, 36], to implement the full version of TScatter on commodity devices, we expect to address the following engineering problems: 1) Since the backscattered WiFi packets have been modified, commodity WiFi receivers may drop these packets due to their failure to pass the cyclic redundancy check (CRC). To address this problem, we need to use a WiFi card that can be configured into monitor mode which can obtain packets with bad checksums [33]; 2) we need to know the scrambling seed (i.e., the initial state of shift register) to decode the tag data. For different WiFi cards, we need to take different approaches. For example, in ath5k supported WiFi cards (e.g., Atheros AR5112 and AR2425), the scrambling seed can be derived by setting the register of the driver [14].

We note that the main contribution of this paper is a novel backscatter design principle (i.e., sample-level modulation) for OFDM backscatter. We extensively evaluated the performance of TScatter in various real-world scenarios. Evaluation results demonstrate that TScatter is able to achieve orders of magnitude lower bit error rate or much higher throughput than existing approaches [33, 35, 36].

Acknowledgments. This work is in part supported by NSF grants CNS-1824491 and CNS-1652669. We thank our shepherd Dr. Shyam Gollakota and other reviewers for their valuable comments.

References

[1] https://github.com/bastibl/gr-ieee802-11.

[2] Ieee standard for information technology–telecommunications and information exchange between systems local and metropolitan area networks–specific requirements - part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications. IEEE Std 802.11-2016 (Revision of IEEE Std 802.11-2012) (Dec 2016).

[3] Computational complexity of mathematical operations — Wikipedia. https://en.wikipedia.org/wiki/Computational_complexity_of_mathematical_operations, 2019. [Online].

[4] BAHL, V. Edge computing: a historical perspective and direction.

[5] BASTUJI, H., AND GARCA-LARREA, L. Sleep/wake abnormalities in patients with periodic leg movements during sleep: factor analysis on data from 24-h ambulatory polygraph. Journal of Sleep Research.

[6] BHARADIA, D., JOSHI, K. R., KOTARU, M., AND KATTI, S. Backfi: High throughput wifi backscatter.

[7] HESSAR, M., NAJAFI, A., AND GOLLAKOTA, S. Netscatter: Enabling large-scale backscatter networks. NSDI '19.

[8] IYER, V., NANDAKUMAR, R., WANG, A., FULLER, S. B., AND GOLLAKOTA, S. Living iot: A flying wireless platform on live insects. MobiCom'19.

[9] IYER, V., TALLA, V., KELLOGG, B., GOLLAKOTA, S., AND SMITH, J. Inter-technology backscatter: Towards internet connectivity for implanted devices. SIGCOMM '16.

[10] JANG, J., AND ADIB, F. Underwater backscatter networking. SIGCOMM '19.

[11] KELLOGG, B., PARKS, A., GOLLAKOTA, S., SMITH, J. R., AND WETHERALL, D. Wi-fi backscatter: Internet connectivity for rf-powered devices. SIGCOMM '14.

[12] KELLOGG, B., TALLA, V., GOLLAKOTA, S., AND SMITH, J. R. Passive wi-fi: Bringing low power to wi-fi transmissions. SIGCOMM '16.

[13] LI, Y., CHI, Z., LIU, X., AND ZHU, T. Passive-zigbee: Enabling zigbee communication in iot networks with 1000x+ less power consumption. SenSys'18.

[14] LI, Z., AND HE, T. Webee: Physical-layer cross-technology communication via emulation. In Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking, MobiCom '17.

[15] LING, P., LU, F., SNOEREN, A. C., AND VOELKER, G. M. Enfold: Downclocking ofdm in wi-fi. GetMobile: Mobile Comp. and Comm..

[16] LIU, V., PARKS, A., TALLA, V., GOLLAKOTA, S., WETHERALL, D., AND SMITH, J. R. Ambient backscatter: Wireless communication out of thin air. SIGCOMM '13.

[17] LIU, X., CHI, Z., WANG, W., YAO, Y., AND ZHU, T. Vmscatter: A versatile MIMO backscatter. NSDI'20.

[18] LUO, Z., ZHANG, Q., MA, Y., SINGH, M., AND ADIB, F. 3d backscatter localization for fine-grained robotics. In 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), NSDI'19.

[19] NADERIPARIZI, S., HESSAR, M., TALLA, V., GOLLAKOTA, S., AND SMITH, J. R. Towards battery-free HD video streaming. NSDI '17.

[20] NANDAKUMAR, R., IYER, V., AND GOLLAKOTA, S. 3d localization for sub-centimeter sized devices. Sensys'18.

[21] NVIDIA. Jetson nano brings ai computing to everyone. https://devblogs.nvidia.com/jetson-nano-ai-computing/, 2019.

[22] PARKS, A. N., LIU, A., GOLLAKOTA, S., AND SMITH, J. R. Turbocharging ambient backscatter communication. SIGCOMM '14.

[23] PENG, Y., SHANGGUAN, L., HU, Y., QIAN, Y., LIN, X., CHEN, X., FANG, D., AND JAMIESON, K. Plora: A passive long-range data network from ambient lora transmissions. SIGCOMM '18.

[24] TALLA, V., HESSAR, M., KELLOGG, B., NAJAFI, A., SMITH, J. R., AND GOLLAKOTA, S. Lora backscatter: Enabling the vision of ubiquitous connectivity. Proc. ACM Interact. Mob. Wearable Ubiquitous Technol..

[25] VASISHT, D., KUMAR, S., RAHUL, H., AND KATABI, D. Eliminating channel feedback in next-generation cellular networks. SIGCOMM'16.

[26] VASISHT, D., ZHANG, G., ABARI, O., LU, H.-M., FLANZ, J., AND KATABI, D. In-body backscatter communication and localization. SIGCOMM '18.

[27] WANG, A., IYER, V., TALLA, V., SMITH, J. R., AND GOLLAKOTA, S. Fm backscatter: Enabling connected cities and smart fabrics. NSDI'17.

[28] XIE, Y., LI, Z., AND LI, M. Precise power delay profiling with commodity wifi. MobiCom '15.

[29] XU, C., AND ZHANG, P. Open-source software and hardware platforms for building backscatter systems. GetMobile: Mobile Comp. and Comm..

[30] XU, X., SHEN, Y., YANG, J., XU, C., SHEN, G., CHEN, G., AND NI, Y. Passivevlc: Enabling practical visible light backscatter communication for battery-free iot applications. MobiCom'17.

[31] YANG, Z., HUANG, Q., AND ZHANG, Q. Nicscatter: Backscatter as a covert channel in mobile devices. MobiCom '17.

[32] ZHANG, P., BHARADIA, D., JOSHI, K., AND KATTI, S. Hitchhike: Practical backscatter using commodity wifi. SenSys '16.

[33] ZHANG, P., JOSEPHSON, C., BHARADIA, D., AND KATTI, S. Freerider: Backscatter communication using commodity radios. CoNEXT '17.

[34] ZHANG, P., ROSTAMI, M., HU, P., AND GANESAN, D. Enabling practical backscatter communication for on-body sensors. SIGCOMM '16.

[35] ZHAO, J., GONG, W., AND LIU, J. X-tandem: Towards multihop backscatter communication with commodity wifi. MobiCom '18.

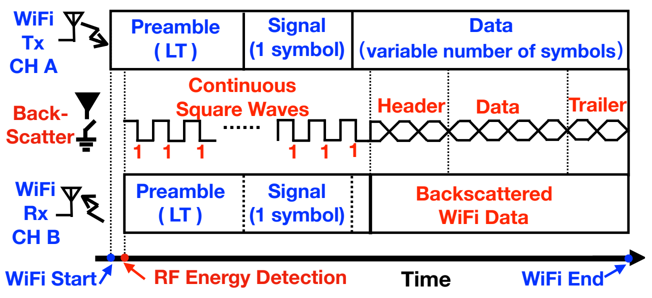


Figure 29: Synchronization between the WiFi sender and the WiFi receiver. The backscatter packet contains four parts: Continuous Square Waves (CSW), Header (in Sec. 5.1), Data and Trailer (in Sec. 5.1). CSW is only a series of square waves and used to reflect the preamble without embedding tag data.

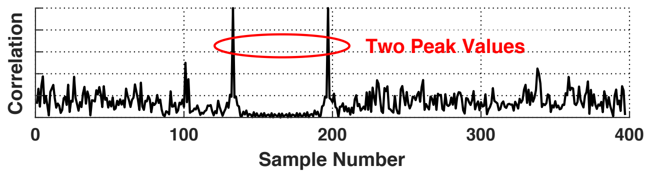


Figure 30: By obtaining the indexes of two peak values, the WiFi receiver can achieve a precise synchronization.

[36] ZHAO, J., SGONG, W., AND LIU, J. Spatial stream backscatter using commodity wifi. MobiSys '18.

[37] ZHAO, R., ZHU, F., FENG, Y., PENG, S., TIAN, X., YU, H., AND WANG, X. Ofdma-enabled wi-fi backscatter. MobiCom '19.

Appendix A Map Tag Data to PN Sequence

No.	Tag Data	PN Sequence
1	0000	01101001111110111011001100111101010010 10
2	0001	1001011011011101000100000101111110001010
3	0010	1000101101010011011010011111001010011000
4	0011	1111000001000111010111101010000010001111
5	0100	0010100001000100100011011001111011110111
6	0101	1101011101101000110001111000011001100001
7	0110	0101110011101010011010000100100101110101
8	0111	0010011111110100101101100110000100110100
9	1000	0101001101110100011100000101011101011100
10	1001	1100110011010010110100110011000110011100
11	1010	1011000111011100101010101001100010001110
12	1011	1110101111001001100111011100111010011001
13	1100	0001001011001011010011101111010011100001
14	1101	1000110101100111000000100111010000111011 1
15	1110	0110011001100101101010110010001101100011
16	1111	0011110001111010011101010000111100100010

Appendix B Synchronization Between Sender and Receiver & CFO Correction

In this section, we build a mathematical model for the synchronization and carrier frequency offset (CFO) correction (i.e., part (*i*) in Fig. 11). Once the WiFi transmission is identified, the WiFi receiver first leverages the preamble to synchronize the symbol timing and then correct the CFO. Therefore, we start with an analysis of how the tag performs the synchronization between the WiFi sender and the WiFi receiver.

Fig. 29 shows our synchronization solution. The preamble includes two long training (LT) sample sequences. Each sequence is composed of 64 samples. By identifying where the two peak values of the correlation between the received samples and training pattern occur, the WiFi receiver can find where the symbol starts [15]. Therefore, to avoid changing the training sample's value, when TScatter detects the WiFi transmission by using the RF energy detector, it first uses continuous square waves to only reflect the preamble and does not embed any data into the preamble. After the square waves, TScatter starts to modulate the WiFi samples.

One may ask whether the phase offset (described in Sec. 6.1) will change the preamble value, therefore affects the synchronization. We note that TScatter does not change the preamble sample's value. Without loss of generality, let us assume the phases of continuous square waves are $\theta_n = 0$ and the training pattern value is $\overline{LT}_i$. Then, the positions of two peak values of the correlation between the received samples $\mathcal{R}_l$ (described in Eqn. 8) and the training pattern $\overline{LT}_i$ are:

$$\begin{aligned}(l_1, l_2) =& \underset{l}{\text{argmax}_2} \Big|\Big| \sum_{i=0}^{63} (\mathcal{R}_{l+i} * \overline{LT}_i) \Big|\Big|_2 \\ =& \underset{l}{\text{argmax}_2} \Big|\Big| e^{j[\beta+\phi]} e^{j*0} \sum_{i=0}^{63} (\mathcal{S}_{l+i} * \overline{LT}_i) \Big|\Big|_2 \\ =& \underset{l}{\text{argmax}_2} \Big|\Big| \sum_{i=0}^{63} (\mathcal{S}_{l+i} * \overline{LT}_i) \Big|\Big|_2 \end{aligned} \tag{19}$$

Where, argmax$_2$ provides the two sample numbers (l_1, l_2) for two peak correlation values.

Eqn. 19 proves that the positions of two peak correlation values are only related to the original sample values $\mathcal{S}_l$ and are not affected by the phase offset. Therefore, the phase offset will not affect the synchronization. Our experimental result (shown in Fig. 30) also illustrates that when the tag uses continuous square waves to reflect the WiFi signals, it does not interfere with the positions of two peak correlation values, and the receiver can achieve a precise synchronization.

In the WiFi receiver, the received sample at frequency f_c is down-converted to baseband with a local carrier frequency $(1+\varepsilon)f_c$. At the receiver side, the frequency offset εf_c causes two phenomenas: carrier frequency offset (CFO) and sampling frequency offset (SFO). The CFO introduces an extra phase offset of $\alpha = 2\pi\varepsilon\frac{f_c}{f_s}$ on the samples during each down-conversion procedure. When the l'th sample of the frame is down-converted, the phase offset applied to the sample is accumulated to $l\alpha$.

$$\mathcal{R}_{l,n} = e^{j(\beta+\phi)} e^{j(l\alpha)} e^{j\theta_n} \mathcal{S}_n \tag{20}$$

Where l and n point to the same sample. Since a WiFi frame contains multiple symbols, l is the sample number in the frame, while n is the sample number in the symbol.

Since the tag does not affect the synchronization, the CFO can be estimated accurately. An estimation of the CFO can be obtained simultaneously when the synchronization is achieved. According to the WiFi specification [2], the WiFi receiver uses LT samples to estimate the CFO. The starting LT sample number is computed as l_1. Let us denote the t'th LT sample as $\mathcal{R}_{l_1+t}$. The CFO estimator is given by:

$$\alpha = \frac{1}{64}\angle\left(\sum_{t=0}^{63} \mathcal{R}^*_{l_1+t}\, \mathcal{R}_{l_1+t+64}\right) \tag{21}$$

To correct the CFO, the received sample $\mathcal{R}_{l,n}$ is multiplied by $e^{-jl\alpha}$. Through the synchronization, the WiFi receiver learns the sample number l. Thus, the received sample can be updated to the corrected sample S_n:

$$\mathcal{R}_n = \mathcal{R}_{l,n} * e^{-jl\alpha} = e^{j(\beta+\phi)} e^{j\theta_n} x_n \cancel{e^{jl\alpha}} \cancel{e^{-jl\alpha}} \tag{22}$$

Eqn. 22 shows that the impact of CFO is canceled by the CFO correction module in the WiFi receiver.

Appendix C SFO Correction

Like the CFO, an SFO also exists, which causes a phase offset to OFDM subcarriers after the DFT [28]:

$$\alpha_{sfo} = -2\pi\varepsilon rk\frac{N+N_g}{N} \tag{23}$$

Where, $N = 64$ is the DFT size, $N_g = 16$ is the cyclic prefix length, r is the symbol number which is learned through the synchronization, k is the subcarrier number and ε can be calculated from the estimated CFO ($\varepsilon = \frac{\alpha f_s}{2\pi f_c}$, described in Sec. 6.1). The OFDM subcarriers are corrupted as $Y_k e^{j\alpha_{sfo}}$.

From Eqn. 23, we know the key to correct the SFO is to estimate CFO accurately. In Appendix B, we have demonstrated the tag does not change the preamble value so that the receiver can achieve a precise CFO estimation. The output subcarriers are multiplied by $e^{-j\alpha_{sfo}}$ (i.e., $Y_k = Y_k \cancel{e^{j\alpha_{sfo}}} \cancel{e^{-j\alpha_{sfo}}}$) to correct the offset.

Simplifying Backscatter Deployment: Full-Duplex LoRa Backscatter

Mohamad Katanbaf[1,2*], Anthony Weinand[1], and Vamsi Talla[1]

[1]Jeeva Wireless
[2]University of Washington

Abstract

Due to the practical challenges in the deployment of existing half-duplex systems, the promise of ubiquitous backscatter connectivity has eluded us. To address this, we design the first long-range full-duplex LoRa backscatter reader. We leverage existing LoRa chipsets as transceivers and use a microcontroller in combination with inexpensive passive elements including a hybrid coupler, inductors, tunable capacitors, and resistors to achieve 78 dB of self-interference cancellation and build a low-cost, long-range, and small-form-factor Full-Duplex LoRa Backscatter reader.

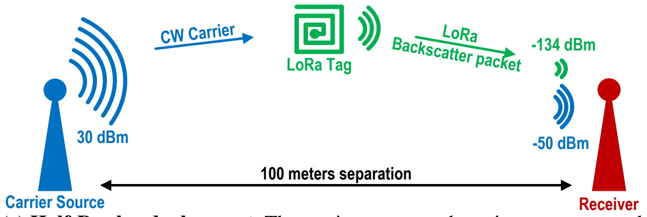
We evaluate our system in various deployments and show that we can successfully communicate with a backscatter tag at distances of up to 300 ft in line of sight, and through obstacles, such as walls and cubicles, in a 4,000 ft^2 office area. We reconfigure our reader to conform to the size and power requirements of a smartphone, and demonstrate communication with a contact-lens-form-factor prototype device. Finally, we attach our reader to a drone and demonstrate backscatter sensing for precision agriculture with an instantaneous coverage of 7,850 ft^2.

1 Introduction

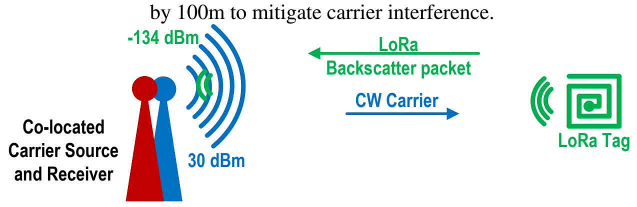
Recent advances [47, 56, 66, 73, 84, 88] have demonstrated the potential of backscatter to replace power-hungry, large, expensive radios [1] with an orders of magnitude lower power, smaller-size, cheaper, potentially battery-free connectivity solution. This promise, however, has run into practical limitations in regard to existing backscatter infrastructure. Full-duplex (FD) RFID readers [18, 22] and other proprietary full-duplex systems [30, 35], in which a single reader communicates with tags are easy to deploy, but these existing readers are large, complex, expensive, and have limited range.

To address this, recent half-duplex (HD) backscatter systems have leveraged the economies of scales and ubiquity of industry-standard protocols such as WiFi [56, 88], Bluetooth [43, 53, 89], ZigBee [64, 89], and LoRa [73, 84] to reduce the cost, size, and complexity of reader infrastructure and achieve longer range. However, these systems suffer from deployment issues, as the half-duplex topology requires two physically-separated radio devices: one for transmitting the carrier, and another for receiving the backscattered data packet. The need to deploy multiple devices in different locations significantly limits the use cases for backscatter.

We present the first Full-Duplex LoRa Backscatter reader which combines the low-cost, long-range, and small-form-factor benefits of a standard-protocols-compliant backscatter system with the simple deployment of a full-duplex system. This addresses one of the remaining pain points of backscatter and opens backscatter to a plethora of applications. A low-cost, long-range, small-form-factor full-duplex reader could be easily integrated into existing devices. This would enable peripheral, wearable, and medical devices such as pill bottles [33, 70], insulin pens [45, 60], smart glasses [59, 75], and contact lenses [36, 48] to use backscatter to directly communicate with a cellphone, tablet, or laptop. Similarly, in agriculture, aerial surveillance drones could be equipped with



(a) **Half-Duplex deployment.** The carrier source and receiver are separated by 100m to mitigate carrier interference.



(b) **Full-Duplex deployment.** The carrier source and receiver are co-located and need 78 dB interference cancellation.

Figure 1: Overview of HD and FD Backscatter Deployments.

*Work was done while the author was at Jeeva Wireless.
[1]active RFIDs are also radios

a backscatter reader to collect data from sensors distributed throughout a field.

To understand the challenge in building an FD LoRa Backscatter reader, consider the traditional HD LoRa backscatter system, shown in Fig. 1(a). The first radio transmits a single-tone carrier at power levels up to 30 dBm. A tag uses subcarrier modulation to backscatter a packet at an offset frequency, which is then received by the second radio, 100 m from the transmitter. The physical separation is necessary to attenuate the out-of-band carrier at the receiver to a level where it does not impact the sensitivity [84]. This illustrates the fundamental challenge in a FD LoRa Backscatter system, as shown in Fig. 1(b): The single-tone carrier needs at least 78 dB of suppression, a 63-million $\times$ reduction in signal strength, between the transmitter and a commodity LoRa receiver chipset, both integrated on the same PCB. This suppression must be implemented in the analog RF domain without substantially increasing the cost, complexity, or power consumption of the system. Furthermore, unlike path-loss attenuation, which is wide-band, typical cancellation techniques have a trade-off between cancellation depth and bandwidth [31, 61, 65]. If the cancellation bandwidth is insufficient, the carrier phase noise will show up as in-band noise at the receiver. Therefore, a second requirement is to bring the phase noise of the carrier at the offset frequency to below the receiver's input noise level.

Existing FD cancellation techniques, including analog, digital, and hybrid designs used in full-duplex radios have different or more relaxed requirements and, as a consequence, are insufficient to meet the needs of our system. Analog and hybrid cancellation techniques require bulky and expensive RF components such as circulators [9], vector modulators [21], and phase shifters [23] to achieve sufficient cancellation, each of which increases cost and size. Digital cancellation techniques require access to IQ samples, which are unavailable on commodity radios, and instead use SDRs [28, 31, 35, 83], FPGAs [85], or DSPs [29], which are all prohibitively expensive. For a detailed analysis of why existing techniques are insufficient, see §8.

Our key idea is to leverage the ubiquity and economies of scale of existing LoRa transceivers and microcontrollers and add inexpensive passive components to fulfill the two requirements of full-duplex operation. This enables us to build a *low-cost, long-range, small-form-factor*, Full-Duplex LoRa Backscatter reader. We use a single-antenna topology with a hybrid coupler to interface the transmitter and the receiver with the antenna. The leakage from the transmitter to the receiver, i.e. self-interference, is a function of the impedance at the coupled port. A microcontroller adaptively tunes an impedance network, tracking variations in the antenna impedance and environmental reflections with the objective of minimizing interference at the receiver.

We introduce a *novel, two-stage, tunable impedance network* to achieve 78 dB suppression of carrier signal. The extent to which a carrier is suppressed is a function of how closely the tunable impedance can track the variations in antenna impedance, which in turn depends on the resolution of the impedance network. A single-stage network is limited by the step size of its digital capacitors and does not have a high enough resolution to reliably achieve 78 dB cancellation [38, 50, 54, 65]. Our two-stage network is built by cascading two stages, each consisting of four, 5-bit tunable capacitors and two fixed inductors with an attenuating resistor network between the stages. The two-stage design provides the fine-grain control and coverage necessary to meet the 78 dB carrier cancellation target across the expected range of variation in antenna impedance. To achieve the second objective of bringing the phase noise of the carrier at the offset frequency below the noise floor of the receiver, while simultaneously obtaining 78 dB cancellation at the carrier frequency, is very challenging. There is a fundamental trade-off between the cancellation depth and bandwidth [31, 61, 65], and we prioritize the 78 dB cancellation requirement at the carrier frequency. We use a COTS synthesizer with low phase noise to relax the cancellation requirement at the offset frequency.

We implement the Full-Duplex LoRa Backscatter system using only COTS components, for a total cost of \$27.54 at low volumes, only 10% more than the cost of two HD units. Our evaluation shows that the two-stage impedance network achieves >78 dB carrier cancellation and >46 dB of offset cancellation in practical scenarios with a tuning time overhead of less than 2.7%. Results are summarized below:

- The FD reader can communicate with tags at distances of up to 300 ft in line of sight. When placed in the corner of a 4,000 ft^2 office space with concrete, glass, and wood structures and walls, tags can operate within the entire space.
- We integrate a low-power configuration of the FD reader into portable devices. We attach the prototype to the back of a smartphone and show that the tags can communicate at distances beyond 50 ft at 20 dBm, 25 ft at 10 dBm, and up to 20 ft at 4 dBm.
- We build two proof-of-concept applications. We prototype a contact-lens-form-factor antenna tag and show that it can communicate with FD reader attached to a smartphone at distances of up to 22 ft and when the reader is inside a user's pocket. We also attach the FD reader to a quadcopter and fly it to 60 ft above a field. The reader is able to communicate with tags placed on the ground at a lateral distance of up to 50 ft, corresponding to an instantaneous coverage of 7,850 ft^2.

2 Background

Our work brings full-duplex operation to a LoRa Backscatter system. We start with a background on LoRa backscatter, followed by a primer on the full-duplex operation.

2.1 LoRa Backscatter Primer

Backscatter communication eliminates RF carrier generation at the tag and, instead, uses switches to reflect existing, ambient RF signals for data transmission. This drastically reduces the cost, size, and power consumption of wireless communication [30,56,84]. In HD backscatter deployment, as shown in Fig. 1(a), a radio source generates the single-tone carrier, a backscatter tag reflects the carrier to synthesize LoRa packets, then a receiver decodes the backscattered packets. However, the carrier signal also ends up as a strong source of interference at the receiver, which degrades the receiver's ability to decode packets. Backscatter systems use two key techniques to mitigate carrier interference [26,43,46,47,49,56,64,73,78,81,84,89,92,93]. First, the tag uses subcarrier modulation to synthesize packets at a frequency offset from the carrier frequency. The receiver operates at the offset frequency, pushing the interference, i.e. the carrier signal, out of band at the receiver. Since receivers are designed to operate in the presence of out-of-band interference, the receiver can decode the backscattered packets with minimal loss in sensitivity. Second, the transmitter and receiver are physically separated to attenuate that carrier to a level where it does not affect the receiver's sensitivity. However, in full-duplex systems, by definition, the transmitter and receiver cannot be physically separated.

LoRa receivers have low sensitivity and high blocker tolerance, making them ideal candidates for long-range backscatter connectivity, as demonstrated by prior HD backscatter designs [73,84]. LoRa has two key protocol parameters that can be used to trade off data rate with receive sensitivity: spreading factor and bandwidth. Since our system transmits up to 30 dBm, the FCC mandates frequency hoppingand a maximum channel dwell time of 400 ms [17]. So, we limit our system to protocols with packet lengths shorter than this limit, which corresponds to a sensitivity of -134dBm. Longer packet lengths are incompatible with frequency hopping unless we implement intra-packet frequency hopping. Doing so would require tuning in the middle of packet reception, which is not feasible on commodity radio receivers.

2.2 Full-Duplex Primer

FD radios transmit and receive at the same time on a single device, allowing simultaneous communication between devices without delay. The main obstacle in achieving FD functionality is self-interference; the strong transmit signal leaks to the sensitive receiver and appears as interference, degrading its performance. Hence, the key is to suppress the interference before it reaches the receiver. Broadly speaking, there are two approaches to FD operation: out-of-band and in-band.

Cellular standards such as WCDMA and LTE implement out-of-band full-duplexing by using Frequency Division Duplexing (FDD), where two distinct fixed frequency bands are used for uplink and downlink. In FDD systems, the operating frequency and frequency offset are fixed, and a frequency selective duplexer is used to suppress the transmitter leakage in the receive band. At first glance, it may look like FD backscatter is similar, as the tag backscatters the data packet at a frequency offset, however, FDD systems use much higher offset frequencies, at least 40 MHz for WCDMA and LTE bands above 800 MHz [20], in line with the requirements of practical frequency duplexers and passive filters. Backscatter systems, in contrast, transmit the carrier and receive the packet within the same frequency band, keeping the frequency offset low to minimize the power consumption of the tag. For example, the LoRa backscatter system operating in the 902-928MHz ISM band uses an offset frequency of 2-4 MHz. With such a small offset, we cannot leverage passive filters or frequency duplexers used in FDD systems.

In recent years, researchers have demonstrated success with In-Band Full-Duplex (IBFD) radios [31,35,41,61], where radios transmit and receive simultaneously on the same frequency. IBFD radios suppress self-interference by using a combination of analog and digital cancellation techniques to bring the signal strength of the typically-wideband carrier below the noise floor of the receiver over the entire receive bandwidth. IBFD radios use isolation and analog cancellation techniques in the RF domain to first bring the carrier signal below the saturation level of the receiver front end. Next, digital cancellation techniques are used in the baseband to suppress the signal below the noise floor across the receiver bandwidth. Since the frequency offset in a backscatter system is small, an FD backscatter system can leverage SI suppression techniques similar to IBFD devices, such as isolation and analog cancellation. However, there are two key differences. First, the FD LoRa Backscatter system uses a single-tone signal as the carrier, so we need to suppress a very narrow-band signal. Second, unlike IBFD radios, the FD LoRa Backscatter system uses existing COTS radios, which do not provide access to signals in the digital baseband of the receiver. Hence, unlike IBFD systems, which use digital cancellation in addition to analog cancellation, we need to achieve 78 dB of SI cancellation entirely within the analog domain.

3 FD LoRa Backscatter Requirements

In this work, we focus on the cancellation requirements for a LoRa backscatter system, but the design techniques and architecture are not LoRa specific. They can be extended to build FD backscatter systems for other wireless standards such as WiFi, Zigbee, Bluetooth, SigFox, or NB-IoT that use a single-tone carrier and subcarrier modulation to synthesize backscatter packets [43,47,56]. However, these techniques are not directly applicable to systems which do not use sub-carrier modulation [30] or use wide-band Wi-Fi or LoRa packets as carrier [64,73,89].

We divide the cancellation requirements into two categories: carrier cancellation and offset cancellation.

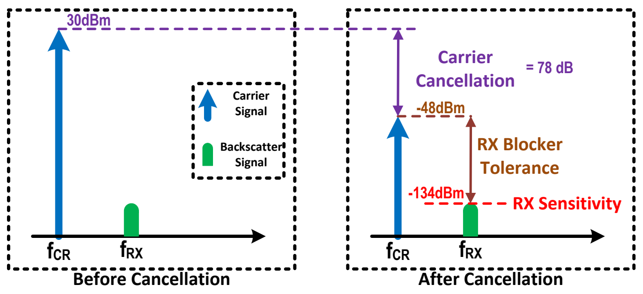


Figure 2: **Carrier Cancellation.** The 30 dBm single-tone carrier needs 78 dB of attenuation to meet the Rx blocker tolerance and ensure packet reception down to Rx sensitivity.

3.1 Carrier Cancellation

We define carrier cancellation (CAN_{CR}) as the required cancellation of the carrier signal (the source of self-interference) at the center frequency. The carrier acts as a *blocker* i.e. a strong signal in the vicinity of the desired signal that can affect a receiver's performance and reduce its sensitivity. A strong blocker can saturate the LNA, forcing it to reduce gain and increase the noise floor. Secondly, post LNA, a blocker can mix with the receiver local-oscillator phase-noise and contribute to in-band noise. Finally, baseband filters have limited stopband attenuation, and even a small portion of the blocker passing through the filter reduces the signal-to-noise and -interference ratio.

We compute the minimum required carrier cancellation as

$$CAN_{CR} > P_{CR} - Rx_{Sen} - Rx_{BT} \tag{1}$$

where P_{CR} is the carrier power, Rx_{Sen} is the receiver sensitivity, and Rx_{BT} is the receiver blocker tolerance.

For example, as per the SX1276 datasheet, the blocker tolerance at a 2 MHz offset for $BW = 125\ kHz$, $SF = 12$, -137 dBm sensitivity protocol is 94 dB [19]. Based on equation 1, for $P_{CR} = 30$ dBm, at least 73 dB of SI cancellation is required. However, the datasheet blocker specification assumes a 3 dB degradation in sensitivity, which is substantial for backscatter systems. Additionally, the datasheet provides blocker specifications for only a subset of frequency offsets and protocol parameters. To get a more comprehensive set of requirements, we perform our own blocker experiments for a range of frequency offsets (2, 3, and 4 MHz) and protocol parameters (366 bps - 13.6 kbps data rates). We connect a LoRa transmitter and a single-tone generator to two variable attenuators and combine their outputs at a LoRa receiver. First, we get a baseline sensitivity by turning off the single tone and increase the LoRa transmitter attenuation till we reach receiver sensitivity, defined by PER < 10%. Next, we turn on the single-tone generator with maximum attenuation and reduce the attenuation, thereby increasing blocker power, until the PER exceeds 10%. We record the maximum tolerable interference power for different frequency offsets, receiver bandwidths, and spreading factors to achieve different data rates and conclude that 78 dB is the most stringent carrier-cancellation specification. As mentioned prior, the blocker performance of a receiver depends on both the RF front end and digital baseband. Our analysis shows that the SX1276 baseband has sufficient digital baseband filtering to suppress the blocker at the offset frequency, and additional filtering would not help in this specific case.

Fig. 2 illustrates the carrier cancellation requirement. Before cancellation, the carrier signal is much stronger, but after cancellation, the SI is dropped to a level that the receiver can tolerate. The difference between these levels is CAN_{CR}. Note that a lower cancellation may suffice for some data rates and frequency offsets, but our design supports all configurations.

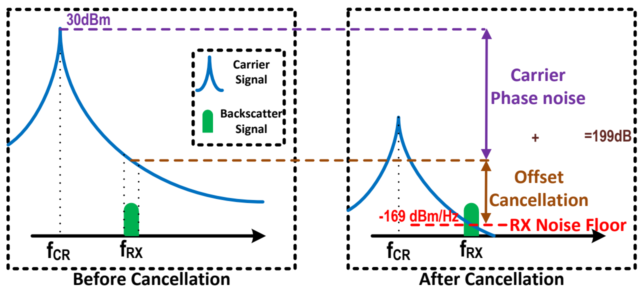


Figure 3: **Offset Cancellation**. The phase noise of the single-tone carrier at the frequency offset after cancellation should be less than the noise floor of the receiver.

3.2 Offset Cancellation

We define offset cancellation (CAN_{OFS}) as cancellation of the carrier signal at the offset frequency. We use a single-tone signal as the carrier. An ideal oscillator generates a pure sine wave, with the entire signal power concentrated at a single frequency. However, practical oscillators have phase-modulated noise components, which spreads the power to adjacent frequencies. This results in noise side bands [76] and is characterized by the source's phase noise, i.e. power spectral density (dBc/Hz) of the noise at a frequency offset from the center frequency. Since the receiver operates at a frequency offset from the carrier, the carrier phase noise shows up as in-band noise to the receiver. For optimal receiver performance, the SI signal after cancellation at the offset frequency should be less than the receiver noise floor. We compute the minimum required offset cancellation as

$$P_{CR} + \mathcal{L}_{CR(\Delta f)} + 10Log(B) - CAN_{OFS} < 10Log(KTB) + Rx_{NF}$$

$$CAN_{OFS} - \mathcal{L}_{CR(\Delta f)} > P_{CR} - 10Log(KT) - Rx_{NF} \tag{2}$$

where $\mathcal{L}_{CR(\Delta f)}$ is the carrier phase noise at the offset frequency (Δf), B is the receiver bandwidth, K is the Boltzmann constant, T is temperature, and Rx_{NF} is receiver noise figure. We show this requirement in Fig. 3. Before cancellation, the backscattered signal is buried under the carrier phase noise, but, after cancellation, the carrier phase noise is pushed below the receiver noise floor.

As per the SX1276 datasheet $Rx_{NF} = 4.5$ dB, so for $P_{CR} = 30$ dBm, $CAN_{OFS} - \mathcal{L}_{CR(\Delta f)} > 199.5$ dB. The offset cancellation depends on transmit power, carrier phase noise, and

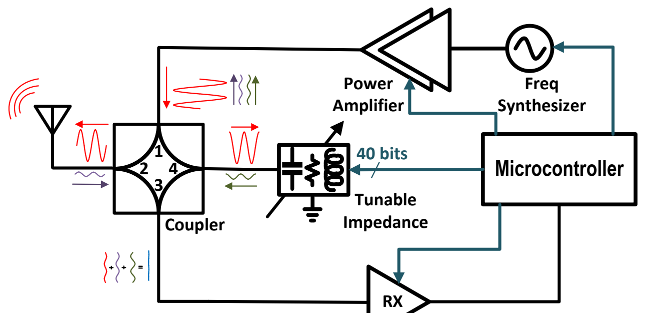


Figure 4: **System Architecture.** We use a single-antenna hybrid-coupler architecture with a two-stage tunable impedance network for cancellation. A microcontroller controls all components and implements the tuning algorithm.

receiver noise figure. Interestingly, since the channel bandwidth appears on both sides of equation 2, offset cancellation is independent of the receiver channel bandwidth. In other words, we can use the carrier phase noise per unit bandwidth and receiver noise floor per unit bandwidth instead of the aggregate noise over the entire bandwidth.

In an HD system, the transmitter and receiver are physically separated, and the carrier attenuation via RF propagation does not significantly change with frequency. So, if the 78 dB carrier cancellation requirement is met, the phase noise at the offset frequency would also be attenuated by the same amount. However, cancellation networks do not have the same wide-band frequency characteristics [38, 40, 54, 65, 77]. The inequality shows that satisfying the offset cancellation requirement for an FD system requires a joint design of the carrier source and the cancellation network. If we use a high-phase-noise carrier, we would need high offset cancellation, whereas if we lower the phase noise of the carrier source, we can relax the offset cancellation requirements. Carrier and offset SI cancellation are functions of offset frequency, and both typically relax with an increase in offset frequency. However, an increase in offset frequency increases the tag power consumption. Thus, the frequency offset presents a trade-off between tag power consumption and SI cancellation requirements. 2-4 MHz is a reasonable compromise; we use a 3 MHz offset frequency in this work.

4 FD LoRa Backscatter Reader Design

The FD LoRa Backscatter system uses a single antenna and a hybrid coupler with a two-stage tunable impedance network to achieve carrier and offset cancellation. The cancellation network uses only passive components, minimizing cost, complexity, power consumption, and noise. Fig. 4 shows the block diagram of our design. The antenna is connected to the transmit and receive paths via a hybrid coupler. The fourth port of the coupler is connected to a tunable impedance network that tracks the antenna impedance to reflect and phase shift a portion of the single-tone carrier, suppressing the SI that flows to the receiver. The carrier signal is generated by a frequency synthesizer and fed to a power amplifier (PA) to transmit up to 30 dBm. An on-board microcontroller controls the synthesizer, PA, receiver radio, and digital tunable impedance network using a SPI interface. We use RSSI readings from the receiver to measure the SI there.

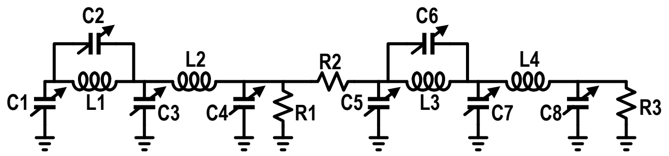
Below, we describe the principle of operation for the hybrid coupler. This is followed by the two-stage tunable impedance network design and the choice of carrier source to meet the carrier and offset cancellation requirements. Finally, we describe the tuning algorithm.

4.1 Coupler: Principle of Operation

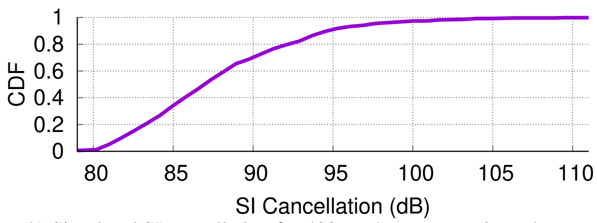
Couplers are four-port devices that divide an incident signal between the output and coupled port while keeping the fourth port isolated [72, 74]; we use this property to isolate the transmitter and receiver. We connect the transmitter to the input port (1), the receiver to the isolated port (3), the antenna to the output port (2), and the tunable impedance network to the coupled port (4). The carrier signal is split between the antenna and coupled ports, leaving the receiver isolated. The received signal at the antenna port is split between the receiver and the transmitter, leaving the tunable impedance isolated. Couplers are reciprocal elements, meaning that the signal split described above is symmetrical. Ideally, we would want the entire PA output to go to the antenna (low TX insertion loss) and the entire received signal from the antenna to go to the receiver (low RX insertion loss). A higher coupling factor would direct more of the PA output to the antenna at the cost of reducing signal transmission from the antenna to the receiver. Since our goal is to maximize the communication range, we must minimize the sum of the transmit and receive insertion losses. A hybrid, or 3 dB coupler, equally divides the input power between the output and coupled ports and minimizes total loss to 6 dB.

Two factors limit the practical SI cancellation of a hybrid coupler. First, every coupler has leakage: a typical COTS coupler provides $\sim$ 25 dB of isolation between the transmit and receive ports [16], far lower than our requirement. Second, and more important, is the effect of the antenna. Typical antennas, including low-cost PIFAs, are characterized by -10 dB return loss, and any reflection from the antenna port would couple to the receiver and further contribute to the SI.

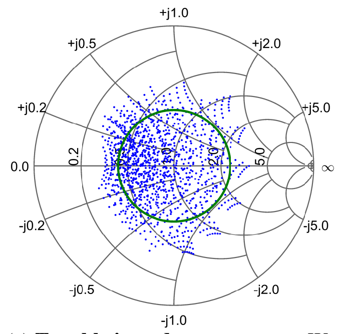
Antenna Impedance Variation. Environmental variations affect antenna impedance, i.e. nearby objects can detune the antenna or create additional reflections that contribute to variation in its reflection coefficient. Since SI cancellation is dependent on antenna impedance, it is essential to characterize the expected variation. We design a 1.9 in $\times$ 0.8 in PIFA for our implementation and use this antenna to quantify impedance variation. We connect the PIFA to an Agilent 8753ES VNA [3] and subject it to environmental variations



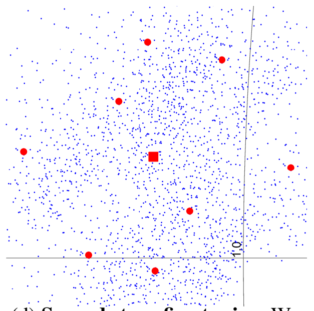
(a) Topology of the two stage impedance network.



(b) Simulated SI cancellation for 400 random antenna impedance



(c) **Tunable impedance coverage.** We show the coverage of the first stage on smith chart.
(step size = 6 LSBs)



(d) **Second stage fine tuning.** We show impedance states with finer resolution achieved using second stage(blue dots)
(step size = 10 LSBs)

Figure 5: Two stage tunable impedance network used for SI cancellation and its simulation results.

by placing the antenna upright on a table, and measure S_{11} as a hand and other objects approach it from different directions. Our results show that the magnitude of the reflection coefficient, $|\Gamma|$, reaches a maximum of 0.38, and we set expected $|\Gamma| < 0.4$ for the antenna.

4.2 Two-Stage Tunable Impedance Network

We use a tunable impedance at the coupled port [50, 54, 65] to negate SI leakage due to variation in antenna impedance. However, the cancellation depth is a function of how precisely we can tune this impedance. To achieve 78 dB carrier cancellation, we need a very fine resolution for the tunable impedance, which is not available in COTS digital capacitors. To solve this, we introduce a novel two-stage tunable impedance network that allows us to achieve deep cancellation, using only passive components.

To understand how the tunable impedance network improves the SI suppression from the coupler, follow the flow of signals in Fig. 4. The carrier (red) splits between the antenna port and the coupled port. The impedance at the coupled port is tuned such that the reflection from this port (green) cancels out both the leakage of the coupler and the reflection from the antenna port (purple) to achieve deep cancellation at the receive port (blue). In the worst case of a significantly detuned antenna, reflection from the antenna is much stronger than the leakage, and this should be canceled by the reflection from the tunable impedance.

We use a topology of two fixed inductors and four digitally tunable capacitors terminated with a resistor to cover the range of expected impedance values required to cancel strong reflections from the antenna [65]. We observe that in a tuning network terminated with a resistor, only a small portion of the signal is reflected, and most of it is dissipated. We replace the termination resistor with a resistive signal divider, followed by a second tunable impedance, as shown in Fig. 5(a). The signal reflected by the second stage flows through the resistive divider twice, effectively lowering the impact of impedance changes in the second stage on the overall reflected signal. This allows us to use the second stage to make much more fine changes in impedance, increasing the tuning resolution and enabling deep cancellation. However, the first stage still determines the tuning range and offset cancellation. The second stage provides the fine resolution to accurately match the reflection from the balanced port with the leakage and reflection from the antenna port. To eliminate dead zones, we choose the resistive divider such that the fine tuning network covers the step size of the coarse tuning network. This coarse and fine tuning approach is similar to using a hybrid coupler combined with an analog feed-forward path. The second stage effectively provides the functionality of a feed-forward path, but without additional noise and at a lower cost, lower complexity, and lower power.

We simulate the behavior of our tunable impedance network to demonstrate the efficacy of our approach. We plot the tuning network reflection coefficient at 915 MHz in Fig. 5 (c) for a step size of six LSBs for each capacitor in the first stage. For visibility, the plot only shows 1,296 impedance states out of more than 1 million first-stage impedance states and more than 1 trillion total states. The plot demonstrates that our design can cover the impedances corresponding to the antenna reflection coefficient circle of $|\Gamma| < 0.4$. Next, we show the fine tuning of the second stage in Fig. 5 (d). We select an initial state (the large, red square in the center) and change each capacitor in the first stage by one LSB to get the other eight red dots. Without the second stage, we would not be able to achieve any impedance between these dots. For each of these nine states, we use a step size of 10 LSBs for each capacitor in the second stage and show the resulting impedances in blue. The blue cloud shows the fine resolution control covering the dead zone between the first-stage steps. Finally, we simulate SI cancellation for 400 random points of antenna impedance inside the $|\Gamma| < 0.4$ circle and plot the cancellation CDF in Fig. 5 (b). Cancellation of > 80 dB is achieved for the 1st percentile, which satisfies our requirement.

4.3 Carrier Source Selection

The phase noise of the carrier source determines the required cancellation at the offset frequency, as shown in equation 2. In order to understand the requirements of the carrier source, we first simulate SI cancellation at different frequency offsets.

We tune the capacitors to achieve a minimum of 78 dB carrier cancellation and then sweep the operating frequency. Our results show that at 3 MHz frequency offset, 47 dB cancellation or more is achieved in 90% of the simulated cases. If we use SX1276 with a phase noise of -130 dBc at 3 MHz offset as our transmitter, then 47 dB of offset cancellation is insufficient.

The tuning network has multiple poles that can be optimized to increase cancellation bandwidth [57, 65]. However, doing so reduces the cancellation at the center frequency. This approach would also require a wide-band receiver to provide feedback on the SI power over the entire bandwidth to tune the capacitors, which is unavailable on the SX1276 (max. BW = 500 kHz). We need > 78 dB of SI cancellation at the carrier frequency and prioritize this requirement. Instead of SX1276, we use ADF4351 [10] frequency synthesizer to generate the single tone carrier, which has a lower phase noise of -153 dBc at 3 MHz offset. Although the ADF4351 is slightly more expensive, it relaxes the offset cancellation requirement to 46.5 dB and eliminates the need for an additional wide-band receiver or a power detector, justifying the design choice.

4.4 Tuning Algorithm

Our design uses a two-stage impedance network with eight digital capacitors, each with five control bits; a total of 40 bits resulting in 2^{40} ($\sim$ 1-trillion) states for the impedance network. Multiple capacitor states can result in the impedance required for 78 dB cancellation, and any one of those is acceptable. As it is impossible to search across such a vast space in real-time, there is a need for an efficient tuning algorithm that can run on a commodity ARM Cortex-M4 microcontroller.

We use a simulated annealing algorithm to tune the capacitors in each stage separately [86]. Simulated annealing is based on the physical process of heating, and then slowly cooling, a material to minimize defects in its structure. For every temperature value, we take ten steps. At each step, we add a random value bounded by a maximum step size to each capacitor and measure the SI using receiver RSSI measurement. We accept the new capacitor values if the SI decreases, or with a temperature-dependent probability when the SI increases. We start with a temperature equal to 512 and divide it by two each round until it reaches one. We set predefined cancellation thresholds for each stage and stop the tuning once the thresholds are met. If the first stage reaches the threshold (set to 50 dB), but the second stage fails to do so, we repeat the tuning until either it converges or reaches a timeout.

5 Implementation

We implement the FD LoRa Backscatter reader for operation in 902-928 MHz on a 3.8 in $\times$ 1.9 in, 4-layer, FR4 PCB. We place the RF components, including antenna, transmitter, receiver, and cancellation network, on the top side of the PCB, and microcontroller and power management on the bottom. We use the SX1276 as the LoRa receiver [19]. The cancellation circuit consists of the X3C09P1 90° hybrid coupler [16] and a two-stage tunable impedance network,shown in Fig. 5 (a). Variable capacitors C_1-C_8 are implemented by pSemi PE64906 tunable capacitors, with 32 linear steps from 0.9 pF - 4.6 pF [11]. We set inductors L_1, L_3 to 3.9 nH and L_2, L_4 to 3.6 nH. We set resistors R_1, R_2, and R_3 to 62 Ω, 240 Ω, and 50 Ω respectively. We use the ADF4351 synthesizer to generate the single-tone carrier, which has 23 dB better phase noise at 3 MHz offset compared to the SX1276. The output power of the carrier can be amplified up to 30 dBm using the SKY65313-21 power amplifier [12]. Our cancellation technique has an expected loss of 7-8 dB; 6 dB of which is the theoretical loss due to hybrid coupler architecture, the rest is due to component non-idealities.

We design a custom coplanar inverted-F PCB antenna. The radiating element of the antenna measures 1.9 in $\times$ 0.8 in and leverages the existing ground plane for omnidirectional radiation. We measure the performance of the antenna in an anechoic chamber, and results show a peak gain of 1.2 dB, half-power beam-width of 80°, and cumulative efficiency of 78%. The transmitter, receiver, and cancellation network are controlled using a SPI interface by an on-board ARM Cortex-M4 STM32F4 microcontroller [13]. The microcontroller implements a state machine in C to transition between tuning, downlink, and uplink operating modes. In the tuning mode, the microcontroller first configures the center frequency and power of the carrier and then tunes the impedance network to minimize SI using the simulated annealing algorithm described in §4.4. After the tuning phase, the MCU sends the downlink OOK message to wake up the backscatter tag. Then, it transitions to the uplink mode where it configures the receiver with the appropriate LoRa protocol parameters to decode backscattered packets. The MCU then repeats this cycle for the next frequency.

5.1 FD Reader Configurations

We configure the FD LoRa Backscatter reader for two different use cases; a 'base-station' setup and a 'mobile' setup. Below we describe each configuration.

Base-Station Configuration: The base-station configuration of the FD LoRa Backscatter reader uses a 8 dBc high gain patch antenna [25]. The synthesizer and PA are set to transmit at 30 dBm. These settings maximize operating range and we use this configuration for the line-of-sight and non-line-of-sight range testing. In the base-station setup method, the power amplifier, synthesizer, receiver, and MCU consume 2,580 mW, 380 mW, 40 mW, and 40 mW, respectively, resulting in total power consumption of 3.04 W. 3.04 W is not a limitation for a plugged-in device such as a smart speaker or WiFi router, but is too high for a portable device.

Mobile Configuration: For applications with lower power consumption and smaller size requirements, we configured the system as a ‘mobile’ version. We use the on-board antenna and configure the synthesizer and PA to transmit at lower power levels of 4 dBm, 10 dBm, and 20 dBm. Since the PA and synthesizer dominate power consumption, reducing transmit power greatly reduces power consumption. In this mobile configuration, power consumption is low enough to draw from conventional portable power sources like a USB battery or a laptop. It is also small enough that, if desired, we are able to attach it to an iPhone 11 Pro without increasing the phone’s footprint, shown in Fig. 11(a).

Lower transmit powers relax cancellation requirements (see §3.2), which can be leveraged to further reduce the power consumption of the synthesizer and the power amplifier. For 20 dBm output power, we can instead use an LMX2571 [8] as the synthesizer which has higher phase noise, but lower power consumption. We can also use a CC1910 [4] as the PA which operates efficiently at 20 dBm output power. Similarly, for output powers of 4 dBm and 10 dBm, we can use a CC1310 [14] as the synthesizer and eliminate the PA. These optimizations would reduce power consumption to levels shown in Table 1. Since we built our system for maximum output power and range, we did not make these hardware changes in this work, but we wish to outline the available design choices for use-cases demanding lower power consumption.

5.2 Cost Analysis

The FD LoRa Backscatter reader is designed with the goal of simplifying the deployment of backscatter technology to unleash the untapped potential of backscatter. Cost plays a critical role in achieving this objective. Table 2 outlines the cost structure of the different components of the system and compares it with a legacy HD LoRa backscatter reader. Our analysis using pricing data from Octopart [1] shows that at low volumes of 1,000 units, the FD reader costs $27.54, only 10% more than the cost of two HD readers. We believe that further optimization and leveraging economies of scale, coupled with the reuse of radios and processing power upon integration with existing devices such as IoT gateways, smartphones, and tablets, can further reduce the solution cost.

5.3 LoRa Backscatter Tag

The LoRa backscatter tag used in this work is based on the design proposed in [84]. The LoRa baseband and subcarrier chirp-spread-spectrum-modulated packets are generated using Direct Digital Synthesis (DDS) on an AGLN250 Igloo Nano FPGA [5]. The output of the FPGA is connected to SP4T ADG904 RF switch [6] to synthesize single-side-band backscatter packets. The backscatter tag design also incorporates an On-Off Keying (OOK) based wake-on radio with sensitivity down to -55 dBm and an ADG919 [7] SPDT switch to multiplex a 0 dBi omnidirectional PIFA between the receiver and the backscatter switching network. The total loss in the RF path (SPDT + SP4T) for backscatter is $\sim$ 5 dB.

Table 1: Estimated Power Consumption of FD LoRa Backscatter Reader

TX Power	Applications	Peak Power
30 dBm	Plugged-in devices	3,040 mW$^+$
20 dBm	Laptops, Tablets	675 mW
10 dBm	Phones, Battery Packs	149 mW
4 dBm	Phones, Battery Packs	112 mW

$^+$ Measured result.

Table 2: Cost Analysis of FD & HD Backscatter Hardware

Components	FD Cost	(2×) HD Cost
Transceiver	$4.16	(2×) $4.16
Synthesizer	$7.15	N/A
Power Amplifier	$1.33	(2×) $1.33
Cancellation Network	$5.78	N/A
MCU	$1.70	(2×) $1.30
Power Management	$2.25	(2×) $1.95
Passives	$2.52	(2×) $1.54
PCB fabrication [2]	$1.07	(2×) $0.79
Assembly [2]	$1.58	(2×) $1.38
Total	**$27.54**	**$24.90**

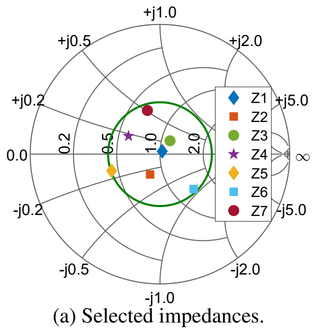
6 Evaluation

First, we validate our cancellation approach by measuring the carrier and offset cancellation of our novel two-stage impedance tuning network. Then, we measure the time overhead incurred by our tuning approach. Next, we evaluate the FD LoRa Backscatter system performance in a wired setup to neutralize multi-path effects, followed by line-of-sight and non-line-of-sight wireless deployments. Finally, we measure the performance of the mobile version of our system.

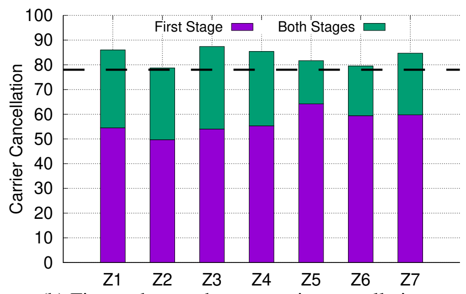
Unless mentioned otherwise, we set the subcarrier modulation frequency to 3 MHz, and configure the tag to transmit 1,000 packets with $SF = 12$, $BW = 250kHz$, (8,4) Hamming Code with an 8-byte payload, a sequence number for calculating PER, and a 2-byte CRC. Additionally, we initiate uplink by sending a downlink OOK-modulated packet at 2 kbps to wake up the tag and align the tag’s backscatter operation to the carrier. Downlink also enables channel arbitration between multiple tags, sending acknowledgments, packet re-transmissions, and other functionalities [56,84]. The evaluation of these features is beyond the scope of this work.

6.1 Cancellation Network

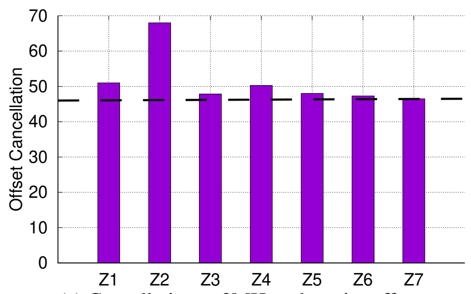
The cancellation network performance depends on the antenna impedance, which is sensitive to environmental variations (see §3.2). To demonstrate that our system can achieve the required cancellation across a range of expected antenna impedances, we simulate antenna impedances with custom circuit boards with an 0402 footprint and an SMA connector. We populate the pads with discrete passives to represent antenna impedances with $0 \leq |\Gamma| \leq 0.4$. We test seven antenna



(a) Selected impedances.



(b) First and second stage carrier cancellation.



(c) Cancellation at 3MHz subcarrier offset.

Figure 6: SI cancellation as a function of variation in antenna impedance.

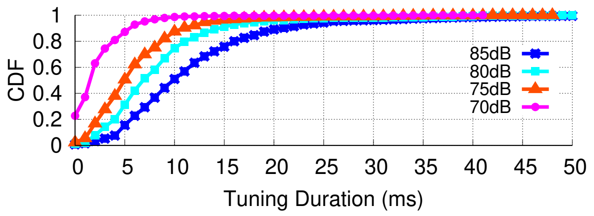


Figure 7: **Tuning algorithm overhead.** We measure the overhead for different thresholds over 10000 packets.

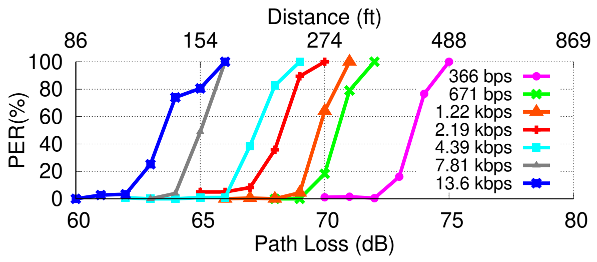


Figure 8: **Receiver Sensitivity Analysis.** We plot the PER as a function of path loss for different data rates.

impedances, as shown on the smith chart in Fig. 6 (a).

To measure cancellation, we attach a board representing an antenna impedance to the antenna port of our FD LoRa Backscatter reader with a Murata measurement probe [15]. We disconnect the receiver and attach the receiver input to a spectrum analyzer using another RF probe. We set the transmitter to 915 MHz and 30 dBm output power. Since the receiver is disconnected, we cannot measure RSSI and, therefore, cannot utilize the tuning algorithm. We manually set the capacitor states in a two-step process similar to the tuning algorithm. First, we fix the second-stage capacitors and manually tune the first stage for the best SI cancellation, then, we manually tune the second stage. Fig. 6(b) shows the SI carrier cancellation results for one- and two-stage tunable impedance networks. Results show that a single stage is insufficient to achieve 78 dB carrier cancellation, whereas the two-stage design meets the specification. Next, we measure offset cancellation by keeping the same capacitor configuration and sweeping the carrier source between 905 - 925 MHz in 100 kHz frequency increments. Fig.6(c) shows the offset cancellation for different antenna impedances at 3 MHz offset. Our results show that we achieve our target of 46.5 dB offset cancellation for all antenna impedances.

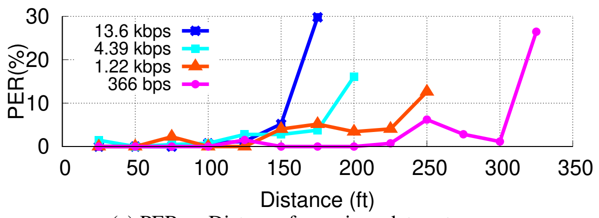
6.2 Tuning Overhead

To measure the performance of our tuning algorithm, we place the FD LoRa Backscatter reader with the PIFA on a table in a typical office environment. We collect 10,000 packets from a tag placed 20 ft away over the course of 80 minutes with multiple people sitting nearby and walking in the vicinity of the system. We modify the target SI cancellation threshold in the tuning algorithm to 70, 75, 80, and 85 dB and run experiments to measure the time required for tuning. The tuning algorithm was able to achieve the target SI in 99% cases. We plot the CDF of tuning overhead for different cancellation thresholds in Fig. 7. As expected, the tuning duration increases with the target threshold. For a threshold of 80 dB, the average tuning duration is 8.3 ms, corresponding to an overhead of 2.7%. The RSSI measurements from the SX1276 chipset are noisy, and we take the mean over 8 readings for each tuning step, which is the key limitation. Each tuning step takes about 0.5 ms, dominated by the SPI transactions and settling time of the receiver. An RF power detector, which is beyond the scope of this work, can be used to provide faster feedback at the expense of increased cost.

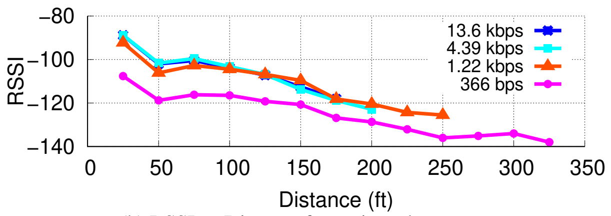
6.3 Receiver Sensitivity Analysis

To evaluate the receive sensitivity of the FD LoRa Backscatter system without the effect of multi-path signal propagation, we create an equivalent wired setup. We use RF cables and a variable attenuator to connect the antenna port of the FD LoRa Backscatter reader to a LoRa backscatter tag. We vary the in-line attenuator to simulate path loss, corresponding to different operating distances between the reader and the tag. We start with an attenuator value at which we receive all packets and then slowly increase the attenuation until no packets are received. We configure the SF and BW parameters to cover a range of sensitivity and data rates.

Fig. 8 plots PER as a function of path loss in a wired setup for different data rates. Since sensitivity is inversely proportional to data rate, lower data rates can operate at higher path loss, which translates to longer operating distances. For a $PER \leq 10\%$, the expected LOS range at the lowest data-

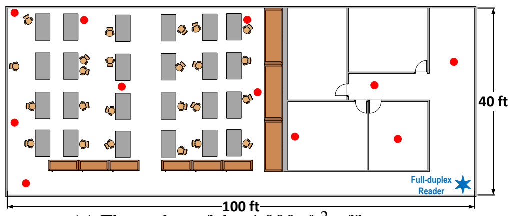


(a) PER vs Distance for various data-rates.

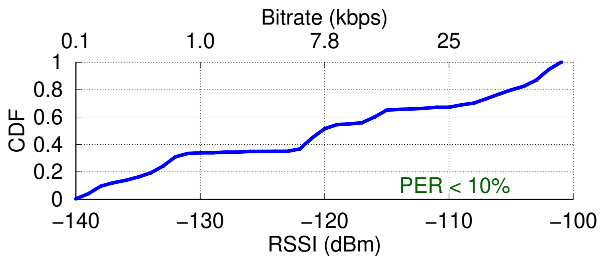


(b) RSSI vs Distance for various data-rates.

Figure 9: **Line-of-Sight Wireless Tests.** We move the backscatter tag away from the reader in line of sight.



(a) Floor plan of the 4,000 ft^2 office space.



(b) RSSI of all measurements in the office space.

Figure 10: **Non-Line-of-Sight Wireless Tests.** We place the backscatter tag at 10 locations shown as red dots.

rate of 366 bps ($SF = 12, BW = 250\ KHz$) is 340 ft, with the range decreasing successively for higher bit rates, down to 110 ft for 13.6 kbps ($SF = 7, BW = 500\ KHz$).

6.4 Line-of-Sight (LOS) Wireless Testing

We deploy the FD LoRa Backscatter system in a nearby park to measure LOS performance. For best performance, we configure the reader as a base-station (see §5) by connecting an 8 dBiC circularly polarized patch antenna [25], placed on a 5 ft tall stand, to the antenna port and set transmit power to 30 dBm. We place the tag at the same height and move it away in 25 ft increments. Fig. 9 plots PER and RSSI as a function of distance for four different data rates. Our results show that, for $PER < 10\%$, at the lowest data rate, the system can operate at a distance of up to 300 ft with a reported RSSI of -134 dBm. For the highest data rate, the operating distance was 150 ft at -112 dBm RSSI.

A prior HD LoRa backscatter system reported a range of 475 m between the two radios [84]; this corresponds to a range of 780 ft for an FD system. Our FD system achieves a shorter range and this can be attributed to two factors. First, the HD system evaluation uses a -143 dBm sensitivity protocol at 45 bps versus the -134 dBm sensitivity at 366 bps used in this work. The 45 bps packets are 2.4 sec long, 6 $\times$ the FCC maximum channel dwell time (see §2.1). Additionally, the FD system uses a hybrid coupler architecture. This reduces cost, but incurs a 7 dB loss (see §5). So, in total, our link budget is reduced by 16 dB. This translates to a 2.5 $\times$ range reduction, close to the 300 ft range of our system.

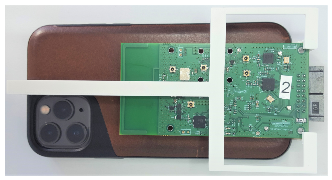
6.5 Non-Line-of-Sight (NLOS) Wireless Tests

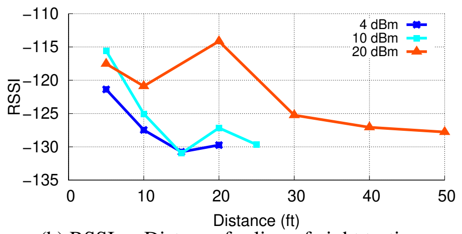
Next, we set up in a 100 ft $\times$ 40 ft office building to evaluate performance in a more realistic NLOS scenario. We place the base-station reader in one corner of the building and move the tag to ten locations to measure performance through cubicles,multiple concrete and glass walls, and down hallways. The floor plan of the building is shown in Fig.10(a). The blue star in the lower-right corner indicates the position of the FD reader, and the red dots indicate the different locations of the tag throughout the office space. We transmit 1,000 packets at each location, and a CDF of the aggregated RSSI data from the test is shown in Fig.10(b). We observed a median RSSI of -120 dBm and PER of less than 10% at all the locations demonstrating that the FD LoRa Backscatter system is fully operational in the office space with a coverage area of 4,000 ft^2.

6.6 Integration with Mobile Devices

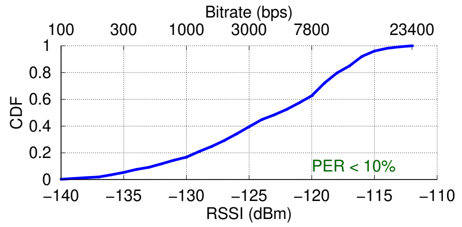
Finally, we evaluate the performance of the mobile version (see §5) of the FD reader. We attach the mobile reader to the back of an iPhone 11 Pro, as shown in Fig. 11(a) and configure the reader to transmit at 4 dBm, 10 dBm, and 20 dBm to resemble the transmit power of mobile devices. We move a backscatter tag away from the reader in 5 ft increments until PER > 10%. Fig. 11(b) plots the RSSI of the received packets as a function of distance. The plots show that at 4 dBm, the mobile reader operates up to 20 ft and the range increases beyond 50 ft (the length of the room and limit of our testing) for a transmit power of 20 dBm. These distances are sufficient for connecting peripheral, wearable, and medical devices to a smartphone using backscatter at extremely low cost, small size, and low power consumption. These experiments were done in an uncontrolled wireless environment and the variation in signal strength at different locations is due to multi-path effects, which is typical of practical wireless testing.

To demonstrate that our system can adapt to variations in environment and antenna impedance, we place the FD LoRa Backscatter enabled smartphone in a subject's pocket and set the transmit power to 4 dBm. We place a tag at the center of an


(a) FD LoRa Backscatter on the back of a smartphone.



(b) RSSI vs Distance for line-of-sight testing.



(c) Packet RSSI with the mobile reader in a user's pocket moving around a table.

Figure 11: Integration with Mobile Devices.

11 ft $\times$ 6 ft table, and the subject walks around the perimeter of the table, receiving more than 1,000 packets. The performance is reliable with PER < 10%, which demonstrates the efficacy of our tuning algorithm. Fig. 11(c) plots the CDF of RSSI for all the packets. The backscatter tag measures 2 in $\times$ $1\frac{1}{2}$ in, resembling the size of a pill bottle. This demonstrates that a mobile smartphone can use backscatter to communicate with a prescription pill bottle or insulin pen, allowing tracking of medication and drug delivery.

7 Applications

We demonstrate two applications for our FD system. First, we show how a mobile reader can collect data from a smart contact lens, a particularly challenging RF environment. Next, we demonstrate a precision agriculture application by mounting the reader to the bottom of a drone, which can be flown over farms and use backscatter to collect data from sensors distributed in a field. The use of a single reader coupled with a highly sensitive long-range backscatter protocol enables these applications, even in these challenging deployments.

7.1 Contact Lens

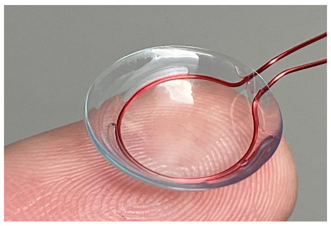
We use the mobile FD LoRa Backscatter system mounted on the back of a smartphone to communicate with a backscatter tag equipped with a smart-contact-lens-form-factor antenna. We use the same backscatter endpoint as with other tests, but we cut off the original PIFA and replace it with a small loop antenna of 1 cm diameter made with 30 AWG enameled wire. The antenna is encapsulated in two contact lenses and filled with contact lens solution to simulate the RF environment of an eye-ball, as shown in Fig. 12(a). Due to its small size and the ionic environment of the contact solution, the loop antenna has an expected loss of 15 - 20 dB.

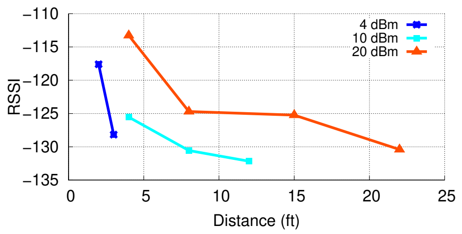
We place the smartphone on a table and configure the mobile reader to transmit at 4, 10, and 20 dBm and move the contact lens backscatter prototype away from the smartphone. Fig. 12(b) shows the RSSI as a function of distance for various transmit powers. We show that the mobile reader at 10 dBm and 20 dBm transmit power can communicate with the contact lens at distances of 12 ft and 22 ft respectively for PER < 10%. Next, we put the mobile reader transmitting at 4 dBm in a 6 ft tall subject's pocket and hold the contact lens prototype near the subject's eye to simulate a realistic use case. Fig. 12(b) plots the CDF of the RSSI of decoded packets when the user was standing and sitting on a chair. The plot shows reliable performance with PER < 10% and a mean RSSI of -125 dBm. This demonstrates the feasibility of using backscatter to provide continuous connectivity between a user's phone and a smart contact lens. This RF-challenged application was made possible even at such a low transmit power due to the high receive sensitivity of the system.

7.2 Drone with an FD Backscatter Reader

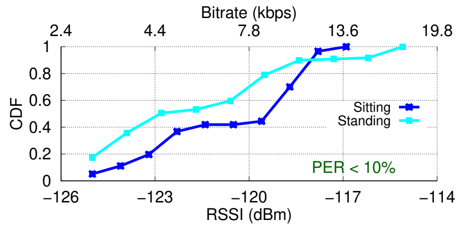
Drones are extensively used for aerial surveillance in precision agriculture [87]. We demonstrate how one can augment a drone's functionality by adding a FD LoRa Backscatter reader to communicate with sensors distributed in a field using backscatter. We attach the mobile version of our reader to the bottom of a low-cost, commercially-available Parrot AR.Drone 2.0 quadcopter (<\$80) [24], as shown in Fig.13(a). We power the reader from the drone's battery using a USB connector to demonstrate the ease of integrating our system. We set the transmit power to 20 dBm to reduce the burden on 7.5 Whr battery of the cheap drone. We place the tag on the ground simulating an agriculture sensor and fly the drone at a height of 60 ft. Due to practical challenges in accurately positioning the drone, we allow the drone to fly laterally during the test up to 50 ft from the center, which results in 80 ft maximum separation from the tag. This corresponds to an instantaneous coverage area of 7,850 ft^2. We collect more than 400 packets over 4 minutes with the drone flying around the coverage zone while keeping its altitude constant.

Fig.13(b) plots CDF of the RSSI of the packets received by the drone over the entire duration of the test for a PER <10%. With a minimum of -136 dBm and median of -128 dBm, this demonstrates good performance for the area tested. With a flight time of 15 min and a top speed of 11 m/s, our cheap drone could, in theory, cover an area greater than 60 acres on a single charge. With a more powerful drone with higher payload capacity and longer flight time, one could integrate a higher gain antenna and transmit at higher power. This would result in a greater instantaneous coverage area and, with longer flight time, could achieve many times greater coverage on a single charge.


(a) Contact lens antenna prototype.

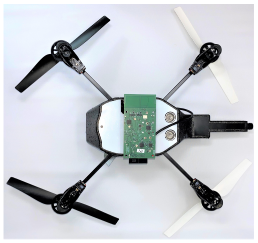


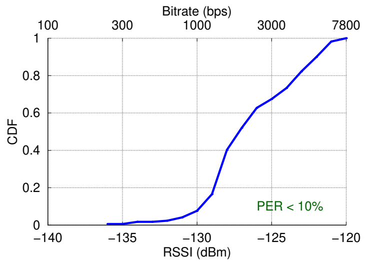
(b) RSSI vs Distance for different transmit power.



(c) RSSI when reader is inside a user's pocket.

Figure 12: A mobile FD LoRa Backscatter reader communicating with a contact lens prototype.


(a) Reader mounted on the drone.



(b) RSSI when drone is at 60 ft altitude.

Figure 13: Backscatter enabled Low-Cost Drone.

8 Related Work

Our work is related to prior efforts in HD backscatter, FD backscatter, and in-band FD communication.

Half-Duplex Backscatter. Our work builds on recent efforts in developing backscatter solutions that are compatible with existing wireless standards such as Bluetooth [43, 47, 53, 89], WiFi [26, 47, 56, 89, 90, 92], Zigbee [64], and LoRa [73, 84] using half-duplex architectures. The backscatter endpoint is based on prior LoRa backscatter design [84], but we take the next step of integrating the single-tone carrier source and LoRa receiver into a single device.

In addition to standards-compliant backscatter, proprietary-protocol communications [46, 78, 79, 93] (to improve data rates and throughput), applications such as wireless video streaming [49, 81], indoor localization [67, 68, 71], and human activity recognition [27, 34, 80] have been realized with HD deployments. The techniques presented in this work can be extended to build an FD version of these systems.

Full-Duplex Backscatter. A BLE monostatic/FD backscatter system was introduced in [42] that uses a 20 dB coupler, phase shifter, and variable attenuator for SI cancellation. However, due to additional losses in the coupler and limited cancellation depth, the communication range was limited to 3 m, even with 33 dBm of output power. In [30], an FD backscatter system using an SDR platform with analog and digital cancellation was introduced where WiFi packets are used as the carrier and the tag backscatters proprietary BPSK, QPSK, and 16-PSK modulated packets which were decoded by the SDR up to a distance of 7 m. Due to the wide-band nature of WiFi carrier signals, [30] needs wide-band SI cancellation. A circulator and a 10 cm $\times$ 10 cm custom PCB with 16 variable-gain delay lines are added to the SDR platform for wide-band analog cancellation, increasing both the solution cost and size. In contrast, the FD LoRa Backscatter system uses commodity LoRa radios and passive components for cancellation and can receive standard LoRa packets up to a distance of 300 ft.

RFID readers are also full-duplex devices that transmit a single-tone carrier and receive backscattered packets from RFID tags. However, EPC Gen2 readers are bulky, expensive [18, 22], and cannot compete with economies of scale of standard protocols. The operating range of passive RFID readers is determined by the power-harvesting sensitivity, not by the backscatter communication link. RFID readers operate at relatively short distances, even if the tag is powered from an alternative energy source [51], due to poor receive sensitivity (-90 dBm) [18, 22]. In contrast, our FD LoRa Backscatter system achieves much longer operating distances at significantly lower cost by leveraging a highly sensitive commodity LoRa receiver, cheap passive components, and a microcontroller for deep SI cancellation. Low-cost RFID readers based on directional couplers have also been investigated [50, 54], but they suffer from high Rx insertion loss and lower SI cancellation depth, which reduces range. In [69], a full-duplex drone relay is presented to extend the range of a fixed RFID reader. The topology requires an additional relay on a drone to extend the fixed RFID reader range to 50 m. In contrast, our FD reader can be mounted on a flying drone to cover a significantly larger area.

In-Band Full-Duplex Radios. In-band full-duplex (IBFD) radios simultaneously transmit and receive at the same frequency. Recent works have used a combination of isolation, analog cancellation, and digital cancellation techniques to suppress SI below the receiver noise floor [31, 35, 41, 62].

Existing isolation techniques use two or more physically-separated antennas [40, 41, 44, 63], discrete circulators [31, 35, 57, 62, 91], integrated circulators [39, 77], or hybrid junctions [38, 65] to isolate transmitter and receiver. The use of multiple antennas increases form factor while achieving limited isolation. Discrete circulators [9] are bulky and expensive. While recent advances in integrated circulators [39, 77] are promising, these devices are unable to handle the 30 dBm output-power requirement. Hybrid junctions, realized using couplers [50, 54] (such as the 3 dB coupler used in this work) or electrical balance duplexers (EBD) [38, 65], incur a loss, but result in small-form-factor and inexpensive solutions. However, existing solutions with COTS components cannot

Table 3: Comparison of State of the Art Analog SI Cancellation Techniques

Reference	Cancellation Technique	TX Signal	RX Signal	Analog Can.	TX Power	Active Comp.+	Size	Cost
[41]	Multiple antenna + auxiliary can. path	WiFi Packet	WiFi Packet	65 dB	8 dBm	Yes	37 cm Ant. Separation	High
[35]	Circulator + 2 tap freq. domain equalization	WiFi Packet	WiFi Packet	52 dB	10 dBm	Yes	1.5×4.0 cm^2	High
[62]	Circulator + 3 complex-tap analog FIR filter	WiFi Packet	WiFi Packet	68 dB	8 dBm	Yes	N.A.	High
[38]	EBD + Double RF adaptive filter	General	General	72 dB	12 dBm	Yes	Custom ASIC*	
[77]	Magnetic-free N-path filter-based Circulator	General	General	40 dB	8 dBm	No	Custom ASIC*	
[65]	EBD + passive tuning network	General	General	75 dB	27 dBm	No	Custom ASIC*	
[30]	Circulator + 16 tap analog FIR filter	WiFi Packet	WiFi Backscatter	60 dB	20 dBm	No	10×10 cm^2	High
[42]	20dB Coupler + Active tuning network	CW	BLE Backscatter	50 dB	33 dBm	Yes	N.A.	High
[55]	10dB Coupler + Atten. + passive tuning network	CW	EPC Gen 2	60 dB	26 dBm	No	2.7×2.0 cm^2	Low
This Work	Hybrid Coupler + passive tuning network	CW	LoRa Backscatter	78 dB	30 dBm	No	2.5×0.8 cm^2	Low

+ Active components include phase shifters, vector modulators, amplifiers and transconductance amplifiers which can contribute additional noise.
* Custom ASICs are incompatible with COTS transceivers and are only viable and cost-efficient at scale.

achieve 78 dB of SI cancellation [42, 50, 54], whereas our proposed two-stage impedance tuning network can be used to achieve this deep cancellation required for building a cheap, low-complexity, long-range FD reader. Furthermore, [38, 65] show a path towards further reducing the cost and size at high volumes by integrating the hybrid junction in silicon.

Analog feed-forward cancellers can be combined with isolation techniques to enhance SI cancellation depth and bandwidth. Various feed-forward PCB and ASIC implementations based on vector modulation [32], finite impulse response filters [31, 38, 62, 91], frequency domain equalization based RF filters [35], N-path filters [94], and Hilbert transform equalization [82] have been proposed. However, these techniques require additional active circuitry, which has a limited power-handling capability and generates noise that increases the receiver noise floor [37]. Furthermore, COTS phase shifters [23] and vector modulators [21] are bulky and expensive, which substantially increases cost, complexity, and form factor. In this work, we utilize the two-stage tunable impedance network architecture to achieve the required 78 dB cancellation depth. Since the transmitter carrier signal is only single-tone, we do not need the feed-forward paths to improve the bandwidth.

In Table 3, we summarize the state-of-the-art techniques used for analog SI cancellation and compare them with our approach in terms of cancellation depth, transmit power-handling capability, size, and cost.

Finally, digital cancellers are often used to further suppress the SI below the receiver noise floor [31, 35, 58, 61] using both linear and non-linear cancellation techniques [52]. Digital cancellation requires access to baseband IQ samples. This function is not available on commodity radio chipsets and is typically implemented on SDRs [28, 31, 35, 83], FPGAs [85], or DSPs [29], which are prohibitively expensive. Instead, since our interference is a single-tone at a frequency offset from the receive channel, we leverage the inherent baseband filtering capabilities of the commodity receiver to suppress the SI at the offset frequency.

9 Conclusion

We present the first low-cost, long-range, small-form-factor Full-Duplex LoRa Backscatter design. We use a single-antenna, hybrid-coupler-based architecture and introduce a novel two-stage tunable impedance network to meet the stringent SI-cancellation requirements using only passive components and a microcontroller. We build hardware using COTS components and prototype proof-of-concept applications for a smart contact lens and backscatter enabled drone.

Acknowledgments

The authors would like to thank Aaron Parks, Jonathan Hamberg, and Ye Wang for their help. We would also like to thank the anonymous reviewers as well as our shepherd, Prof. Lin Zhong for their helpful feedback on the paper.

References

[1] www.octopart.com.

[2] www.pcbminions.com.

[3] Agilent 8753es Vector Network Analyzer, 2001. https://www.keysight.com/en/pd-1000002292%3Aepsg%3Apro-pn-8753ES.

[4] CC1190 datasheet, 2010. https://www.ti.com/product/CC1190.

[5] AGLN250 datasheet, 2014. https://www.microsemi.com/product-directory/fpgas/1689-igloo#igloo-nano.

[6] ADG904 datasheet, 2016. https://www.analog.com/media/en/technical-documentation/data-sheets/ADG904.pdf.

[7] ADG919 datasheet, 2016. https://www.analog.com/media/en/technical-documentation/data-sheets/ADG918_919.pdf.

[8] LMX2571 datasheet, 2016. https://www.ti.com/product/LMX2571.

[9] SKYFR-001400 datasheet, 2016. https://www.skyworksinc.com/en/Products/Circulators-and-Isolators/SKYFR-001400.

[10] ADF4351 datasheet, 2017. https://www.analog.com/media/en/technical-documentation/data-sheets/ADF4351.pdf.

[11] PE64906 datasheet, 2017. https://www.psemi.com/pdf/datasheets/pe64906ds.pdf.

[12] SKY65313-21 datashett, 2017. https://www.skyworksinc.com/en/Products/Front-end-Modules/SKY65313-21.

[13] stm32F413RG datasheet, 2017. https://www.st.com/resource/en/datasheet/stm32f413rg.pdf.

[14] CC1310 datasheet, 2018. https://www.ti.com/product/CC1310.

[15] MM8430 datasheet, 2018. https://media.digikey.com/pdf/Data%20Sheets/Murata%20PDFs/Microwave_Coaxial_Conn_Cat030E.pdf.

[16] Anaren X3C09P1-03S datasheet, 2019. https://www.anaren.com/catalog/xinger/90-degree-hybrid-couplers.

[17] Electronic Code of Federal Regulations, Title 47, Chapter i, Subchapter A, Part 15, Subpart C, section 15.247, 2019. www.ecfr.gov.

[18] ST25RU3993 datasheet, 2019. https://www.st.com/en/nfc/st25ru3993.html.

[19] SX1276 datasheet, 2019. https://www.semtech.com/products/wireless-rf/lora-transceivers/sx1276.

[20] 3GPP TS 36.101 V16.6.0 , 2020. https://portal.3gpp.org/.

[21] HMC630 datasheet, 2020. https://www.analog.com/en/products/hmc630.html#product-overview.

[22] Impinj Speedway Revolution R120 UHF RFID Reader, 2020. https://www.atlasrfidstore.com/rfid-readers/.

[23] JSPHS-1000 datasheet, 2020. https://www.minicircuits.com/pdfs/JSPHS-1000+.pdf.

[24] Parrot AR Drone 2.0, 2020. https://www.parrot.com/us/drones/parrot-ardrone-20-elite-edition.

[25] S9028PCL Circular Polarity Antenna datasheet, 2020. https://connectivity-staging.s3.us-east-2.amazonaws.com/s3fs-public/2018-10/ANT-DS-S9028PCL%20S9028PCR-0515.pdf.

[26] Ali Abedi, Mohammad Hossein Mazaheri, Omid Abari, and Tim Brecht. Witag: Rethinking backscatter communication for wifi networks. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks - HotNets '18*. ACM Press, 2018.

[27] Abeer Ahmad, Akshay Athalye, Milutin Stanacević, and Samir R. Das. Collaborative channel estimation in backscattering tag-to-tag networks. In *Proceedings of the 1st ACM International Workshop on Device-Free Human Sensing - DFHS'19*. ACM Press, 2019.

[28] Rami Akeela and Behnam Dezfouli. Software-defined radios: Architecture, state-of-the-art, and challenges. *Computer Communications*, 128:106–125, sep 2018.

[29] Manu Bansal, Aaron Schulman, and Sachin Katti. Atomix: A framework for deploying signal processing applications on wireless infrastructure. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 173–188, Oakland, CA, May 2015. USENIX Association.

[30] Dinesh Bharadia, Kiran Raj Joshi, Manikanta Kotaru, and Sachin Katti. Backfi: High throughput wifi backscatter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication - SIGCOMM '15*. ACM Press, 2015.

[31] Dinesh Bharadia, Emily McMilin, and Sachin Katti. Full Duplex Radios. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 375–386, New York, NY, USA, 2013. ACM.

[32] D. J. van den Broek, E. A. M. Klumperink, and B. Nauta. An In-Band Full-Duplex Radio Receiver With a Passive Vector Modulator Downmixer for Self-Interference Cancellation. *IEEE Journal of Solid-State Circuits*, 50(12):3003–3014, December 2015.

[33] Bruce Brown. Smart pill bottle, 2019. https://healthtechinsider.com/2019/05/13/smart-pill-bottle/.

[34] Michael Buettner, Richa Prasad, Matthai Philipose, and David Wetherall. Recognizing daily activities with RFID-based sensors. In *Proceedings of the 11th international conference on Ubiquitous computing - Ubicomp '09*. ACM Press, 2009.

[35] Tingjun Chen, Mahmood Baraani Dastjerdi, Jin Zhou, Harish Krishnaswamy, and Gil Zussman. Wideband full-duplex wireless via frequency-domain equalization. In *The 25th Annual International Conference on Mobile Computing and Networking*. ACM, aug 2019.

[36] Julian Chokkattu. The Display of the Future Might Be in Your Contact Lens, 2020. https://www.wired.com/story/mojo-vision-smart-contact-lens/.

[37] K. D. Chu, M. Katanbaf, C. Su, T. Zhang, and J. C. Rudell. Integrated CMOS transceivers design towards flexible full duplex (FD) and frequency division duplex (FDD) systems. In *2018 IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–11, April 2018.

[38] Kun-Da Chu, Mohamad Katanbaf, Tong Zhang, Chenxin Su, and Jacques C. Rudell. A broadband and deep-TX self-interference cancellation technique for full-duplex and frequency-domain-duplex transceiver applications. In *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*. IEEE, feb 2018.

[39] Mahmood Baraani Dastjerdi, Sanket Jain, Negar Reiskarimian, Arun Natarajan, and Harish Krishnaswamy. 28.6 full-duplex 2×2 MIMO circulator-receiver with high TX power handling exploiting MIMO RF and shared-delay baseband self-interference cancellation. In *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*. IEEE, feb 2019.

[40] Tolga Dinc and Harish Krishnaswamy. A t/r antenna pair with polarization-based reconfigurable wideband self-interference cancellation for simultaneous transmit and receive. In *2015 IEEE MTT-S International Microwave Symposium*. IEEE, may 2015.

[41] M. Duarte, A. Sabharwal, V. Aggarwal, R. Jana, K. K. Ramakrishnan, C. W. Rice, and N. K. Shankaranarayanan. Design and Characterization of a Full-Duplex Multiantenna System for WiFi Networks. *IEEE Transactions on Vehicular Technology*, 63(3):1160–1177, March 2014.

[42] Joshua F. Ensworth, Alexander T. Hoang, Thang Q. Phu, and Matthew S. Reynolds. Full-duplex bluetooth low energy (BLE) compatible backscatter communication system for mobile devices. In *2017 IEEE Topical Conference on Wireless Sensors and Sensor Networks (WiSNet)*. IEEE, jan 2017.

[43] Joshua F. Ensworth and Matthew S. Reynolds. BLE-backscatter: Ultralow-power IoT nodes compatible with bluetooth 4.0 low energy (BLE) smartphones and tablets. *IEEE Transactions on Microwave Theory and Techniques*, 65(9):3360–3368, sep 2017.

[44] E. Everett, A. Sahai, and A. Sabharwal. Passive Self-Interference Suppression for Full-Duplex Infrastructure Nodes. *IEEE Transactions on Wireless Communications*, 13(2):680–694, February 2014.

[45] Bradford W. Gildon. InPen smart insulin pen system: Product review and user experience. *Diabetes Spectrum*, 31(4):354–358, sep 2018.

[46] Mehrdad Hessar, Ali Najafi, and Shyamnath Gollakota. Netscatter: Enabling large-scale backscatter networks. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI'19, pages 271–283, Berkeley, CA, USA, 2019. USENIX Association.

[47] Vikram Iyer, Vamsi Talla, Bryce Kellogg, Shyamnath Gollakota, and Joshua Smith. Inter-technology backscatter: Towards internet connectivity for implanted devices. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 356–369, New York, NY, USA, 2016. ACM.

[48] Cheonhoo Jeon, Jahyun Koo, Kyongsu Lee, Minseob Lee, Su-Kyoung Kim, Sangbaie Shin, Sei Kwang Hahn, and Jae-Yoon Sim. A smart contact lens controller IC supporting dual-mode telemetry with wireless-powered backscattering LSK and EM-radiated RF transmission using a single-loop antenna. *IEEE Journal of Solid-State Circuits*, 55(4):856–867, apr 2020.

[49] Colleen Josephson, Lei Yang, Pengyu Zhang, and Sachin Katti. Wireless computer vision using commodity radios. In *Proceedings of the 18th International Conference on Information Processing in Sensor Networks - IPSN '19*. ACM Press, 2019.

[50] Sung-Chan Jung, Min-Su Kim, and Youngoo Yang. A reconfigurable carrier leakage canceler for UHF RFID reader front-ends. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 58(1):70–76, jan 2011.

[51] Sai Nithin R. Kantareddy, Ian Mathews, Rahul Bhattacharyya, Ian Marius Peters, Tonio Buonassisi, and Sanjay E. Sarma. Long range battery-less PV-powered RFID tag sensors. *IEEE Internet of Things Journal*, 6(4):6989–6996, aug 2019.

[52] Mohamad Katanbaf, Kun-Da Chu, Tong Zhang, Chenxin Su, and Jacques C. Rudell. Two-way traffic ahead: RF\analog self-interference cancellation techniques and the challenges for future integrated full-duplex transceivers. *IEEE Microwave Magazine*, 20(2):22–35, feb 2019.

[53] Mohamad Katanbaf, Vivek Jain, and Joshua R. Smith. Relacks: Reliable backscatter communication in indoor environments. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 4(2):1–24, jun 2020.

[54] Edward A. Keehr. A low-cost, high-speed, high-resolution, adaptively tunable microwave network for an SDR UHF RFID reader reflected power canceller. In *2018 IEEE International Conference on RFID (RFID)*. IEEE, apr 2018.

[55] Edward A. Keehr. A low-cost software-defined UHF RFID reader with active transmit leakage cancellation. In *2018 IEEE International Conference on RFID (RFID)*. IEEE, apr 2018.

[56] Bryce Kellogg, Vamsi Talla, Shyamnath Gollakota, and Joshua R. Smith. Passive wi-fi: Bringing low power to wi-fi transmissions. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, pages 151–164, Berkeley, CA, USA, 2016. USENIX Association.

[57] Seiran Khaledian, Farhad Farzami, Besma Smida, and Danilo Erricolo. Inherent self-interference cancellation for in-band full-duplex single-antenna systems. *IEEE Transactions on Microwave Theory and Techniques*, 66(6):2842–2850, jun 2018.

[58] A. Kiayani, M. Z. Waheed, L. Anttila, M. Abdelaziz, D. Korpi, V. Syrjälä, M. Kosunen, K. Stadius, J. Ryynänen, and M. Valkama. Adaptive Nonlinear RF Cancellation for Improved Isolation in Simultaneous Transmit #x2013;Receive Systems. *IEEE Transactions on Microwave Theory and Techniques*, 66(5):2299–2312, May 2018.

[59] Kai Klinker, Manuel Wiesche, and Helmut Krcmar. Smart glasses in health care: A patient trust perspective. In *Proceedings of the 53rd Hawaii International Conference on System Sciences*. Hawaii International Conference on System Sciences, 2020.

[60] David C. Klonoff and David Kerr. Smart pens will improve insulin therapy. *Journal of Diabetes Science and Technology*, 12(3):551–553, feb 2018.

[61] D. Korpi, M. Heino, C. Icheln, K. Haneda, and M. Valkama. Compact Inband Full-Duplex Relays With Beyond 100 dB Self-Interference Suppression: Enabling Techniques and Field Measurements. *IEEE Transactions on Antennas and Propagation*, 65(2):960–965, February 2017.

[62] D. Korpi, J. Tamminen, M. Turunen, T. Huusari, Y. S. Choi, L. Anttila, S. Talwar, and M. Valkama. Full-duplex mobile device: pushing the limits. *IEEE Communications Magazine*, 54(9):80–87, September 2016.

[63] Abhishek Kumar and Sankaran Aniruddhan. A 2.35 GHz cross-talk canceller for 2×2 MIMO full-duplex wireless system. In *2019 IEEE MTT-S International Microwave Symposium (IMS)*. IEEE, jun 2019.

[64] Yan Li, Zicheng Chi, Xin Liu, and Ting Zhu. Passive-zigbee: Enabling zigbee communication in IoT networks with 1000x+ less power consumption. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems - SenSys '18*. ACM Press, 2018.

[65] B. van Liempd, B. Hershberg, S. Ariumi, K. Raczkowski, K. F. Bink, U. Karthaus, E. Martens, P. Wambacq, and J. Craninckx. A 70 dBm IIP3 Electrical-Balance Duplexer for Highly Integrated Tunable Front-Ends. *IEEE Transactions on Microwave Theory and Techniques*, 64(12):4274–4286, December 2016.

[66] Vincent Liu, Aaron Parks, Vamsi Talla, Shyamnath Gollakota, David Wetherall, and Joshua R. Smith. Ambient backscatter: Wireless communication out of thin air. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 39–50, New York, NY, USA, 2013. ACM.

[67] Zhihong Luo, Qiping Zhang, Yunfei Ma, Manish Singh, and Fadel Adib. 3d backscatter localization for fine-grained robotics. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 765–782, Boston, MA, February 2019. USENIX Association.

[68] Yunfei Ma, Xiaonan Hui, and Edwin C. Kan. 3d real-time indoor localization via broadband nonlinear

backscatter in passive devices with centimeter precision. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking - MobiCom '16*. ACM Press, 2016.

[69] Yunfei Ma, Nicholas Selby, and Fadel Adib. Drone relays for battery-free networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication - SIGCOMM '17*. ACM Press, 2017.

[70] Joseph Mauro, Kelly B. Mathews, and Eric S. Sredzinski. Effect of a smart pill bottle and pharmacist intervention on medication adherence in patients with multiple myeloma new to lenalidomide therapy. *Journal of Managed Care & Specialty Pharmacy*, 25(11):1244–1254, nov 2019.

[71] Rajalakshmi Nandakumar, Vikram Iyer, and Shyamnath Gollakota. 3d localization for sub-centimeter sized devices. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems - SenSys '18*. ACM Press, 2018.

[72] L.I. Parad and R.L. Moynihan. Split-tee power divider. *IEEE Transactions on Microwave Theory and Techniques*, 13(1):91–95, jan 1965.

[73] Yao Peng, Longfei Shangguan, Yue Hu, Yujie Qian, Xianshang Lin, Xiaojiang Chen, Dingyi Fang, and Kyle Jamieson. Plora: a passive long-range data network from ambient lora transmissions. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication - SIGCOMM18*. ACM Press, 2018.

[74] C.Y. Pon. Hybrid-ring directional coupler for arbitrary power divisions. *IEEE Transactions on Microwave Theory and Techniques*, 9(6):529–535, nov 1961.

[75] Philipp A. Rauschnabel and Young K. Ro. Augmented reality smart glasses: an investigation of technology acceptance drivers. *International Journal of Technology Marketing*, 11(2):123, 2016.

[76] Behzad Razavi. *RF microelectronics*. Prentice Hall, Upper Saddle River, NJ, 2nd ed edition, 2012.

[77] Negar Reiskarimian, Mahmood Baraani Dastjerdi, Jin Zhou, and Harish Krishnaswamy. Analysis and design of commutation-based circulator-receivers for integrated full-duplex wireless. *IEEE Journal of Solid-State Circuits*, 53(8):2190–2201, aug 2018.

[78] James Rosenthal, Apoorva Sharma, Eleftherios Kampianakis, and Matthew S. Reynolds. A 25 mbps, 12.4 pJ/b DQPSK backscatter data uplink for the NeuroDisc brain–computer interface. *IEEE Transactions on Biomedical Circuits and Systems*, 13(5):858–867, oct 2019.

[79] Mohammad Rostami, Jeremy Gummeson, Ali Kiaghadi, and Deepak Ganesan. Polymorphic radios: A new design paradigm for ultra-low power communication. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 446–460, New York, NY, USA, 2018. ACM.

[80] Jihoon Ryoo, Yasha Karimi, Akshay Athalye, Milutin Stanaćević, Samir R. Das, and Petar Djurić. BARNET towards activity recognition using passive backscattering tag-to-tag network. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services - MobiSys 18*. ACM Press, 2018.

[81] Ali Saffari, Mehrdad Hessar, Saman Naderiparizi, and Joshua R. Smith. Battery-free wireless video streaming camera system. In *2019 IEEE International Conference on RFID (RFID)*. IEEE, apr 2019.

[82] A. El Sayed, A. Ahmed, A. K. Mishra, A. H. M. Shirazi, S. P. Woo, Y. S. Choi, S. Mirabbasi, and S. Shekhar. A full-duplex receiver with 80mhz bandwidth self-interference cancellation circuit using baseband Hilbert transform equalization. In *2017 IEEE Radio Frequency Integrated Circuits Symposium (RFIC)*, pages 360–363, June 2017.

[83] M. S. Sim, M. Chung, D. Kim, J. Chung, D. K. Kim, and C. B. Chae. Nonlinear Self-Interference Cancellation for Full-Duplex Radios: From Link-Level and System-Level Performance Perspectives. *IEEE Communications Magazine*, 55(9):158–167, 2017.

[84] Vamsi Talla, Mehrdad Hessar, Bryce Kellogg, Ali Najafi, Joshua R. Smith, and Shyamnath Gollakota. Lora backscatter: Enabling the vision of ubiquitous connectivity. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 1(3):1–24, sep 2017.

[85] Lekhobola Tsoeunyane, Simon Winberg, and Michael Inggs. Software-defined radio FPGA cores: Building towards a domain-specific language. *International Journal of Reconfigurable Computing*, 2017:1–28, 2017.

[86] Peter J. M. van Laarhoven and Emile H. L. Aarts. Simulated annealing. In *Simulated Annealing: Theory and Applications*, pages 7–15. Springer Netherlands, 1987.

[87] Deepak Vasisht, Zerina Kapetanovic, Jong-ho Won, Xinxin Jin, Ranveer Chandra, Ashish Kapoor, Sudipta N. Sinha, Madhusudhan Sudarshan, and Sean Stratman. Farmbeats: An iot platform for data-driven agriculture. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, page 515–528, USA, 2017. USENIX Association.

[88] Pengyu Zhang, Dinesh Bharadia, Kiran Joshi, and Sachin Katti. Hitchhike: Practical backscatter using commodity wifi. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*, SenSys '16, pages 259–271, New York, NY, USA, 2016. ACM.

[89] Pengyu Zhang, Colleen Josephson, Dinesh Bharadia, and Sachin Katti. Freerider: Backscatter communication using commodity radios. In *Proceedings of the 13th International Conference on emerging Networking and Technologies - CoNEXT17*. ACM Press, 2017.

[90] Pengyu Zhang, Mohammad Rostami, Pan Hu, and Deepak Ganesan. Enabling practical backscatter communication for on-body sensors. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference - SIGCOMM '16*. ACM Press, 2016.

[91] T. Zhang, C. Su, A. Najafi, and J. C. Rudell. Wideband Dual-Injection Path Self-Interference Cancellation Architecture for Full-Duplex Transceivers. *IEEE Journal of Solid-State Circuits*, 53(6):1563–1576, June 2018.

[92] Jia Zhao, Wei Gong, and Jiangchuan Liu. Spatial stream backscatter using commodity WiFi. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services - MobiSys 18*. ACM Press, 2018.

[93] Renjie Zhao, Fengyuan Zhu, Yuda Feng, Siyuan Peng, Xiaohua Tian, Hui Yu, and Xinbing Wang. OFDMA-enabled wi-fi backscatter. In *The 25th Annual International Conference on Mobile Computing and Networking - MobiCom19*. ACM Press, 2019.

[94] J. Zhou, T. H. Chuang, T. Dinc, and H. Krishnaswamy. Integrated Wideband Self-Interference Cancellation in the RF Domain for FDD and Full-Duplex Wireless. *IEEE Journal of Solid-State Circuits*, 50(12):3015–3031, December 2015.

One Protocol to Rule Them All: Wireless Network-on-Chip using Deep Reinforcement Learning

Suraj Jog[†], Zikun Liu[†], Antonio Franques[†], Vimuth Fernando[†], Sergi Abadal[⋆], Josep Torrellas[†], Haitham Hassanieh[†]
University of Illinois at Urbana Champaign[†], Polytechnic University of Catalonia[⋆]

Abstract– Wireless Network-on-Chip (NoC) has emerged as a promising solution to scale chip multi-core processors to hundreds and thousands of cores. The broadcast nature of a wireless network allows it to significantly reduce the latency and overhead of many-to-many multicast and broadcast communication on NoC processors. Unfortunately, the traffic patterns on wireless NoCs tend to be very dynamic and can change drastically across different cores, different time intervals and different applications. New medium access protocols that can learn and adapt to the highly dynamic traffic in wireless NoCs are needed to ensure low latency and efficient network utilization.

Towards this goal, we present NeuMAC, a unified approach that combines networking, architecture and deep learning to generate highly adaptive medium access protocols for wireless NoC architectures. NeuMAC leverages a deep reinforcement learning framework to create new policies that can learn the structure, correlations, and statistics of the traffic patterns and adapt quickly to optimize performance. Our results show that NeuMAC can quickly adapt to NoC traffic to provide significant gains in terms of latency, throughput, and overall execution time. In particular, for applications with highly dynamic traffic patterns, NeuMAC can speed up the execution time by $1.37\times - 3.74\times$ as compared to 6 baselines.

1 Introduction

Recently, there has been an increasing interest from both industry and academia to scale network-on-chip (NoC) multicore processors to hundreds and thousands of cores [11, 21, 25, 49]. To enable such massive networks on chip, computer architects have proposed to augment NoC multicore processors with wireless links for communication between the cores [7, 9, 54, 65, 91]. The broadcast nature of wireless networks enables the NoC to significantly reduce the number of packets that the cores need to communicate to each other as well as the latency of packet delivery [1, 38]. Both aspects play a central role in scaling the number of cores on an NoC multicore processor (See Background Section 3 for details) [1, 8, 38, 50, 56]. These benefits have motivated RF circuits designers to build and test wireless NoC transceivers and antennas that can deliver multi-Gbps links while imposing a modest overhead (0.4–5.6%) on the area and power consumption of a chip multiprocessor [31, 93, 99, 100].

While the use of wireless can significantly benefit NoCs, it brings on new challenges. In particular, the wireless medium is shared and can suffer from packet collisions. Designing efficient medium access protocols for wireless NoCs is, however, difficult. The traffic patterns in NoCs tend to change drastically across applications. Even during the execution of a single application the traffic pattern can change as fast as tens of microseconds [4, 38]. As a result static MAC protocols such as TDMA, FDMA and CSMA perform poorly [17, 33, 35, 61, 70, 71, 89]. Further, due to thread synchronization primitives likes barriers and locks in parallel programming, the wireless NoC exhibits complex hard-to-model dependencies between packet delivery on the network and execution time. As a result, even adaptive protocols that try to switch between TDMA and CSMA or optimize for long-term throughput [40, 65, 66], perform poorly in the context of wireless NoCs since they remain agnostic to these domain specific and intricate dependencies. Hence, the design of efficient medium access protocols has been identified as a key bottleneck for realizing the full potential of a wireless NoC multiprocessor [6, 12].

In this paper, we present NeuMAC, a unified approach that combines networking, architecture and deep learning to generate highly adaptive medium access protocols for a wireless network on chip architecture. NeuMAC leverages a reinforcement learning framework with deep neural networks to generate new MAC protocols that can learn traffic patterns and dynamically adapt the protocol to handle different applications running on the multi-core processor. Reinforcement Learning (RL) has proved to be a very powerful tool in AI for generating strategies and policies that can optimize for complex objectives [68, 81]. RL allows NeuMAC to make better decisions by learning from experience. In particular, many basic functions, like FFT, graph search, sorting, shortest path, etc., tend to repeatedly appear in many applications. Past work also shows that a number of unique periodic traffic patterns emerge in multiple different programs, and as the number of cores increases, the traffic patterns show increasingly predictable spatiotemporal correlations and dependencies [3, 4]. NeuMAC learns these statistics and correlations in the traffic patterns, to be able to both predict future traffic patterns based on traffic history and adapt its MAC protocol to best suit the predicted future traffic. Furthermore, RL enables NeuMAC to account for hard-to-model complex dependencies between execution time and delivery of packets. In particular, we carefully engineer the reward function in RL to optimize for execution time rather than to simply improve the latency and throughput of the network.

Indeed, RL has been leveraged for wireless MAC protocols in the context of heterogenous wireless networks [43, 101], sensor networks [41], and IoT networks [62]. However, bringing these benefits to wireless networks on chip faces a number of unique challenges. First, past work runs RL inference for every packet at each time step, which is not feasible for WNoCs since the time scale of operation in a multicore processor is in the order of nanoseconds. Hence, per time-slot inference would significantly delay every packet transmission. Second, due to compute resource constraints, it is also not feasible to run RL inference at every core of the wireless NoC. While the second challenge can be addressed using a centralized controller for the RL model, it would still incur significant communication overhead and latency to collect the states from the nodes (e.g. traffic injections or buffer occupancy) and to inform the nodes when to transmit.

NeuMAC addresses these challenges by designing a framework where the controller is trained to generate high-level MAC policies simply by listening to on-going transmissions on the wireless medium. This allows NeuMAC to eliminate any communication from the cores to the controllers. Moreover, to amortize the overhead of inference and policy updates, NeuMAC only updates the cores with a new MAC policy once every interval spanning many execution cycles (e.g. ten thousand cycles). We also train NeuMAC to learn policies that are highly adaptive and simple to update, to reduce communication overhead from the controller to cores.

Finally, NeuMAC also needs to operate within the strict timing and resource constraints of the multicore processor. Modern deep neural networks, however, are designed with up to a billion tunable parameters and operate on high dimensional input spaces [47, 80]. Consequently, they require large amounts of memory and computational resources, and also suffer high inference latencies (tens of milliseconds) [46, 63]. To address this, we design NeuMAC's RL framework such that the input and output of the neural network scale linearly with the number of cores. This ensures that NeuMAC is expressive enough to service the highly dynamic network traffic while at the same time operate under the limited memory and computational resources. Specifically, NeuMAC's neural network requires three orders of magnitude less parameters, and adds a small area overhead to the multicore processor. It also has an inference latency that is small enough to meet the strict timing constraints of the multicore during run-time as we show in detail in Appendix A.

We evaluate NeuMAC by integrating it with a cycle-level architectural simulator for CPU-GPU heterogeneous computing that faithfully models the intricacies of multi-core processors [87]. We augmented the simulator with an on-chip wireless network that accurately models transmissions, collision handling and packet losses. We test NeuMAC's performance on real applications chosen from diverse domains such as graph analytics, vision and numerical simulations. We compare NeuMAC against six baselines including wired NoC, standard CSMA, TDMA, optimal CSMA protocols [79], adaptive protocols [38, 65], and an optimal oracle. Our evaluation reveals the following:

- For a 64-core NoC, NeuMAC is capable of learning traffic patterns and adapting the medium access protocol at a granularity of $10\mu s$ to achieve a median gain of $2.56\times-9.18\times$ in packet latency and $1.3\times-17.3\times$ in network throughput over different wireless NoC baselines.
- NeuMAC's throughput and latency gains translate into an average of $10\%-47\%$ speedup in execution time over wireless NoC baselines which goes up to $1.37\times-3.74\times$ for certain applications. The results also show a $3.4\times$ speedup on average over a purely wired NoC.
- NeuMAC's gains in execution time are close to the upper bound that can be achieved by a wireless network with infinite capacity and zero latency.
- As the number of cores scale up to 1024 cores, NeuMAC's performance gain increases to 3 orders of magnitude lower latency and up to $64\times$ higher throughput over baseline protocols.
- NeuMAC is robust to lossy channels, and sees minimal degradation in performance with upto 10% packet losses. We also test NeuMAC's sensitivity to noise in the observed state and show almost no loss in performance.

Contributions: We make the following contributions:

- We introduce the first MAC protocol that can learn and adapt to the highly dynamic traffic at very fine granularity in a wireless NoC processor. The protocol also accounts for non-trivial dependencies between packet delivery and computation speedups by optimizing for execution time.
- We design a lightweight deep reinforcement learning framework that introduces little overhead to the multi-core processor and can operate within tight timing, power and area constraints of chip multicore processors.
- We extensively evaluate our design and demonstrate significant improvement in network performance and reduction in the overall execution time on the multicore processor.

2 Motivation and Insights

The wireless traffic patterns on a multicore processor have been shown to vary significantly across different applications. Even for a single application, the traffic can vary across different cores (spatially) and across different time intervals (temporally) [4, 6, 12, 38, 83].

Fig. 1(a) shows examples of traffic traces captured from a cycle-level architectural simulator for three different common benchmark applications on a 16-core multiprocessor. The x-axis shows the time in clock cycles, the y-axis shows the core ID, and the scatter points show the injection of traffic at each core. For clarity, we only show a portion of the execution spanning ten thousand cycles. Some applications, like *PageRank* shown in Fig. 1(a)(i), have almost constant traffic on all cores and can benefit from a contention-free protocol like TDMA. Other applications, like computing the *Shortest Path in a Graph* shown in Fig. 1(a)(ii), have very bursty traffic and can benefit from a contention-based protocol like

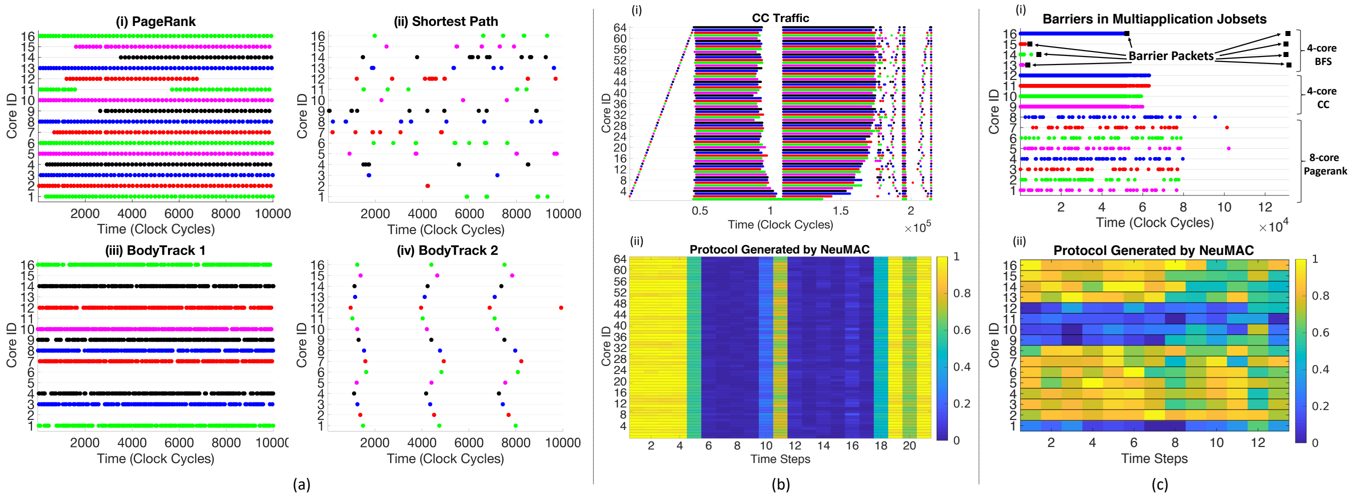


Figure 1: **Illustrative Examples: (a)** Traffic Pattern on a 16-core multiprocessor for different applications. The X-axis shows clock cycles, and the Y-axis corresponds to each of the 16 cores. The figures depict the scatter plots representing the packet injections into the buffer of each core. The different colors for packet injections are used for different cores. **(b)** NeuMAC can quickly adapt to fast changing traffic thus ensuring efficient network utilization throughout the application's execution. In the generated protocol, high probability values (closer to yellow in colormap) represent a CSMA-like protocol whereas low probability values (closer to blue) represent a TDMA-like protocol. **(c)** NeuMAC can learn and optimize for the intricate dependencies between the executions on different cores, and in turn optimize directly for end-to-end execution.

CSMA. Moreover, in most applications, the traffic pattern changes within the execution of the application. For example, Fig. 1(a)(iii)-(iv) show the traffic patterns at different times in the execution of *BodyTrack*, a computer vision application for tracking body pose. In the first time interval, since there is steady injection of packets into the network on the 10 active cores, a contention-free scheme will be optimal to minimize collisions, whereas in the second time interval, a CSMA-like based scheme for all 16 cores will perform better due to the sparse traffic injection. Next, we present concrete examples showcasing the range of protocols that NeuMAC can generate for different traffic patterns.

A. Adapting to Dynamic Traffic Patterns: To further appreciate the spatial and temporal changes across the execution of an entire application, we show the traffic trace for the application CC (Connected Components of a graph), running on a 64-core processor in Fig. 1(b)(i). Here we can see that the traffic varies significantly across the application's execution.

Fig. 1(b)(ii) presents the protocol generated by NeuMAC. At a very high level, NeuMAC's protocol is simple. Each core gets its own dedicated time slot where it can transmit with probability 1 if it has traffic. Additionally, core i can also transmit in time slots assigned to the other cores with some contention probability p_i. By setting these probability values p_i for each core, NeuMAC dictates the MAC protocol on the wireless NoC. The figure shows these contention probabilities p_i's for each core generated by NeuMAC. We present NeuMAC's protocol design in more detail in Section 4.3.

From Fig. 1(b)(ii), we can see that NeuMAC is able to adapt quickly to the changes in the traffic patterns, becoming more TDMA-like when the traffic is dense (contention probabilities p_i's are 0 and everyone transmits only in their assigned slot), and becoming more CSMA-like with sparse traffic (contention probabilities p_i's are high and cores can start transmitting in other's assigned time slots). In the case of CC, we can see that initially the traffic pattern is extremely sparse and structured such that a simple "Aloha" protocol would suffice. As a result, in the beginning the cores contend for the channel aggressively under NeuMAC's protocol. However, once the traffic pattern becomes more dense, NeuMAC adapts the protocol to be more TDMA-like, thus ensuring high network utilization. Finally, once the traffic pattern becomes less dense after $18*10^4$ cycles, the cores again start to contend for the channel with higher probability, thus emulating a CSMA-like protocol. Note that, while NeuMAC is able to quickly detect traffic changes from dense to sparse at time steps 11 and 18 (From Fig. 1(b)(ii)), it does not immediately increase contention probabilities for the cores. Instead the change is gradual, and this is because of the outstanding packets remaining in the buffers immediately after the phase with dense traffic injection. As a result, immediately switching the probabilities would lead to large number of collisions.

The above example demonstrates that NeuMAC is able to learn fine-grained highly dynamic MAC protocols that can quickly adapt to support different kinds of traffic patterns, while accounting for subtle characteristics of network operations such as buffer build-ups even though this information is not explicitly fed into NeuMAC's RL model. While there has been a lot of work on adaptive and optimal CSMA protocols [51, 73, 77, 102], these works are theoretical and make unrealistic assumptions. In particular, they optimize for long term throughput and assume that the protocol can reach a steady-state operation much faster than the variation in traffic patterns, which does not hold for wireless NoCs. As a result, these protocols perform poorly as we show in section 6.

B. Optimizing for Synchronization Primitives: Another challenge in designing efficient protocols stems from synchronization primitives. These primitives impose intricate dependencies between the execution of threads on different cores, leading to a non-trivial relationship between the delivery time of packets on the NoC and the progress of execution on each core. For example, in parallel computing it is common practice for software developers to use `barriers` for synchronization. These barriers are placed throughout the code of a multithreaded application in order to force each thread to stop at a certain point, blocking its execution until all participating threads catch up. Most standard libraries for parallel programming use barriers in many of its primitive routines in order to ensure the correctness of the program, such as OpenMP's `For` loop [23], or MPI's `Send/Recv` [45]. Therefore, there is complex but predictable structure in the traffic patterns caused by these synchronization primitives that can be exploited to improve parallel speedup and scalability of high performance applications. Hand tuning protocols to account for these dependencies is non-trivial. For example, the cores themselves do not explicitly know that they are involved in a barrier before they actually reach the barrier and execution halts. [28, 82]. Past work on designing MAC protocols mainly optimizes for throughput and latency, and is agnostic to such dependencies.

As a concrete example, consider the multiapplication jobset comprising of three concurrent applications, namely a 4-core BFS, a 4-core CC and a 8-core Pagerank, running on a 16-core multiprocessor as shown in Fig. 1(c)(i). In the traffic trace, one can observe two sets of barrier packets in the execution of BFS, denoted by black squares. The other two applications have no barriers in this portion of their executions. Here, note that core 16 has significantly more packets to transmit before arriving at its barrier, whereas core 13, 14 and 15 arrive at their barriers sooner. As a result, the execution on cores 13, 14 and 15 is blocked until core 16 clears its barrier, thus rendering the compute resources of these three cores useless as they idly wait for core 16. Additionally, at the same time core 16 also has to contend for the channel with traffic from CC, which itself has a lot of ongoing communication. Ideally, the MAC protocol in this case should prioritize traffic of the core that is falling behind, so that it arrives to the barrier and clears it as soon as possible, allowing the blocked cores to proceed execution and thus optimizing overall execution time. In Fig. 1(c)(ii), we can see that NeuMAC can learn to account and optimize for such dependencies. At the start, NeuMAC assigns high contention probabilities to cores 13 to 16 so that it can clear the barrier point at the earliest, while assigning low contention probabilities to cores 9 to 12. Once the barrier is cleared, NeuMAC increases the contention probabilities for the CC cores, so that it can transmit on the channel while the other applications go through low communication periods, thereby ensuring high network utilization.

Protocols like CSMA, TDMA and even adaptive protocols

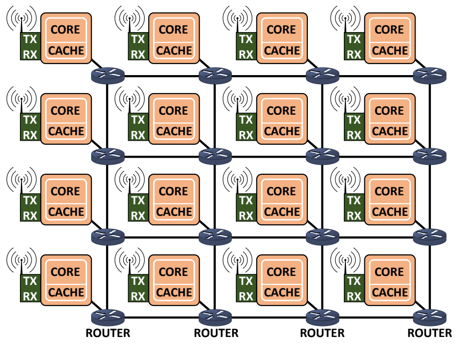


Figure 2: NoC Architecture with Wireless Links

cannot optimize for such situations as they would treat every packet in the network as equally important, thus sharing the channel equally between BFS and CC here. This would result in core 16 clearing its barrier much later, thus harming end-to-end execution time. However, since NeuMAC is trained to directly optimize the high-level objective of end-to-end execution time instead of network metrics like latency, it is able to learn to prioritize the packets of some cores over others. In this example, with NeuMAC's protocol, core 16 arrives at its barrier 2.4× faster as compared to CSMA, and 3.75× faster as compared to TDMA. This in turn leads to an overall improvement in execution time of 43% and 81% over CSMA and TDMA respectively.

3 Background

3.1 Wireless Network on Chip

Network-on-Chip (NoC) architectures have played a fundamental role in scaling the number of processing cores on a single chip which led to unprecedented parallelism and speedups in execution time [30, 75, 88, 94]. Prior to NoC, multicore processors used a shared bus architecture which had very poor scalability. As the core count increases, the power required to drive the bus grows quickly due to the increase in the capacitance of the bus wires [15]. The bus also starts to suffer from large latency [74]. As a result, shared buses become impractical for designs beyond 16 cores [59].

Unlike a shared bus, wired NoCs use packet-switched communication with every core connected to a router as shown in Fig. 2 [78]. As the packet moves from source to destination, it is buffered, decoded, processed, encoded, and retransmitted by each router along the multi-hop path. However, as we scale the number of cores, computation slows down due to the high communication latency and overhead of the network [10, 57, 97]. This problem is known as the "Coherency Wall" [58], where the execution on each core is faster than the NoC's ability to ensure that the memory caches of the cores are coherent. Hence, the speedup gained by parallelism and multithreading is outweighed by the network's communication cost for keeping the caches coherent [5, 8, 58].

Recent work proposes to augment NoC multicore processors with wireless links for communication between the cores [7, 9, 54, 65, 91]. Wireless links benefit chip multicore processors in two important aspects:[1]

- **Lower Latency:** Wireless enables every core to reach every other core in just a single hop. In contrast, in a purely wired NoC, a packet must go through multiple NoC routers, incur queuing, transmission, and processing delay at every hop which ends up taking multiple execution cycles [1]. Hence, as the number of cores increase, wireless can deliver packets with significantly lower latency and within the tight timing requirements of execution on the cores [1].
- **Broadcast:** Since wireless is a broadcast medium, transmitted packets are directly heard at all other cores which significantly simplifies the NoC's ability to ensure the coherency of the memory caches. In particular, any local changes in the memory cache of a core can instantaneously be replicated at all other cores through a single packet transmission [38]. In contrast, today's wired NoCs must send multiple parallel unicast/multicast transmissions to synchronize the caches, which leads to a large overhead that scales poorly as the number of cores increases [8, 50, 56].

Several wireless NoC transceivers and antennas have been built and shown to deliver 10 to 50-Gbps links while imposing modest overhead (0.4–5.6%) on the area and power consumption of a chip multiprocessor [31, 39, 93, 99, 100]. The wireless transceivers typically operate in the millimeter-wave and sub-THz spectrum which enables miniaturizing the antennas and avoids antenna coupling. Antennas are either planar integrated dipoles or vertical monopoles drilled through the silicon die [24, 86]. The wireless signals propagate through the enclosed chip packaging and attenuate by few tens of dBs [85, 86]. On-Off Keying (OOK) is the choice of modulation since it requires significantly lower powerand achieves a very low Bit Error Rate (BER) for on-chip wireless links [39, 60, 99]. We adopt the collision and packet loss handling protocols from past work [1, 38].

3.2 Deep Reinforcement Learning

We provide a brief primer on RL based on [84]. In RL, an *agent* interacts with an *environment*, and learns to generate a policy directly from experience as shown in Fig. 3. In our case, NeuMAC is the *agent*, the multiprocessor is the *environment*, and the generated MAC protocol is the policy.

• **Agent & Environment:** The *agent* starts with no apriori knowledge. Then, at each time step t, the agent observes the state s_t of the environment, and takes an action a_t. Following the action, the environment transitions to state s_{t+1}, and the agent receives a reward r_t. The state transitions and the rewards are stochastic and assumed to have the Markov property.

[1]Note that other technologies such as optical links have poor performance [2, 9, 37].

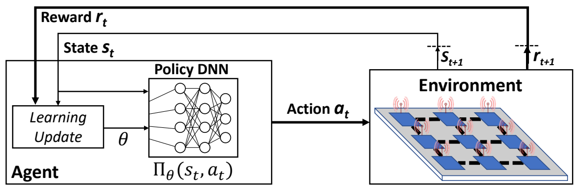


Figure 3: Deep Reinforcement Learning Framework.

During training, the *agent* gains experience by taking actions and observing the state transitions and rewards in response to these actions. The actions the *agent* takes aim to maximize an objection function known as the expected cumulative discounted reward: $\mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right]$, where $\gamma \in (0, 1]$ is the discount factor for future rewards.

• **Policy:** The action a_t picked by the *agent* is dictated by a *policy* π, where π represents a probability distribution over the space of actions and states : $\pi(s, a) \rightarrow [0,1]$. That is, $\pi(s, a)$ is the probability that action a is taken in state s by the agent following policy π. For most large-scale practical problems, the policy π is modeled with a Deep Neural Network (DNN), as they are very powerful function approximators. The DNN is parameterized by θ, which are the learnable parameters of the model, and we represent the policy as $\pi_\theta(s, a)$. θ is also referred to as the *policy parameters*.

• **Training:** The objective of training in RL is to learn the policy parameters θ so as to maximize the expected cumulative reward received from the environment. Towards this end, we focus on a class of RL algorithms called *policy gradient algorithms*, where the learning takes place by performing *gradient descent* on the policy parameters. In practice, the training methodology follows the *Monte Carlo method* where the agent samples multiple trajectories obtained by following the policy π_θ, and uses the empirically computed cumulative discounted reward as an unbiased estimator of the expected value. This empirical value is then used to update the policy parameters via the gradient descent step. The result is a known algorithm: REINFORCE which we use in this paper. For more details, we refer the reader to [84].

4 NeuMAC Design

4.1 Overview

NeuMAC consists of two components. (1) A standard NoC multicore processor with N cores where each core has been augmented with a wireless transceiver as shown in Fig. 2. (2) A NeuMAC agent that periodically generates new medium access policies based on the traffic patterns it sees on the wireless NoC. The agent is housed in a simple neural accelerator that resides on the same chip with a small area and power overhead (See Appendix A for hardware details).

Fig. 4 shows the working of NeuMAC. The NeuMAC agent is equipped with a wireless transceiver through which it can listen on the channel, and also send protocol updates to the

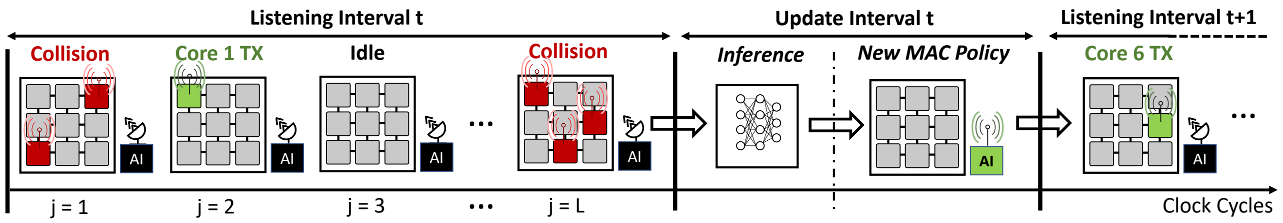


Figure 4: An Overview of NeuMAC's Protocol

cores. The NeuMAC agent listens on the wireless channel for a period called the "*Listening Interval*" where it collects traffic data about core transmissions, collisions and idle slots. It, then, feeds this data to a trained RL neural network that implicitly predicts the future traffic patterns and generates a new policy to be used as the medium access protocol during the next *Listening Interval*. NeuMAC updates the policies at the cores by sending an update message with the policy parameters. Each *Listening Interval* and *Update Interval* constitute a single step in the RL framework.

One point to note is that, although the cores share a common clock for their normal CPU operation[2], it is infeasible to coordinate medium access for each clock cycle through a shared centralized scheduler, since the exchange of control messages between the cores and the scheduler would itself incur latencies of multiple clock cycles. [6]

4.2 Design Challenges

The above design is governed by several strict timing and resource constraints of wireless NoC. In particular, it must address the below challenges while at the same time ensuring NeuMAC's ability to generate versatile and expressive medium access protocols to service the dynamic and fast-varying traffic patterns.

C1. *Centralized Agent:* Ideally, we would have wanted NeuMAC to adopt a distributed design where every core is equipped with its own NeuMAC agent that dictates its own MAC protocol. However, introducing a neural accelerator at every core would be prohibitively expensive in terms of area and power. Hence, NeuMAC is constrained to a centralized approach with a single agent.

C2. *Cores to Agent Communication Overhead:* To obtain an accurate view of traffic patterns, NeuMAC must obtain the packet injection rate and buffer occupancy across time at each core in the network. However, relaying this information from every core back to the centralized agent would result in huge communication overhead. Instead, NeuMAC leverages the broadcast nature of wireless networks to collect traffic patterns simply by listening for transmissions on the wireless medium. While the collected information is less expressive than the history of packet injection and buffer occupancy at each core, it retains sufficient information to allow NeuMAC to predict traffic patterns while at the same time completely eliminating communication overhead from the cores to the centralized agent.

C3. *Agent to Cores Communication Overhead:* One option is to have the agent tell each core whether to transmit or not at every CPU clock cycle. However, this would require running inference and relaying information to each core at every clock cycle which would lead to prohibitively large communication overhead. To address this, NeuMAC amortizes the communication overhead (*Update Interval*) from the agent to the cores by performing inference once every *Listening Interval* spanning thousands of clock cycles. In our implementation, we use an interval of $L = 10{,}000$ clock cycles ($10\mu s$) which is large enough to reduce the overhead to less than 6% and small enough to ensure that the traffic patterns remain stable and can be learned by the RL agent.

C4. *Complexity of the MAC Policy:* NeuMAC generates a policy that dictates the MAC protocol of each core for the following *Listening Interval*. Ideally, NeuMAC would generate a deterministic transmission schedule for every core to follow. Such a design is extremely expressive since it could allow NeuMAC to generate any possible schedule. However, such a design would require the RL deep neural network to output an action space with $N \times L$ dimensions where N is the number of cores and L is the number of clock cycles (e.g. 10,000). Such a neural network would be unsuitable for a resource-constraint setting like NoC. To address this, we carefully design a parameterized MAC policy that can support a flexible range of medium access protocols while ensuring that the neural network only needs to output a few parameters to dictate the desired policy.

C5. *Reward engineering:* The reward during training needs to be designed so as to guide NeuMAC towards the high-level objective. While most past work on learning link-layer and network-layer protocols only use network-level metrics such as throughput and latency for the reward signal, in our case we need to choose domain specific rewards so as to optimize for the end goal, which is application execution speedup on the multicore.

C6. *Low Footprint Neural Network:* NeuMAC's neural network must adhere to strict timing, power and area constraints of a chip multiprocessor. Thus, our design cannot simply adapt a known RL model as it would require large amounts of memory and computational resources, and would also suffer high inference latencies (tens of milliseconds) [46, 63].

[2] Unlike a distributed system of machines, a shared clock for a manycore system is feasible since all cores are housed on the same silicon die.

To address this, we design NeuMAC's RL framework such that the state space (input to the neural network) and action space (output) scale linearly with the number of cores. Our design ensures that NeuMAC is expressive enough while at the same time can operate under NoC's resource constraints.

4.3 NeuMAC's MAC Policy

As discussed above, the MAC policy that the agent dictates to the cores should have the following properties:

1. The policy should span a wide range of protocols, all the way from TDMA to CSMA.
2. It should be possible to describe the policy with few parameters to reduce the communication overhead and the output of the neural network.
3. It should allow for a simple neural network architecture to learn a mapping from observed traffic patterns to the most efficient MAC protocol.

In order to achieve these properties, we adopt a two-layer protocol design. The first layer consists of a deterministic underlying TDMA schedule, where each core is assigned a unique time slot for transmission in a round-robin fashion. For example, for time slots $j \in [1, \cdots, L]$, core i is assigned the slots $\{j \mid j \mod N = i\}$ where N is the number of cores. The second layer consists of a probabilistic transmission schedule like CSMA, where each core is assigned a contention probability. Specifically, during its assigned time slot, core i transmits on the channel with probability 1 if it has an outstanding packet in its buffer. During other cores' assigned time slots, core i can transmit with probability p_i. In the event of a collision, exponential backoff is implemented by halving p_i of the colliding cores similar to CSMA. On the other hand, if a transmission is successful, p_i is reset to it's initial value.

To generate this policy for an NoC with N cores, the RL neural network needs to output an action space that can be defined as $a_t = [a_{1,t}, a_{2,t}, \ldots, a_{N,t}]$ where $a_{i,t} \in [0, 1]$ represents the initial contention probability of core i during *"Listening Interval" t* (i.e., time step t in the RL framework). The contention probability of core i is then initialized as $p_i = a_{i,t}$. Different choices of a_t result in different protocols on the multicore. For instance, setting $a_{i,t} = 0$ for all i results in a simple TDMA protocol since every core only transmits on the channel during its assigned slot. On the other hand, $a_{i,t} = c > 0$ for all i mimics a CSMA-like protocol with varying degrees of aggressiveness on the channel. The pseudo code for NeuMAC's protocol is presented in Alg. 1.

The above formulation satisfies our design objectives. First, it enables NeuMAC to gracefully shift between a pure TDMA and a CSMA scheme, while supporting all intermediate protocols. The design also gives the flexibility to control each core individually, so that the NeuMAC can potentially increase contention probabilities for cores that observe high traffic intensity. Second, since the MAC protocol at core i is

Algorithm 1 NeuMAC Protocol

```
$L \leftarrow$ Number of Clock Cycles in Listening Interval
$[a_{1,t}, a_{2,t}, \ldots, a_{N,t}] \leftarrow$ Action space generated by RL agent at time step$t$
$[p_1, p_2, \ldots, p_N] \leftarrow [a_{1,t}, a_{2,t}, \ldots, a_{N,t}]$

At core $i$:
for $j \in \{1, \cdots, L\}$ do
    $Buffer_i(j) \leftarrow$ Outstanding packet in the buffer for core $i$
    if $Buffer_i(j) \neq \emptyset$ then
        if $j \mod N = i$ then                ▷ TDMA Slot Assigned to Core $i$
            Transmit with probability 1
        else
            Transmit with probability $p_i$
    if Transmission from Core $i$ collides then
        $p_i = p_i/2$
    else
        $p_i = a_{i,t}$
```

characterized by only one number (the contention probability $a_{i,t}$), there is very small communication overhead during the *Update Interval*, where the NeuMAC agent has to transmit a single broadcast packet with N numbers. Each core, receives the packet and extracts it own contention probability. Finally, the design keeps the action space constrained and linear in the number of cores which allows for a simple neural network that can be easily trained and is more likely to converge.

4.4 RL Formulation and Training

Given the above design, we now formalize the state space, reward, policy and training of NeuMAC's RL framework.

• **State Space Design:** The NeuMAC agent takes state information s_t as input and generates a MAC policy characterized by the action space a_t described above. The state information is generated purely by listening to ongoing transmissions on the channel. As described earlier, this allows us to eliminate all communication overhead from the cores to the RL agent. However, it only provides information about the activity on the channel rather than the traffic injection into the network. Moreover, in the event of a collision, NeuMAC cannot know which cores attempted to transmit. Despite these limitations, NeuMAC's state space retains enough information to infer traffic patterns. In particular, during each CPU cycle, NeuMAC will either detect an idle channel, a collision, or a successful transmission from some core i. We define our state at time step t, s_t, as an $(N+1) \times 1$ vector that keeps track of the number of successful transmissions from each core and the number of collisions observed during the cycles in the RL time step (*Listening Interval*). Specifically, the i^{th} element of s_t counts the number of successful packet transmissions by core i, and the $N+1^{th}$ element counts the number of collisions. The number of idle slots is implicitly encoded in the state since it is equal to $L - \sum_{i=1}^{N+1} s_{i,t}$ where L is the number of cycles in a *Listening Interval*. The state s_t is then used by the NeuMAC agent to generate the MAC protocol policy for the next time step.

• **Reward Engineering:** The reward signal is designed to guide the agent towards policies that optimize for the desired objective. Most past work that uses RL for learning networking protocols employs network-level metrics like throughput or latency as the reward signal. However, in our case, we need the reward signal to directly represent our end goal, which is to optimize for speedups in application execution time on the multicore. While network-level metrics like throughput are correlated to the execution time, they do not always capture the intricate dependencies between the execution on threads and packet delivery on the network. In Section 6, we see that there are instances where a protocol performs significantly worse in terms of average network throughput, but still has better end-to-end application execution time.

As a result, we design our reward signal to reflect our high level objective of minimizing application execution time. Specifically, for each time step t, the reward is set to $-L_t$ where L_t represents the number of clock cycles where the application was executing. Hence, for all but the last time step, the reward signal r_t is set to $-L$. For the last time step, reward is set to $-k$, where k is the number of clock cycles at which the application terminates execution. The intuition behind this choice for the reward signal is as follows. Recall that the objective of reinforcement learning is to maximize the cumulative reward, i.e. $-\sum_t L_t$. This is equivalent to minimizing $\sum_t L_t$, which ultimately means the application utilizing fewer CPU clock cycles for execution. While this choice of reward signal does correlate with improving network-level metrics such as packet latency and throughput, it is not the central objective and thus it is possible that sometimes the NeuMAC agent compromises on network performance for improvement in execution time. Note that in our formulation, we set the discount factor $\gamma = 1$.

• **Policy:** We represent our policy π as a deep neural network (also called policy network) which takes as input the state s_t, and maps it to a_t in the action space. Note that in our problem, the action space is continuous. In such cases it is common to discretize the continuous action space $a \in [0,1]^N$ similar to [52], and convert the problem into a classification problem where the agent now chooses which combination of a_i's to pick. However, an obvious issue with this approach is the curse of dimensionality. Even with 2 quantization levels for each a_i, the total number of discretized actions in $a \in [0,1]^N$ becomes 2^N. Thus the neural network architecture needs to have an output dimension of 2^N which becomes infeasible for our resource constrained environment.

Therefore, we avoid discretizing the action space and, instead, model the actions as following a Gaussian distribution with mean μ and variance σ. The deep learning model is now trained to output the parameters of this Gaussian distribution, as described in [84]. The NeuMAC agent picks the action for the next time step simply by sampling from the distribution $\mathcal{N}(\mu, \sigma)$. In NeuMAC, the policy network outputs N parameters μ_i corresponding to N distributions, one for each core i. The variance σ is set to 1 at the start of training to encourage exploration, and annealed down to 0.05 as NeuMAC's policy improves. Finally, during inference, the variance σ is set to 0.05, the action $a_{i,t}$ for core i is sampled from the corresponding distribution $\mathcal{N}(\mu_i, \sigma)$, and clipped to ensure that $a_{i,t} \in [0,1]$.

• **Training Algorithm:** We train our policy network end-to-end in an *episodic* setting. In each episode, an instance of an application is executed on the multicore, and the wireless network on chip follows the MAC protocol as dictated by the NeuMAC's policy network. The episode terminates when the application completes execution. In order to learn a policy that generalizes well, we train the network for multiple episodes with each episode observing a different application trace. For every episode, we run M separate Monte Carlo simulations to explore the probabilistic space of possible actions using the current policy, and use the resulting data to improve the policy for all applications. Specifically, we record the state, action, and reward information for all time steps of each episode. We then use this data to train our policy using the popular REINFORCE algorithm along with a baseline subtraction step, as described in [67].

4.5 Neural Network Architecture

Our network is composed of three fully connected layers with 128, 128 and 64 neurons respectively. The first two layers are followed by ReLU activation units, whereas the final layer is followed by a sigmoid unit to output the probability values a_i's between 0 and 1. During training, the weights use 16 bit floating points. Once trained, the learned weights are quantized to 8 bit fixed points for the inference stage. This is standard for run-time optimization in deep learning [53], and does not adversely affect performance.

The proposed fully connected network architecture here is simple and ties in very well with our design objectives. Recall that NeuMAC performs one inference step every 10,000 CPU clock cycles, and we require the inference step to add little overhead. The architecture here is composed of 32,000 learnable parameters, and at 8-bit quantization, it can be stored in a 32 KB on-chip SRAM cache to ensure fast memory accesses. Since inference latencies in most neural network architectures tend to be memory bound (including Fully connected and CNN architectures) [26,53], improving memory access latencies plays a big role in speeding up overall inference time. Further, the simple structure of a fully connected network allows for straightforward memory access patterns, since the inference step is a straightforward computation amounting to consecutive matrix multiplications. In Appendix A we provide energy-delay characterization of this architecture.

One point to note is that NeuMAC's deep RL agent is trained offline, and does not undergo any training during run-time since training is resource intensive. However, retraining can be triggered periodically depending on performance re-

Name	Description
BFS [13]	Breadth-first search
Bodytrack [20]	Tracking a body-pose through images
Canneal [20]	Compute optimal routing for gates on a chip
CC [13]	Compute connected components of a graph
Pagerank [13]	Compute pagerank for nodes in a graph
SSSP [13]	Single source shortest path
Volrend [96]	Rendering of 3D objects
StreamCluster [20]	Cluster streams of points
Community [13]	Compute modularity of a graph

Table 1: Summary of Applications

quirements and this retraining will be performed offline. The updated model parameters can then be migrated to the neural hardware accelerator by simply rewriting the SRAM memory blocks on the accelerator corresponding to the neural network's model parameters. This update can happen through the multicore's wireless NoC communication channel and won't add much overhead since our model is restricted to just 32,000 parameters, each of 8 bits.

5 Implementation

Evaluation Environment: We evaluate NeuMAC on a cycle-level execution-driven architectural simulator, Multi2sim [87]. Multi2sim is a popular end-to-end heterogenous system simulator tool used in the architecture community to test and validate new hardware designs with standard benchmarks. We evaluate NeuMAC for multicores with core count $n = 64$ at 22nm technology running at 1GHz. We use the same architecture parameters as [38]. We augment Multi2sim with an on-chip wireless network that accurately models transmissions, collision handling and packet losses.

While NeuMAC could be potentially trained directly using multi2sim, it is extremely slow and would result in prohibitively large training times. Therefore, for NeuMAC's training phase, we use a light-weight custom-built Wireless Network-on-Chip simulator along with traffic traces captured from Multi2sim. Our custom simulator models the data dependencies and synchronization primitives (such as locks and barriers) in the applications, so as to faithfully mimic the behavior of multi-threaded applications.

In order to evaluate NeuMAC's generalizability and effectiveness for a broad use case, we test NeuMAC on 9 different applications chosen from diverse domains such as graph analytics, vision, and numerical simulations (Summary in Table 1). Additionally, we also test with multi-application jobsets where different groups of cores are executing different multithreaded applications. While training is performed using our custom simulator, we evaluate NeuMAC using Multi2sim. We integrate Multi2sim with NeuMAC's trained RL agent, and our evaluations account for the RL agent's DNN inference latency and communication latency between the multicore and RL agent.

Training and Evaluation Details: For each application, we collect 500 different traces, each generated with different inputs to the applications in order to capture the variations between different runs. We evaluate NeuMAC using k-fold cross validation, where we train the model on 8 applications and test performance on the ninth application. Thus, we ensure that the NeuMAC agent is never explicitly trained on the application it is being evaluated on, and our results show that NeuMAC can generalize well to different applications. We train NeuMAC for a total of 4000 episodes, and for each episode we run $M = 16$ Monte Carlo simulations in parallel. The policy network is trained using ADAM optimizer [55] with a learning rate of 0.001.

6 Evaluation Results

6.1 Baselines

We compare with the following baselines:

(1) **CSMA with Exponential Backoff:** CSMA/CA protocol from 802.11 networks, with backoff window ranging from 1 to 1024. [1, 71] use CSMA MAC in the context of WNoCs.

(2) **TDMA:** Cores are allocated fixed slots for transmission in round-robin fashion. [5, 34] evaluate TDMA for WNoCs.

(3) **Switch-thresh:** [38, 65] propose a protocol that switches between a static CSMA and a static TDMA protocol based on per-core preset thresholds for channel activity and buffer occupancy. The optimal threshold values vary across applications and we choose values that are best in the average case.

(4) **Optimal CSMA Algorithm:** There is a large body of work that designs throughput optimal CSMA algorithms. However, most of these works are theoretical, and make simplifying assumptions like ignoring collisions or static traffic arrival rates, due to which they perform significantly worse than even regular CSMA protocols in practice. Among the optimal CSMA algorithms we tested, we found queue-based algorithms to perform best. We implement an extension of the popular Q-CSMA algorithm [79], where each node uses its buffer queue buildup to infer its transmission aggressiveness on the channel. While this algorithm is not truly distributed in nature, we ignore the global communication overheads in evaluations to favor the baseline performance.

(5) **Wired Baseline:** We also compare performance against a purely wired baseline, where all cache coherency traffic is serviced through the wired network-on-chip.

(6) **Infinite Capacity Channel:** We also compare NeuMAC's performance against an oracle with infinite channel capacity where the wireless medium can support multiple concurrent transmissions without suffering collisions, and every packet can be transmitted immediately without any channel contention delays. This baseline gives us an upper bound on how much improvement in end-to-end execution time is possible from improving the wireless NoC performance.

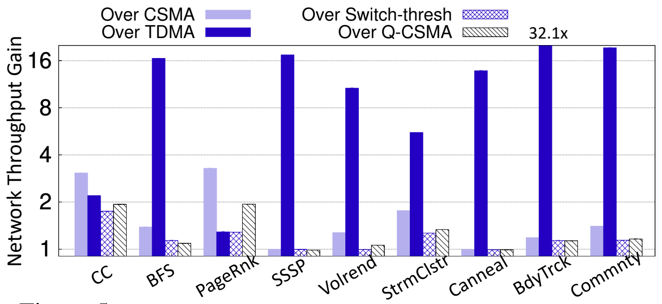


Figure 5: Gains in Wireless Network Throughput. (y axis in logscale)

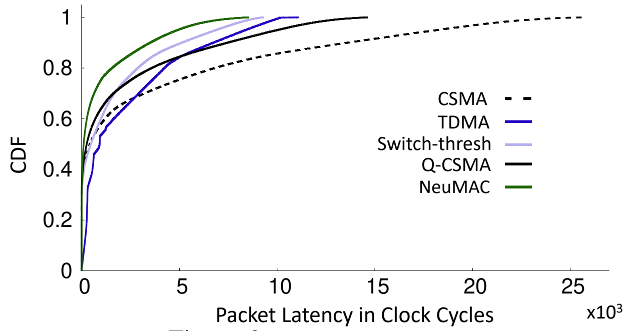


Figure 6: CDF of packet latency

Apps	CSMA	Switch-thresh	Q-CSMA	NeuMAC
CC	75.30%	55.58%	76.24%	8.72%
BFS	50.42%	28.28%	49.57%	3.81%
Pagernk	77.36%	11.26%	77.79%	2.19%
SSSP	11.08%	9.48%	9.44%	8.88%
Volrend	44.17%	7.93%	46.11%	2.49%
Strmclstr	62.57%	19.21%	62.69%	31.24%
Canneal	2.55%	2.87%	2.09%	2.04%
Bdytrck	30.5%	29.06%	29.8%	28.87%
Cmmnty	46.76%	32.02%	49.24%	5.8%

Table 2: % of Collisions

6.2 Quantitative Results

We first evaluate NeuMAC's performance against baselines on single application executions, followed by evaluations on the more realistic scenarios where multiple applications are running on the multicore. We also test NeuMAC's performance under lossy network conditions, and conclude by presenting scaling results where we demonstrate that NeuMAC's gains increase as the multicore scales to thousands of cores.

A. Single Application Wireless Network Performance: We begin by evaluating the wireless network performance against baselines along three metrics – (i) Wireless network throughput, (ii) Packet latency on the wireless network, and (iii) Number of collisions on the channel. We note that while NeuMAC is not explicitly trained to optimize for network metrics, their performance is correlated to faster execution times on the NoC.

(i) Network Throughput: In Fig. 5, we plot the gains in average network throughput achieved by NeuMAC against the baselines. Compared to CSMA and TDMA, NeuMAC achieves a mean improvement of 1.8× and 9.63× respectively across the benchmarks, and a maximum improvement of 3.3× and 32.1× respectively. TDMA has poor performance for average network throughput since cores have to wait for their turn to transmit even when the traffic is sparse, which leads to underutilization of channel.

Compared to Switch-thresh and Q-CSMA, NeuMAC achieves a mean improvement of 1.2× and 1.33×, and a maximum improvement of 1.7× and 1.9× respectively. While these protocols are improve over CSMA and TDMA, they still cannot react and adapt quickly enough to accommodate the fast changing traffic patterns on the multicore.

(ii) Packet Latency: In Fig. 6, we plot the CDF of packet latency due to queuing in the Wireless Network-on-Chip across all applications. It is interesting to note that while at the tail TDMA performs better than CSMA, in the median case TDMA performs significantly worse than CSMA. This is because the high packet latencies at the tail are due to dense traffic in the network which TDMA is better suited for, whereas at the median where traffic is less dense, TDMA leads to much higher packet latencies. NeuMAC, on the other hand, is able to adapt to all these different scenarios and provides an improvement in packet latency across all baselines. Over CSMA and TDMA, NeuMAC improves median packet latency by 4.11× and 9.18×, and improves 90th percentile latency by 3.89× and 1.92× respectively. Over Switch-thresh and Q-CSMA, the gains respectively are 4.66× and 2.56× at the median, and 1.47× and 2.13× at 90th percentile.

(iii) Collisions on Wireless Channel: In Table 2 we show % of collisions on the wireless channel across different benchmarks. We omit TDMA here since TDMA by design does not suffer from collisions. As observed, NeuMAC has significantly fewer collisions than the CSMA algorithms. Switch-thresh is the next best performing protocol, but NeuMAC in most cases still has fewer collisions.

B. Single Application End-to-End Execution Speedup:

(i) Speedups over Purely Wired Network-on-Chip: In Table 3, we show application speed-ups achieved by NeuMAC and the Infinite Capacity baseline respectively, over the purely wired NoC. NeuMAC can speed up benchmarks by up to 9.7× for StreamCluster and 6.53× for BFS, and on average provides a speedup of 3.42× across benchmarks. Additionally, we see that NeuMAC gets very close to the upper bound of the speedup value, achieving up to 99.5% of the maximum speedup possible in the case of BFS, and 98% of the maximum speedup possible on average. This result demonstrates that NeuMAC is able to fully exploit the potential offered by the wireless NoC.

(ii) Speedups over Baselines: Fig. 7 shows execution time gains of NeuMAC over the baselines on the wireless NoC. As can be observed, there is no one baseline protocol that performs well across all applications. While in applications

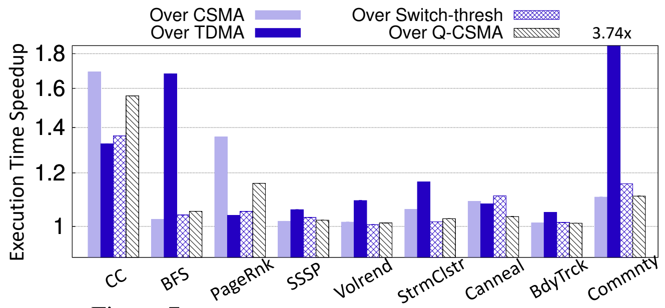


Figure 7: Execution Time Results (y axis in logscale)

Apps	NeuMAC	Inf. Cap. baseline	% Achieved
CC	1.96x	2.06x	95%
BFS	6.53x	6.56x	99.5%
Pagerank	1.07x	1.11x	96.4%
SSSP	2.24x	2.25x	99.5%
Volrend	1.32x	1.33x	99.2%
Strmclstr	9.70x	9.77x	99.28%
Canneal	1.14x	1.15x	99.13%
Bodytrack	1.37x	1.38x	99.3%
Community	3.77x	3.82x	98.6%

Table 3: Speedups over Purely Wired Network-on-Chip.

like Pagerank, TDMA performs the best, in other applications such as BFS it is significantly worse. NeuMAC, on the other hand, performs well across all benchmarks. In Table 4, we see that NeuMAC achieves a maximum of 69.18% speedup over CSMA for CC and 274.56% speedup over TDMA for Community, and compared to Switch-thresh and Q-CSMA, NeuMAC offers speedups up to 37.09%-55.94%.

C. Multi-Application Jobs: In Table. 5, we present execution time speedup results for multiapplication runs on the multicore. For each run, we randomly choose one application among the 9, and execute it using either 4, 16 or 32 threads. We choose a sufficient number of applications such that all 64 cores are utilized, and in total we test on 100 different multiapplication jobsets. Note that the NeuMAC agent was never explicitly trained on such multiapplication traffic traces. From Table. 5, we can see that NeuMAC's gains increase over the baselines compared to single benchmark experiments (Table. 4), and goes as high as 6.15× (515.04%) speedup over TDMA. These higher gains in multiapplication jobsets can be attributed to the more complex nature of packet dependencies between threads, which NeuMAC can exploit to further speed up execution time as illustrated in Section 2.

C. Lossy Networks: To evaluate NeuMAC's robustness to varying channel conditions, we conduct experiments in lossy network settings. We vary the packet loss rates in the wireless NoC from 0% up to 10%, and in the event of a loss, the packet is retransmitted. In Fig. 9, we compare the average application speedup achieved over the baselines as the loss rate increases. We observe that NeuMAC is able to generalize very well to varying channel conditions and loss rates, and

Speedups	CSMA	TDMA	Switch-thresh	Q-CSMA
Max	69.18%	274.56%	37.09%	55.94%
Min	1.26%	4.88%	0.63%	1.12%
Mean	18.21%	46.90%	9.73%	11.94%

Table 4: Summary of Execution Time Speedups by NeuMAC. The per-application speedups are shown in Fig. 7.

Speedups	CSMA	TDMA	Switch-thresh	Q-CSMA
Max	93.18%	515.04%	48.16%	26.78%
Min	13.3%	24.72%	4.41%	5.82%
Mean	33.93%	166.32%	19.97%	17.48%

Table 5: Summary of Execution Time Speedups by NeuMAC for Multiapplication runs

can maintain the same gains over the baselines throughout. Note that NeuMAC was never trained explicitly for lossy network settings. Despite this, it is able to generalize since it can implicitly infer the channel conditions from the channel activity like increased number of collisions.

We also test NeuMAC's sensitivity to errors in the observed state caused by packet losses at the NeuMAC agent's transceiver during the *"Listening Interval"*. We conduct experiments where we vary the packet loss rate from 0% to 2% in order to introduce noise in the observed state. We find that even under 2% loss rate, NeuMAC's suffers a median performance degradation of only 0.85% across all benchmarks compared to its performance with perfect state information.

D. Scaling Trends: We believe that a learning based approach like NeuMAC can greatly benefit the wireless NoC performance as the number of cores scale to thousands of cores. To demonstrate this we show the gains that NeuMAC achieves over baseline protocols for different metrics as the cores vary from 4 to 1024 in Fig. 8. Since multi2sim and other architectural simulators cannot scale beyond a hundred cores, we evaluate these results in our custom simulator by training a separate NeuMAC model for each core count. From Fig. 8, we can see that NeuMAC's gains over the baselines scale favorably with the number of cores. This is because NeuMAC is able to generate fine-grained MAC protocols by controlling the actions of each core individually, and thus can generate highly optimized protocols that improve substantially upon the baselines at high core counts.

7 Related Work

A. Wireless Network-on-Chip Protocols: The majority of past networking research on wireless NoC does not leverage the broadcast nature of wireless to enable instantaneous cache synchronization and instead focuses on using wireless only between far apart cores to reduce the latency. These complementary works focus on problems related to optimizing network topology [32,35,105], packet routing [61,90,106], flow control [18,42] and improving the reliability of the PHY layer for far apart cores [76,85,86]. However, such designs have limited gains over wired NoCs [4]. More recent work in architecture research exploits the broadcast nature of wireless to

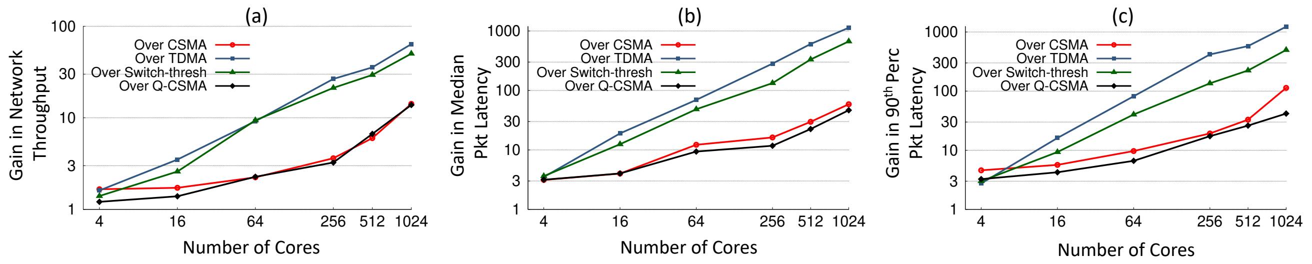


Figure 8: Scaling Trends in NeuMAC's Gains for (a) Wireless Network Throughput (b) Median Packet Latency and (c) 90^{th} Percentile Packet Latency

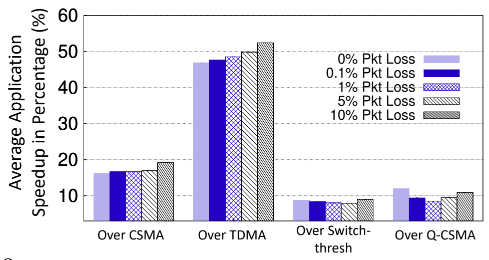


Figure 9: Effect of Packet losses on NeuMAC's application speedup performance compared to Baselines.

boost the performance of wireless enable NoCs [34,38,65,71]. These systems either use *contention-free* mechanisms such as token passing [34] or *contention-based* mechanisms such as carrier sense with exponential backoff [29,71]. The closest to our work are [38,65] which attempt to adapt to traffic patterns by switching between a CSMA or a token passing protocol based on a preset threshold. However, hand tuning the threshold values is a challenging task and does not provide the flexibility and expressibility of NeuMAC to support complex and highly variable traffic patterns.

B. Network-on-Chip Technologies: Past work on wired NoCs proposes the use of deep learning and RL to learn efficient packet routing protocols [98], learn memory access patterns to reduce cache misses [103], and reduce static and dynamic power consumption on an NoC [36]. To the best of our knowledge, ours is the first work that attempts to exploit deep reinforcement learning techniques to generate medium access protocols for Wireless NoCs.

C. Deep Learning in Wireless Networks: Deep RL has recently been applied in wireless networks to optimize duty cycling in sensor networks [64], resource allocation in cellular networks [22,27], dynamic spectrum access [72,92], rate adaptation in CSMA networks [69],and control policies at the PHY layer [52]. [104] provides an extensive survey of deep learning in wireless networks. The closest to our work are [14,16,19,101] which use reinforcement learning to modify the backoff parameters in CSMA or decide whether to transmit or not for every packet at every time step. However, such designs are not applicable in the context of wireless NoCs owing to the unique set of constraints imposed by the NoC, such as the much smaller time-scale of operation rendering neural network inference per transmission slot infeasible, the limited SRAM memory to store model parameters and the enormous action space to explore. These constraints require significant redesign to NeuMAC's deep RL framework where it has to now generate high-level, versatile and adaptable protocols that can be deployed for thousands of clock cycles, and generating such protocols cannot be reduced to a simple classification task per transmission-slot (e.g. transmit or not).

8 Limitations and Discussion

Some points are worth noting: First, given the enormous costs and engineering efforts involved in prototyping a full chip with integrated processors, memory, and NoC, it is outside the scope of this work to implement NeuMAC in hardware. As a result, we evaluate NeuMAC on a full-system cycle-accurate architectural simulator, as is the norm among computer architecture researchers. These full-system simulators exhaustively model all components of a CPU and also ensure that all timing dependencies are simulated accurately [87]. As a result, the trends and insights obtained from such architectural simulations often carry over to full fledged prototypes. Moreover, the wireless channel in this WNoC application domain is in fact very stable as opposed to WLAN channels which are extremely dynamic. This is because the multicore is isolated in a chip package, and the wireless channel can be precisely measured and characterized, thus allowing compensation for multipath fading and other artifacts. As a result, the wireless BER in these environments can be as low as 10^{-16} [33], making such a simulation based evaluation representative.

Second, in parallel programming for multicore processors, programmers today try hard to avoid broadcast transmissions as the overhead of running the cache coherency protocol is high. With wireless NoC, the overhead of broadcast traffic is now limited which opens the door to rewriting applications in a manner that embraces broadcast, and can in turn benefit even more from an adaptive protocol like NeuMAC.

Lastly, in this paper we focus on the MAC layer since it is considered a roadblock to realize the full potential of wireless NoCs. However, studying the challenges and opportunities at the other layers such as PHY remains exciting and promising avenue which we leave for future work.

Acknowledgements: We thank the shepherd and anonymous reviewers for their feedback and comments. We also thank Ameya Patil for helping guide the hardware accelerator design and characterization. The work is funded in part by NSF Award 1750725.

References

[1] S. Abadal, A. Cabellos-Aparicio, E. Alarcon, and J. Torrellas. Wisync: An architecture for fast synchronization through on-chip wireless communication. *ACM SIGOPS Operating Systems Review*, 50(2):3–17, 2016.

[2] S. Abadal, M. Iannazzo, M. Nemirovsky, A. Cabellos-Aparicio, H. Lee, and E. Alarcón. On the area and energy scalability of wireless network-on-chip: A model-based benchmarked design space exploration. *IEEE/ACM Transactions on Networking (TON)*, 23(5):1501–1513, 2015.

[3] S. Abadal, R. Martínez, J. Solé-Pareta, E. Alarcón, and A. Cabellos-Aparicio. Characterization and modeling of multicast communication in cache-coherent manycore processors. In *Computers & Electrical Engineering*, 2016.

[4] S. Abadal, A. Mestres, R. Martínez, E. Alarcon, and A. Cabellos-Aparicio. Multicast on-chip traffic analysis targeting manycore noc design. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 370–378. IEEE, 2015.

[5] S. Abadal, A. Mestres, M. Nemirovsky, H. Lee, A. González, E. Alarcón, and A. Cabellos-Aparicio. Scalability of broadcast performance in wireless network-on-chip. *IEEE Transactions on Parallel and Distributed Systems*, 27(12):3631–3645, 2016.

[6] S. Abadal, A. Mestres, J. Torrellas, E. Alarcón, and A. Cabellos-Aparicio. Medium access control in wireless network-on-chip: a context analysis. *IEEE Communications Magazine*, 56(6):172–178, 2018.

[7] S. Abadal, M. Nemirovsky, E. Alarcón, and A. Cabellos-Aparicio. Networking challenges and prospective impact of broadcast-oriented wireless networks-on-chip. In *Proceedings of the 9th International Symposium on Networks-on-Chip*, page 12. ACM, 2015.

[8] S. Abadal, B. Sheinman, O. Katz, O. Markish, D. Elad, Y. Fournier, D. Roca, M. Hanzich, G. Houzeaux, M. Nemirovsky, et al. Broadcast-enabled massive multicore architectures: A wireless rf approach. *IEEE micro*, 35(5):52–61, 2015.

[9] S. Abadal, J. Torrellas, E. Alarcón, and A. Cabellos-Aparicio. Orthonoc: a broadcast-oriented dual-plane wireless network-on-chip architecture. *IEEE transactions on parallel and distributed systems*, 29(3):628–641, 2018.

[10] S. Abadal, *et al.* OrthoNoC: A Broadcast-Oriented Dual-Plane Wireless Network-on-Chip Architecture. *IEEE Trans. Parallel Distrib. Syst.*, 2018.

[11] N. Abeyratne, R. Das, Q. Li, K. Sewell, B. Giridhar, R. G. Dreslinski, D. Blaauw, and T. Mudge. Scaling towards kilo-core processors with asymmetric high-radix topologies. In *Proceedings of the HPCA-19*, pages 496–507, 2013.

[12] A. B. Achballah, S. B. Othman, and S. B. Saoud. Problems and challenges of emerging technology networks- on- chip: A review. *Microprocessors and Microsystems*, 53:1–20, 2017.

[13] M. Ahmad, F. Hijaz, Q. Shi, and O. Khan. Crono: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores. In *2015 IEEE International Symposium on Workload Characterization*, pages 44–55. IEEE, 2015.

[14] R. Ali, N. Shahin, Y. B. Zikria, B.-S. Kim, and S. W. Kim. Deep reinforcement learning paradigm for performance optimization of channel observation–based mac protocols in dense wlans. *IEEE Access*, 7:3500–3511, 2018.

[15] Altera. An alternative to bus-based interconnects for large-scale design. In *White Paper*, 2008.

[16] S. Amuru, Y. Xiao, M. van der Schaar, and R. M. Buehrer. To send or not to send-learning mac contention. In *2015 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6. IEEE, 2015.

[17] M. Baharloo, A. Khonsari, P. Shiri, I. Namdari, and D. Rahmati. High-average and guaranteed performance for wireless networks-on-chip architectures. In *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 226–231. IEEE, 2018.

[18] J. H. Bahn and N. Bagherzadeh. Efficient parallel buffer structure and its management scheme for a robust network-on-chip (noc) architecture. In *Computer Society of Iran Computer Conference*, pages 98–105. Springer, 2008.

[19] H. Bayat-Yeganeh, V. Shah-Mansouri, and H. Kebriaei. A multi-state q-learning based csma mac protocol for wireless networks. *Wireless Networks*, 24(4):1251–1264, 2018.

[20] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81. ACM, 2008.

[21] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, B. Liu, A. Tran, E. Adeagbo, and B. Baas. A 5.8 pj/op 115 billion ops/sec, to 1.78 trillion ops/sec 32nm 1000-processor array. In *2016 IEEE Symposium on VLSI Circuits (VLSI-Circuits)*, pages 1–2. IEEE, 2016.

[22] U. Challita, L. Dong, and W. Saad. Deep learning for proactive resource allocation in lte-u networks. In *European wireless technology conference*, 2017.

[23] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[24] H. M. Cheema and A. Shamim. The last barrier: On-chip antennas. *IEEE Microw. Mag.*, 14(1):79–91, 2013.

[25] Y.-h. Chen, J. Emer, and V. Sze. Eyeriss v2: A Flexible and High-Performance Accelerator for Emerging Deep Neural Networks. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(2):292–308, 2019.

[26] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE journal of solid-state circuits*, 52(1):127–138, 2016.

[27] S. Chinchali, P. Hu, T. Chu, M. Sharma, M. Bansal, R. Misra, M. Pavone, and S. Katti. Cellular network traffic scheduling with deep reinforcement learning. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[28] D. Culler, J. P. Singh, and A. Gupta. *Parallel computer architecture: a hardware/software approach*. Gulf Professional Publishing, 1999.

[29] P. Dai, J. Chen, Y. Zhao, and Y.-H. Lai. A study of a wire–wireless hybrid noc architecture with an energy-proportional multicast scheme for energy efficiency. *Computers and Electrical Engineering*, 45:402–416, 2015.

[30] G. De Micheli and L. Benini. Networks on chips: 15 years later. *Computer*, (5):10–11, 2017.

[31] S. Deb, K. Chang, X. Yu, S. P. Sah, M. Cosic, A. Ganguly, P. P. Pande, B. Belzer, and D. Heo. Design of an energy-efficient cmos-compatible noc architecture with millimeter-wave wireless interconnects. *IEEE Transactions on Computers*, 62(12):2382–2396, 2012.

[32] S. Deb, A. Ganguly, K. Chang, P. Pande, B. Beizer, and D. Heo. Enhancing performance of network-on-chip architectures with millimeter-wave wireless interconnects. In *ASAP 2010-21st IEEE International Conference on Application-specific Systems, Architectures and Processors*, pages 73–80. IEEE, 2010.

[33] S. Deb, A. Ganguly, P. P. Pande, B. Belzer, and D. Heo. Wireless noc as interconnection backbone for multicore chips: Promises and challenges. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2(2):228–239, 2012.

[34] D. DiTomaso, A. Kodi, S. Kaya, and D. Matolak. iWISE: Inter-router wireless scalable express channels for network-on-chips (NoCs) architecture. In *2011 IEEE 19th Annual Symposium on High Performance Interconnects*, pages 11–18. IEEE, 2011.

[35] D. DiTomaso, A. Kodi, D. Matolak, S. Kaya, S. Laha, and W. Rayess. A-winoc: Adaptive wireless network-on-chip architecture for chip multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 26(12):3289–3302, 2014.

[36] D. DiTomaso, A. Sikder, A. Kodi, and A. Louri. Machine learning enabled power-aware network-on-chip design. In *Proceedings of the Conference on Design, Automation & Test in Europe*, pages 1354–1359. European Design and Automation Association, 2017.

[37] R. K. Dokania and A. B. Apsel. Analysis of challenges for on-chip optical interconnects. In *Proceedings of the 19th ACM Great Lakes symposium on VLSI*, pages 275–280. ACM, 2009.

[38] V. Fernando, A. Franques, S. Abadal, S. Misailovic, and J. Torrellas. Replica: A Wireless Manycore for Communication-Intensive and Approximate Data. In *ASPLOS*, 2019.

[39] D. Fritsche, *et al.* A Low-Power SiGe BiCMOS 190-GHz Transceiver Chipset With Demonstrated Data Rates up to 50 Gbit/s Using On-Chip Antennas. *IEEE Trans. Microw. Theory Techn.*, 65(9):3312–3323, 2017.

[40] S. H. Gade, S. S. Rout, M. Sinha, H. K. Mondal, W. Singh, and S. Deb. A utilization aware robust channel access mechanism for wireless nocs. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2018.

[41] S. Galzarano, A. Liotta, and G. Fortino. Ql-mac: A q-learning based mac for wireless sensor networks. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 267–275. Springer, 2013.

[42] A. Ganguly, K. Chang, S. Deb, P. P. Pande, B. Belzer, and C. Teuscher. Scalable hybrid wireless network-on-chip architectures for multicore systems. *IEEE Transactions on Computers*, 60(10):1485–1502, 2011.

[43] A. Gomes, D. F. Macedo, and L. F. Vieira. Automatic mac protocol selection in wireless networks based on reinforcement learning. *Computer Communications*, 149:312–323, 2020.

[44] S. K. Gonugondla, B. Shim, and N. R. Shanbhag. Perfect error compensation via algorithmic error cancellation. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 966–970. IEEE, 2016.

[45] W. Gropp, W. D. Gropp, E. Lusk, A. D. F. E. E. Lusk, and A. Skjellum. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.

[46] J. Hanhirova, T. Kämäräinen, S. Seppälä, M. Siekkinen, V. Hirvisalo, and A. Ylä-Jääski. Latency and throughput characterization of convolutional neural networks for mobile computer vision. In *Proceedings of the 9th ACM Multimedia Systems Conference*, pages 204–215, 2018.

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

[48] M. Horowitz. 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14. IEEE, 2014.

[49] Jack Clark. Intel: Why a 1,000-core chip is feasible, Press Release, 2010.

[50] N. E. Jerger, L.-S. Peh, and M. Lipasti. Virtual circuit tree multicasting: A case for on-chip hardware multicast support. In *2008 International Symposium on Computer Architecture*, pages 229–240. IEEE, 2008.

[51] L. Jiang and J. Walrand. A distributed csma algorithm for throughput and utility maximization in wireless networks. *IEEE/ACM Transactions on Networking*, 18(3):960–972, 2009.

[52] S. Joseph, R. Misra, and S. Katti. Towards Self-Driving Radios: Physical-Layer Control using Deep Reinforcement Learning. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*, pages 69–74. ACM, 2019.

[53] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 1–12, 2017.

[54] A. Karkar, T. Mak, K.-F. Tong, and A. Yakovlev. A survey of emerging interconnects for on-chip efficient multicast and broadcast in many-cores. *IEEE Circuits and Systems Magazine*, 16(1):58–72, 2016.

[55] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[56] T. Krishna, L.-S. Peh, B. M. Beckmann, and S. K. Reinhardt. Towards the ideal on-chip fabric for 1-to-many and many-to-1 communication. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 71–82. ACM, 2011.

[57] T. Krishna, *et al.* Towards the ideal on-chip fabric for 1-to-many and many-to-1 communication. In *Proceedings of the MICRO-44*, 2011.

[58] R. Kumar, T. Mattson, G. Pokam, and R. V. D. Wijngaart. The case for message passing on many-core chips. In *Multiprocessor System-on-Chip*, pages 115–123. Springer, 2011.

[59] R. Kumar, V. Zyuban, and D. M. Tullsen. Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 408–419. IEEE, 2005.

[60] S. Laha, *et al.* A New Frontier in Ultralow Power Wireless Links: Network-on-Chip and Chip-to-Chip Interconnects. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, 34(2):186–198, 2015.

[61] S.-B. Lee, S.-W. Tam, I. Pefkianakis, S. Lu, M. F. Chang, C. Guo, G. Reinman, C. Peng, M. Naik, L. Zhang, et al. A scalable micro wireless interconnect structure for CMPs. In *Proceedings of the 15th annual international conference on Mobile computing and networking*, pages 217–228. ACM, 2009.

[62] T. Lee, O. Jo, and K. Shin. Corl: Collaborative reinforcement learning-based mac protocol for iot networks. *Electronics*, 9(1):143, 2020.

[63] Y. Liu, Y. Wang, R. Yu, M. Li, V. Sharma, and Y. Wang. Optimizing {CNN} model inference on cpus. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 1025–1040, 2019.

[64] Z. Liu and I. Elhanany. Rl-mac: a reinforcement learning based mac protocol for wireless sensor networks. *International Journal of Sensor Networks*, 1(3-4):117–124, 2006.

[65] N. Mansoor and A. Ganguly. Reconfigurable wireless network-on-chip with a dynamic medium access mechanism. In *Proceedings of the 9th International Symposium on Networks-on-Chip*, page 13. ACM, 2015.

[66] N. Mansoor, S. Shamim, and A. Ganguly. A Demand-Aware Predictive Dynamic Bandwidth Allocation Mechanism for Wireless Network-on-Chip. In *Proceedings of the SLIP '16*, 2016.

[67] H. Mao, M. Alizadeh, I. Menache, and S. Kandula. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pages 50–56. ACM, 2016.

[68] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh. Learning scheduling algorithms for data processing clusters. *arXiv preprint arXiv:1810.01963*, 2018.

[69] N. Mastronarde, J. Modares, C. Wu, and J. Chakareski. Reinforcement learning for energy-efficient delay-sensitive csma/ca scheduling. In *2016 IEEE Global Communications Conference (GLOBECOM)*, pages 1–7. IEEE, 2016.

[70] D. W. Matolak, A. Kodi, S. Kaya, D. DiTomaso, S. Laha, and W. Rayess. Wireless networks-on-chips: architecture, wireless channel, and devices. *IEEE Wireless Communications*, 19(5):58–65, 2012.

[71] A. Mestres, S. Abadal, J. Torrellas, E. Alarcón, and A. Cabellos-Aparicio. A mac protocol for reliable broadcast communications in wireless network-on-chip. In *Proceedings of the 9th International Workshop on Network on Chip Architectures*, pages 21–26. ACM, 2016.

[72] O. Naparstek and K. Cohen. Deep multi-user reinforcement learning for distributed dynamic spectrum access. *IEEE Transactions on Wireless Communications*, 18(1):310–323, 2018.

[73] J. Ni, B. Tan, and R. Srikant. Q-csma: Queue-length-based csma/ca algorithms for achieving maximum throughput and low delay in wireless networks. *IEEE/ACM Transactions on Networking*, 20(3):825–836, 2011.

[74] J. Oh, M. Prvulovic, and A. Zajic. Tlsync: support for multiple fast barriers using on-chip transmission lines. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 105–115. IEEE, 2011.

[75] S. Pasricha and N. Dutt. *On-chip communication architectures: system on chip interconnect*. Morgan Kaufmann, 2010.

[76] M. Rahaman and M. Chowdhury. Improved bit error rate performance in intra-chip rf/wireless interconnect systems. In *Proc. ACM/IEEE Great Lake Symp. VLSI*, 2008.

[77] S. Rajagopalan, D. Shah, and J. Shin. Network adiabatic theorem: an efficient randomized protocol for contention resolution. In *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, pages 133–144, 2009.

[78] D. Sánchez, *et al.* An Analysis of On-Chip Interconnection Networks for Large-Scale Chip Multiprocessors. *ACM T. Archit. Code Op.*, 7(1), 2010.

[79] D. Shah, J. Shin, et al. Randomized scheduling algorithm for queueing networks. *The Annals of Applied Probability*, 22(1):128–171, 2012.

[80] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[81] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

[82] Y. Solihin. *Fundamentals of parallel multicore architecture*. CRC Press, 2015.

[83] V. Soteriou, H. Wang, and L. Peh. A Statistical Traffic Model for On-Chip Interconnection Networks. In *Proceedings of MASCOTS '06*, 2006.

[84] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[85] X. Timoneda, S. Abadal, A. Cabellos-Aparicio, D. Manessis, J. Zhou, A. Franques, J. Torrellas, and E. Alarcón. Millimeter-wave propagation within a computer chip package. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2018.

[86] X. Timoneda, S. Abadal, A. Franques, D. Manessis, J. Zhou, J. Torrellas, E. Alarcón, and A. Cabellos-Aparicio. Engineer the channel and adapt to it: Enabling wireless intra-chip communication. *arXiv preprint arXiv:1901.04291*, 2018.

[87] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli. Multi2sim: a simulation framework for cpu-gpu computing. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 335–344. IEEE, 2012.

[88] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar. An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS. *IEEE Journal of Solid-State Circuits*, 43(1):29–41, 2008.

[89] V. Vijayakumaran, M. P. Yuvaraj, N. Mansoor, N. Nerurkar, A. Ganguly, and A. Kwasinski. Cdma enabled wireless network-on-chip. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 10(4):28, 2014.

[90] C. Wang, W.-H. Hu, and N. Bagherzadeh. A wireless network-on-chip design for multicore platforms. In *2011 19th International Euromicro conference on parallel, distributed and network-based processing*, pages 409–416. IEEE, 2011.

[91] S. Wang and T. Jin. Wireless network-on-chip: A survey. *The Journal of Engineering*, 2014(3):98–104, 2014.

[92] S. Wang, H. Liu, P. H. Gomes, and B. Krishnamachari. Deep reinforcement learning for dynamic multichannel access in wireless networks. *IEEE Transactions on Cognitive Communications and Networking*, 4(2):257–265, 2018.

[93] N. Weissman and E. Socher. 9mw 6gbps bi-directional 85–90ghz transceiver in 65nm cmos. In *2014 9th European Microwave Integrated Circuit Conference*, pages 25–28. IEEE, 2014.

[94] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. Brown III, and A. Agarwal. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27(5):15–31, 2007.

[95] H.-S. P. Wong and S. Salahuddin. Memory leads the way to better computing. *Nature nanotechnology*, 10(3):191, 2015.

[96] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. *ACM SIGARCH computer architecture news*, 23(2):24–36, 1995.

[97] X. Xiang, *et al.* A model for application slowdown estimation in on-chip networks and its use for improving system fairness and performance. In *ICCD '16*, 2016.

[98] J. Yin, Y. Eckert, S. Che, M. Oskin, and G. H. Loh. Toward more efficient noc arbitration: A deep reinforcement learning approach. In *Proc. IEEE 1st Int. Workshop AI-assisted Des. Architecture*, 2018.

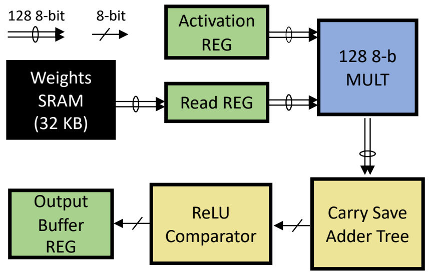


Figure 10: Illustrative Block Diagram of hardware macro employed for overhead characterization of NeuMAC's deep network

[99] X. Yu, J. Baylon, P. Wettin, D. Heo, P. P. Pande, and S. Mirabbasi. Architecture and design of multichannel millimeter-wave wireless noc. *IEEE Design & Test*, 31(6):19–28, 2014.

[100] X. Yu, H. Rashtian, S. Mirabbasi, P. P. Pande, and D. Heo. An 18.7-gb/s 60-ghz ook demodulator in 65-nm cmos for wireless network-on-chip. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 62(3):799–806, 2015.

[101] Y. Yu, T. Wang, and S. C. Liew. Deep-reinforcement learning multiple access for heterogeneous wireless networks. *IEEE Journal on Selected Areas in Communications*, 37(6):1277–1290, 2019.

[102] S.-Y. Yun, Y. Yi, J. Shin, and D. Y. Eun. Optimal CSMA: A survey. In *ICCS*, pages 199–204, 2012.

[103] Y. Zeng and X. Guo. Long short term memory based hardware prefetcher: A case study. In *Proceedings of the International Symposium on Memory Systems*, pages 305–311. ACM, 2017.

[104] C. Zhang, P. Patras, and H. Haddadi. Deep learning in mobile and wireless networking: A survey. *IEEE Communications Surveys & Tutorials*, 2019.

[105] D. Zhao and Y. Wang. Sd-mac: Design and synthesis of a hardware-efficient collision-free qos-aware mac protocol for wireless network-on-chip. *IEEE Transactions on Computers*, 57(9):1230–1245, 2008.

[106] D. Zhao, Y. Wang, J. Li, and T. Kikkawa. Design of multi-channel wireless noc to improve on-chip communication capacity. In *Proceedings of the Fifth ACM/IEEE International Symposium on Networks-on-Chip*, pages 177–184. IEEE, 2011.

A Energy and Latency Overhead Characterization

It is widely acknowledged that deep learning inference has high latency and energy overheads. However, since NeuMAC needs to optimize the performance of a multicore CPU, it needs to operate at very small time scales. As a result, it is imperative that NeuMAC's inference step be efficient in time and energy. In this appendix, we characterize the overheads of running inference on NeuMAC's Deep RL agent.

Towards this end, we design an illustrative hardware macro for NeuMAC's neural accelerator (shown in Fig. 10). The trained quantized weights of NeuMAC's network are stored in the 32 KB on-chip SRAM. The primary compute elements in the macro are the (i) 128 element 8-bit multiplier, that can perform 128 parallel multiplications of 8-bit numbers, (ii) followed by a 7-layer carry save adder tree, which can add up to 128 8-bit numbers. Thus, the multiplier block and adder tree block together can implement either one 128 dimensional dot product, or two 64 dimensional dot products in a one iteration. The ReLU non-linear activation is implemented using comparators, which finally writes the result into an output buffer. It is important to note that this hardware macro is significantly simpler than a full scale neural network accelerator, such as [53].

Next, we elaborate on the pipeline for computing one inference step on NeuMAC's RL agent. Note that computing the value of one element in the first hidden layer of NeuMAC's neural network requires one 64 dimensional dot product[3]. Therefore, computing the values of all elements in the first hidden layer requires a total of 128 counts of 64 dimensional dot products. Similarly, computing the values at the second hidden layer requires 128 counts of 128 dimensional dot products, and computing the final layer requires 64 counts of 128 dimensional dot products. Hence, to compute one inference step in NeuMAC's deep network, we need to perform a total of 192 counts of 128-element dot products, and 128 counts of 64-element dot products. Further, since we can implement two 64-element dot products in parallel, one inference step requires an equivalent of 256 counts of 128 dimensional dot products to compute the output. Using this above macro design along with conservative and widely accepted hardware estimates, we next show that the design of NeuMAC's neural network architecture adds only marginal overheads, allowing it to operate under the resource constrained setting of a wireless NoC.

Latency Overhead: Here we estimate the latency of computing one inference step on NeuMAC's RL agent. The memory array is organized as 16 blocks of 64 by 256 memory elements, making a total of 32 KB storage. For 45nm technology, read access time from such memory sizes can be conservatively estimated to be around 2 ns [95]. Similarly, a 32-dimensional dot product can be computed within 2 ns [44]. Hence, we pipeline the data flow in three stages, first after the memory read, second after adding the outputs of 32 multipliers, and third at the output of the comparator bank. Hence, each stage has a maximum latency of 2 ns. As a result of such pipelining, one 128 element dot product is computed every 2 ns, that is, every 2 clock cycles[4]. As noted previously, one inference step requires 256 counts of 128 dimensional dot products. Hence, the total latency for one inference step is $256 \times 2 = 512$ *ns* (512 clock cycles). This inference latency of 512 cycles results in a small overhead of less than 6% per time step in our RL formulation. One point to note is that, the final deep network output is quantized to 8 bits. Hence, the sigmoid filter after the last layer can be implemented via a 256 element look-up table at a negligible latency overhead.

[3]Although NeuMAC's input has 65 elements, for simplicity sake we perform calculations with 64 element input.

[4]Our CPU clock is 1 GHz.

Energy Overhead: Next, we estimate energy consumption of the hardware macro. We use the energy values from the widely-cited paper [48], which approximately characterizes energy consumption of various compute elements and memory accesses. The dominant energy consumption steps are the reads from the memory array and the computations on the MAC (Multiply-ACcumulate) unit. From [48], 8 bit multiplies consume 0.2 pJ, and 8-bit additions consume 0.03 pJ. One 128 dimensional dot product on the MAC unit involves 128 multiplications and 127 additions. Thus the total energy comes to 29.41 pJ. Memory reads of 64 bits from 2 KB memory blocks requires 5 pJ. Thus, the 128 bit memory reads for each dot product requires 10 pJ. As a result, one 128 element dot product on the hardware accelerator requires 39.41 pJ, and with 256 counts, the energy consumed for a single inference step is 10088.96 pJ. Given that we require one inference every 10,000 ns, the neural accelerator consumes approximately only 1 mW of power on average. In comparison, a single transceiver on the multicore consumes 16 mW [38]. Lastly, note that the numbers in [48] are at 45 nm technology, so 1 mW is a conservative estimate.

Area Overhead: Lastly, the area overhead of the hardware macro is small. Since area is dominated by memory, the 32 KB of SRAM and few registers in the hardware accelerator impose a small overhead in comparison to the 512 KB of cache memory at each of the 64 cores. Thus we envision that such a hardware macro can reside on the same die and share the same clock as the multicore processor.

Thus, even a simple accelerator like the one demonstrated in Fig. 10 can enable NeuMAC's agent to operate under the resource constrained setting of a wireless NoC. Note that we do not employ any other advanced hardware optimization techniques and rely on reported hardware numbers that are widely accepted rather than the state-of-the-art today.

LightGuardian: A Full-Visibility, Lightweight, In-band Telemetry System Using Sketchlets

Yikai Zhao[†], Kaicheng Yang[†], Zirui Liu[†], Tong Yang[†,§], Li Chen[¶],
Shiyi Liu[†], Naiqian Zheng[†], Ruixin Wang[†], Hanbo Wu[†], Yi Wang[‡,§], Nicholas Zhang[¶]

[†]*Department of Computer Science, Peking University, China*
[§]*Peng Cheng Laboratory, Shenzhen, China* [¶]*Huawei Theory Lab, China*
[‡]*Southern University of Science and Technology*

Abstract

[1] Network traffic measurement is central to successful network operations, especially for today's hyper-scale networks. Although existing works have made great contributions, they fail to achieve the following three criteria simultaneously: 1) **full-visibility**, which refers to the ability to acquire any desired per-hop flow-level information for *all* flows; 2) **low overhead** in terms of computation, memory, and bandwidth; and 3) **robustness**, meaning the system can survive partial network failures. We design LightGuardian to meet these three criteria. Our key innovation is a (small) constant-sized data structure, called *sketchlet*, which can be embedded in packet headers. Specifically, we design a novel SuMax sketch to accurately capture flow-level information. SuMax can be divided into sketchlets, which are carried in-band by passing packets to the end-hosts for aggregation, reconstruction, and analysis. We have fully implemented a LightGuardian prototype on a testbed with 10 programmable switches and 8 end-hosts in a FatTree topology, and conduct extensive experiments and evaluations. Experimental results show that LightGuardian can obtain per-flow per-hop flow-level information within $1.0 \sim 1.5$ seconds with consistently low overhead, using only 0.07% total bandwidth capacity of the network. We believe LightGuardian is the first system to collect per-flow per-hop information for all flows in the network with negligible overhead.

1 Introduction

Network traffic measurement is central to successful network operations, especially for today's hyper-scale networks with more than 10^5 devices [1–6]. Meanwhile, at end-hosts, knowing the traffic information in the core of the network can also benefit application performance [7–9]. To infer application performance and user experience, the community consensus is to measure at flow-level granularity. Thus, an ideal measurement system is expected to achieve: 1) **full-visibility**, which we define as the ability to acquire any desired per-hop flow-level information[2] for *all* flows. Typical desired information includes routing path, per-hop latency, jitters, and packet drops. 2) **lightweight** in terms of computation, memory, and bandwidth, independent of the scale of network and the traffic dynamics; 3) **robustness**: the system should survive partial network failures, including link failures, device failures, and bandwidth depletion [6, 10–12].

Although existing works have made great contributions, they fail to meet the above criteria simultaneously. We coarsely characterize them into four categories:

- **Partial/Sampling** solutions [13–17] only sample packets or flows, or collect detailed statistics based on a pre-configured list of conditionals [18, 19]. For instance, Everflow [20] samples each SYN packet, and Cisco switches use the "match" keyword to specify which network flows need to be counted. Therefore, only a subset of the network traffic is measured with questionable accuracy.
- **Probing** solutions [6,21–24] measures the states of devices or links by sending probing packets, and only these probes are measured.
- **Sketch-based** solutions [25–34] collects the information of every packet in a compact data structure, namely sketch, on network devices. Current sketches are unable to collect important flow-level information, such as jitters and packet drops, and are not robust to network failures, particularly device failures. Most prior sketches cannot be implemented on P4-capable switches (§ 2.2).
- **In-band** solutions carry information in every packet header. AM-PM [35] cannot achieve full-visibility with only one bit per packet. Although INT [36, 37] can potentially achieve full-visibility, its bandwidth and processing overhead grows quickly with the scale of the network. In both the postcard [38] (mirroring packets on each switch) or the passport [36] mode (mirroring packets on only the sink switches), the number of packets is at least doubled, which is a huge burden for the network. *Flow [39] uses a cache to group packet-level telemetry information according to the flow IDs. But its bandwidth overhead is still proportional to the number of packets.

[1]Co-primary authors: Yikai Zhao, Kaicheng Yang, and Zirui Liu. Corresponding authors: Tong Yang (yangtongemail@gmail.com) and Yi Wang (wy@ieee.org).

[2]In this paper, per-hop flow-level information means per-flow per-hop information.

In summary, no existing work can achieve full-visibility without considerable performance overhead, and none focuses on lightweight and robust collection mechanism of network-wide flow-level information.

We design LightGuardian to meet the above three criteria simultaneously. Our key innovation is a (small) constant-sized data structure, called *sketchlet*. A sketchlet is a fragment of the sketch data structure on network devices (physical or virtual), carried in-band in a packet's header. At the end-host, LightGuardian collects the sketchlets, reconstructs the original sketch, and consequently obtains the accurate measurement results of all flows.

To support and make full use of sketchlets, LightGuardian incorporates three key techniques:

- **Accurate & versatile device-local sketches:** As current sketches fail to capture important flow-level statistics (per-hop latency and jitters), we design a novel sketch: the *SuMax sketch* to support common measurement tasks, as well as new tasks of operational importance. With our insight that recording both the sum and the maximum can accurately perform these tasks (§ 4.2), we design the sketch with two types of cells: the *sum cell*s and the *maximum cell*s. SuMax can be readily deployed on programmable network devices, and we have fully implemented it on a P4-capable switch (§ 7.1). Although SuMax is not the only way to measure flow-level statistics, it can support almost all measurement tasks thanks to its versatility.
- **In-band telemetry with sketchlets:** We propose a novel approach that combines in-band telemetry and device-local sketches. An INT-enabled device appends measurement data to each packet. INT alone consumes an enormous amount of bandwidth and multiplies the number of packets (§ 2.2). On the other hand, for sketch-based solutions, although sketch is a compact coding of flow-level information, the size of a sketch should be sufficiently large to ensure accuracy, thus cannot be embedded in packet headers. Combining the advantages of both approaches, our key novelty is to split the sketch with flow-level information into constant-sized sketchlets that can be embedded into selected packet headers. Since the number of flows in the network is much smaller than that of packets and the sketch is a compactly coded representation of flow-level statistics, the bandwidth overhead of sketchlets is significantly lower than that of INT, while accurate flow-level measurement can still be retrieved.
- **Incremental network-wide aggregation:** The receiving end-hosts can either forward the sketchlets to a global analyzer, or reconstruct the sketch locally to obtain measurement information of flows and devices inside the network. The information can assist end-host applications in performance optimization in a distributed fashion, which lessens the burden on the centralized network control/management plane. To guarantee robustness, we design the reconstruction algorithm to be tolerant of losses and reordering of sketchlets. Our algorithm can approximate a sketch with a subset of its sketchlets, and the reconstruction accuracy is incrementally improved with more arriving sketchlets. Our experimental results show that 80% sketchlets can achieve an accurate estimation (§ 8.1), while collecting 80% sketchlets only needs $1.0 \sim 1.5$ seconds.

To the best of our knowledge, LightGuardian is the first system to measure per-flow per-hop latency distribution and detect abnormal jitters with high accuracy for *all flows* on every participating network device, while maintaining low overhead. It also collects useful traffic data for operations and diagnostics previously unavailable in existing systems. Since LightGuardian aims to measure various per-hop flow-level information, after detecting end-to-end problems, users can use our system to locate culprit network devices. Besides, LightGuardian's on-device mechanism is not limited to physical devices, and can be readily used in cloud networking environments with virtualized network functions.

We have fully implemented a LightGuardian prototype on a testbed with 10 Tofino switches and 8 end-hosts in a FatTree topology. As a whole, our prototype can obtain per-flow per-hop information within $1.0 \sim 1.5$ seconds with consistently low overhead (0.07% of total bandwidth) on the network. We also conduct large-scale simulations using mininet [40] and P4 behavior model [41], confirming the correctness, robustness, and performance of LightGuardian. We release all source code anonymously [3].

In this paper, we make four key contributions:

We propose sketchlets and design a lightweight in-band telemetry system. Using sketchlets, our system makes the entirety of traffic information in the core of the network available at end-hosts for analytics and diagnostics. LightGuardian is lightweight, which takes up negligible bandwidth, and it can aggregate all sketchlets within 4 seconds.

We design the SuMax sketch to support common and more important measurement tasks with high accuracy. For common tasks (*e.g.*, flow size estimation), SuMax achieves 6.78 times smaller error rate. Further, LightGuardian can locate the culprit devices in the context of packet drops, inflated latency, and abnormal jitters, achieving almost 100% accuracy with less than 0.5MB memory.

We design an incremental reconstruction algorithm to achieve robustness and failure tolerance. Our experimental results show that, even when 50% end-hosts fail, the analyzer can still reconstruct 89% of all sketches. In addition, device and link failures do not affect the reconstruction of sketches of other devices (§ 8.3.2).

We implement a LightGuardian prototype and make it open-source. We also build a testbed and conduct extensive experiments, which confirms that our system can reach the design criteria.

[3] https://github.com/Light-Guardian/LightGuardian

Table 1: Comparison with the state-of-the-arts. In this table, "Impl." refers to implementation platforms; "P4 BM" refers to P4 behavior model [41]; "RMT" refers to Re-configurable Match Tables; and "PFPH" refers to "Per-Flow and Per-Hop".

Measurement Tasks		CM	FlowRadar	EverFlow	INT	AM-PM	LightGuardian
Device-local	Flow Size	✓	✓	×	✓	×	✓
	Flow Size Distr.	✓	✓	×	✓	×	✓
	Entropy	✓	✓	×	✓	×	✓
	Cardinality	✓	✓	×	✓	×	✓
Network-wide	PFPH Latency Distr.	×	×	×	✓	×	✓
	PFPH Packet Drops	×	×	×	✓	×	✓
	PFPH Jitters	×	×	×	✓	×	✓
	Forwarding Path	×	✓	✓	✓	×	✓
Impl.	RMT switches	✓	×	×	✓	✓	✓
	P4 BM	✓	✓	×	✓	✓	✓

2 Background and Related Work

2.1 Measurement Tasks

Existing measurement tasks can be classified into two categories: device-local measurement tasks and network-wide measurement tasks. Device-local measurement tasks refer to measuring flow-level information in a single node (an end-host, a switch or a router), and there have been various sketch-based solutions, such as sketches of CM [25], CU [26], Count [27], UnivMon [28], Elastic [29], SketchLearn [30], SketchVisor [42], and more [32–34, 43–48]. However, there are very few sketches designed for network-wide measurement tasks. This paper focuses on the following four network-wide measurement tasks.

1) Estimating Latency: We aim to estimate per-flow per-hop latency distribution. Existing works acquire end-to-end latency by sending probing packets. And they monitor specific flows by tracking their packets [49]. However, these solutions can hardly locate the victim flows and the culprit devices simultaneously. In contrast, per-flow per-hop latency distribution can help a lot but is more challenging.

2) Detecting Packet Drops: There are three causes of packet drops: random drops, loops, and blackholes. For random drops, the state-of-the-art LossRadar [50] uses a Bloom filter [51] and an Invertible Bloom Lookup Table (IBLT) [52] to accurately find drops. LossRadar works excellently in many cases. However, it consumes a lot of memory when a large flow drops many packets, which frequently happens when there are network misconfigurations [53]. For loops and blackholes, the state-of-the-art FlowRadar [54] also uses a Bloom filter and an IBLT, sharing the same advantages and shortcomings as that of LossRadar.

3) Detecting Abnormal Jitters: Jitters refer to drastic changes of packet inter-arrival time of a given flow. We aim to find abnormal jitters in the per-flow per-hop manner. Jitters are often caused by queuing, congestion, high bandwidth load, or network attacks. It can significantly affect the performance of streaming media (*e.g.*, audio, video, music). To detect jitters, end-to-end methods [55–57] have been proposed. However, they cannot work in the per-flow per-hop manner.

4) Tracing Forwarding Path: We aim to trace the forwarding path of any flow. Given a flow, tracing forwarding path can check whether the actual forwarding path is consistent with expectation. It can help test and/or debug new network protocols and network architectures, solutions for network congestion, load balance, and flow scheduling. Existing works for tracing forwarding path include FlowRadar [54], SwitchPointer [58], Service traceroute [59], and more [60–63].

2.2 Related Work

As shown in Table 1, compared with the state-of-the-art solutions, only our system supports device-local and network-wide measurement tasks, and it is implemented in both RMT switches (*e.g.*, Tofino) and P4 behavior model, achieving per-flow per-hop measurements. In this section, we mainly introduce the following four categories of measurement solutions. For other measurement solutions, please refer to references [64–72].

Partial/Sampling Solutions. Many measurement systems [14–17, 20, 46, 73–76] are developed by sampling packets. Sampling can significantly reduce the overhead of both time and space, but inevitably sacrifices accuracy and misses important events. Typical systems include NetFlow [73], sFlow [74], OpenSketch [46], OpenSample [17], Everflow [20], NitroSketch [77], and more [14–16, 76]. The state-of-the-art UnivMon [28] obtains elegant theoretical guarantees using a multiple sample solution at the cost of high time complexities. Sampling solutions probably miss many small flows, and thus cannot achieve the ideal goal of fully-visibility.

Probing Solutions. These solutions monitor the network by sending tailored packets. Typical systems include Pingmesh [78], NetBoncer [6], NetSonar [22], NetNorad [23], and more [24, 79–82]. Recently, AM-PM [35] gains wide recognition in industry. AM-PM divides packet streams into time periods, and the middle packet in each period is essentially a probing packet. Therefore, it only records per-period packet information, but is unaware of flow-level information. Probing solutions cannot achieve the ideal goal of fully-visibility because they cannot measure per-flow information.

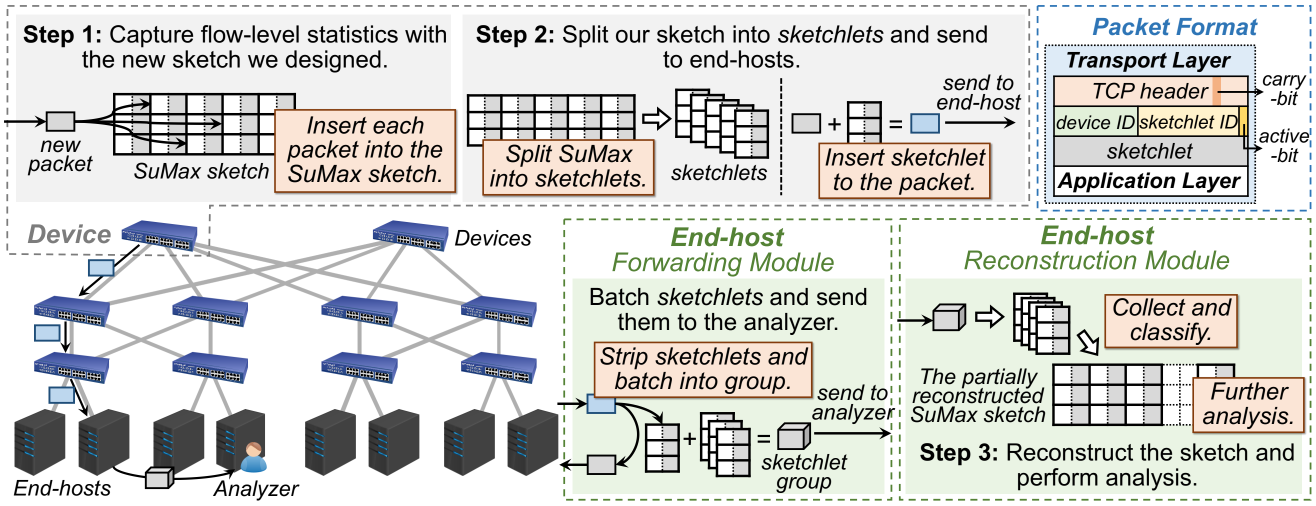

Step 1: Capture flow-level statistics with the new sketch we designed.
new packet
SuMax sketch
Insert each packet into the SuMax sketch.
Step 2: Split our sketch into sketchlets and send to end-hosts.
Split SuMax into sketchlets.
sketchlets
send to end-host
Insert sketchlet to the packet.
Packet Format
Transport Layer
TCP header
carry -bit
device ID
sketchlet ID
active -bit
sketchlet
Application Layer
Device
Devices
End-hosts
Analyzer
End-host Forwarding Module
Batch sketchlets and send them to the analyzer.
Strip sketchlets and batch into group.
send to analyzer
sketchlet group
End-host Reconstruction Module
Collect and classify.
The partially reconstructed SuMax sketch
Further analysis.
Step 3: Reconstruct the sketch and perform analysis.

Figure 1: LightGuardian Overview and Workflow.

Sketch-based Solutions. There are a great number of sketch-based solutions, which can be further divided into three categories. First, design and optimization of sketch algorithms. Typical solutions include sketches of CM [25], CU [26], Count [27], Elastic [29], and more [28, 30–32, 34]. But they cannot measure the latency and jitters. Some are implemented in P4 behavior model, but only a very few (BeauCoup [33], Elastic [29]) are fully implemented in real programmable switches . Due to the limitations of RMT (Re-configurable Match Tables) based programmable switches, *e.g.*, limited concurrent memory access, single stage memory access, and *etc.*, the implementation in real switches is significantly more challenging. Thus, we aim to design a new sketch to support both device-local and network-wide measurement tasks, while can be easily implemented in RMT switches. Second, measurement systems with dedicated sketches. Typical solutions include FlowRadar [54], SketchLearn [30], NitroSketch [77], and more [33, 46, 50]. The dedicated sketches can barely achieve fully-visibility. Third, measurement systems with shining features. Typical solutions include Marple [83], Sonata [84], DREAM [85], Scream [86], and OmniMon [87]. By carefully designing the resource manager and telemetry operator, OmniMon [87] achieves both resource efficiency and high accuracy. Our SuMax sketch can also be applied in OmniMon. All existing solutions do not focus on the overhead with aggregating sketches all over the network.

In-band Telemetry Solutions. They insert packet-level information into every packet. Well-known solutions include INT [36] and its successor PINT [88]. INT is considered as the most promising solution for network measurement because of two reasons. First, it can achieve fully-visibility because it is flexible to carry any desired packet-level information. Second, it can be implemented in RMT switches in a per-packet manner. However, its bottleneck lies in the aggregation of INT information. The INT information is distributed in every network packet, and it is obviously very challenging to aggregate that per-packet information. INT has two aggregation strategies: postcard and passport, which mirror every packet with only INT information in each switch or only the sink switches. Although the INT information in each packet is small (*e.g.*, 100 bytes), the total bandwidth overhead is huge. What is worse, the number of packets in the network will be doubled, which is a heavy burden for packet processing. Another in-band telemetry solution *Flow [39] uses a cache to group packet-level telemetry information according to the flow IDs. In this way, some information (*e.g.*, 5-tuple flow ID) in one group is recorded only once. However, its bandwidth overhead is still proportional to the number of packets.

3 LightGuardian Overview

As shown in Figure 1, LightGuardian captures flow-level statistics on each participating network device (physical or virtual)[4] using sketches. The devices periodically split the sketches into sketchlets and send the sketchlets to the end-hosts by piggybacking them in headers of appropriate packets. Then at the receiving end, the end-hosts batch the sketchlets into groups and send them to a global analyzer when the network load is low. Finally, the analyzer reconstructs the sketches and perform analysis.

1) Capture flow-level statistics with novel sketches. LightGuardian captures flow-level statistics by deploying our SuMax sketch on each participating device. Every packet is processed into the sketch *without sampling*. Typical collected statistics include the flow size (number of packets/bytes), per-hop delay distribution, the arrival time of the last packet, and the maximum inter-arrival time. The above statistics are used for detecting packet drops and measuring per-hop latency and maximum inter-arrival time, which are essential tasks for industrial community. To support more tasks, we can also include more collected statistics, *e.g.*, the number of out-of-order packets, the highest sequence number, and *etc.*

2) Split sketches into sketchlets and send them to the end-hosts. The participating devices periodically split their

[4]Since most legacy switches and routers do not have programmable dataplane capabilities, they cannot participate in LightGuardian (and their existence in the network will not hinder the functions of LightGuardian).

sketches into sketchlets and send them to the end-hosts. Specifically, at the start of each measurement interval, the devices initiate a new SuMax sketch to record the flow-level statistics. In the end, the sketches are divided into sketchlets of several bytes (24 bytes in our implementation). Each sketchlet is one column[5] (or several columns) of the sketch. Each switch then attaches these sketchlets to appropriate incoming packets. Specifically, we choose the packets that have not yet carried a sketchlet with a fixed probability (*e.g.*, 0.05).

3) [Optional] Batch sketchlets and forward to a global analyzer. LightGuardian has two working modes: 1) Local analysis mode, where each end-host uses local sketchlets to perform analysis for local applications; 2) Global analysis mode, where a global analyzer collects all sketchlets and performs analysis for network operators. If an end-host does not want to perform local analysis, it can choose to forward sketchlets to a global analyzer. A system daemon process running on each end-host strips sketchlets off the packets, and maintains the received sketchlets. When the process has collected enough sketchlets, it batches the received sketchlets into groups and forward them to the global analyzer. For LightGuardian, more than 350 sketchlets are grouped into a UDP packet and share 42 bytes packet header, which significantly reduces the number of additional packets for measurement.

4) Reconstruct sketches and perform analysis. The end-host or the global analyzer (or end-hosts) can reconstruct the sketchlets into sketches and perform further analysis. The process of reconstructing sketches proceeds simultaneously with the process of collecting sketchlets. After collecting enough sketchlets, the analyzer can perform accurate estimation using the partially reconstructed sketch. According to our experiments, after receiving 55% sketchlets, our LightGuardian reports 90% valid results, while the average relative error (ARE) is only 0.088. Further, the estimation results are incrementally refined with more and more sketchlets collected, the ARE reduces to 1×10^{-2}, 1×10^{-3}, 2×10^{-4} when 80%, 90%, and 100% sketchlets are received, respectively.

In this way, LightGuardian well achieves the three mentioned design goals. For full-visibility, LightGuardian deploys SuMax sketch on each network device to monitor various per-flow per-hop information for all flows. For low overhead, LightGuardian uses small and constant-sized sketchlets to transmit measurement information, which makes the in-band overhead grow sub-linearly with the network/traffic scale. For robustness, the reconstruction process of LightGuardian does not require collecting all sketchlets whereas providing desirable accuracy. Besides, any end-host with limited computation resources can play the role of the global analyzer, which makes our system robust.

[5]A sketch consists of multiple bucket arrays, and a column refers to the buckets with the same index in each array.

4 Device-local Sketch Design: SuMax

4.1 Motivation

We design the SuMax sketch to achieve accurate measurement of flow-level information on network devices of different platforms: software (CPU, or OVS [89]), P4 behavior model [41], programmable switches. To make LightGuardian widely applicable, this paper focuses on P4 behavior model and programmable switch platforms, as the software implementation is straightforward. Using P4 also ensures our implementation can be compiled to available and future P4 back-ends, such as SmartNIC, FPGA and GPU.

UnivMon [28] and HashPipe [90] are implemented in P4 behavior model, but can hardly be implemented in RMT switches. To address these issues, Basat *et al.* proposed using a recirculate method [91], inevitably incurring complexities and degradation of switch throughput. BeauCoup [33] and Elastic [29] have been implemented in RMT switches (*i.e.*, Tofino switches) by complicated designs and programs. Further, the above four sketches cannot be directly used for network-wide measurement tasks, such as estimating latency and jitters. We found CM [25] is the most friendly sketch for programmable switches. On the one hand, we optimize its accuracy under the constraints of programmable switches. On the other hand, we extend its functions to support both device-local and network-wide tasks. In the meantime, we try to keep the designed sketches as simple as possible.

4.2 Rationale and Design Space for Sketches

We first introduce the well-known CM sketch [25]. It is a typical sketch algorithm that sums packet attributes (*e.g.*, packet number, bytes number). It uses d counter arrays $\mathcal{A}_0, \cdots, \mathcal{A}_{d-1}$. For each array, it has a hash function $\mathcal{H}_i(\cdot)$ to map a flow[6] uniformly and randomly into a counter. When a packet of flow f with attribute value α arrives, CM selects the counter $\mathcal{A}_i[\mathcal{H}_i(f)]$ for each array $\mathcal{A}_i$ and increments these counters by α. To query the attribute sum of flow f, CM returns the minimum value among $\mathcal{A}_0[\mathcal{H}_0(f)], \cdots, \mathcal{A}_{d-1}[\mathcal{H}_{d-1}(f)]$, which is still a sum of attributes of some flows. Therefore, CM has only over-estimation errors. Similarly, the CU sketch [26] increments only the smallest counter(s), significantly improving the accuracy but not supporting pipeline implementation.

We propose to record both of the sum value and the maximum value[7] to support versatile tasks. We insist that all packet attributes can be accurately estimated by keeping only the sum and maximum values. We also insist that either sum or maximum value is indispensable. For example, sketches of CM, CU, Count, FlowRadar cannot be used to find maximum latency or inter-arrival time and last arrival time, because they only record the sum value without maximum value.

[6]A flow has many packets sharing the same flow ID, which can be any combination of 5-tuple: source IP address, source port, destination IP address, destination port, protocol type.

[7]Note that [92] also suggests that the sketch algorithm can be used to find the maximum value in a sequence.

Table 2: Symbols frequently used in this paper.

Symbol	Meaning
f	An arbitrary flow
α	An attribute that needs to be recorded in the sum cell (*e.g.*, packet size)
β	An attribute that needs to be recorded in the maximum cell (*e.g.*, arrival time)
d	SuMax consists of d bucket arrays
w	Each array consists of w buckets
$\mathcal{A}_i$	The i-th bucket array
$\mathcal{A}_i^{sum}[\cdot]$	The *sum cell* in a bucket
$\mathcal{A}_i^{max}[\cdot]$	The *maximum cell* in a bucket
$\mathcal{H}_i$	A hash function from a flow to $\{0,\cdots,w\}$

4.3 Data Structure and Operations

Data Structure (Figure 2): Our SuMax consists of d bucket arrays $\mathcal{A}_0,\cdots,\mathcal{A}_{d-1}$. Each array $\mathcal{A}_i$ contains w buckets $\mathcal{A}_i[0],\cdots,\mathcal{A}_i[w-1]$. Each bucket has two cells: a *sum cell* and a *maximum cell*, recording the sum value and the maximum value of attributes, respectively. Each array $\mathcal{A}_i$ is associated with a hash function $\mathcal{H}_i(.)$ that maps a flow into one of its buckets. To support various tasks, we may need more than one sum value or maximum value in each bucket. For convenience, we only show using one sum value and one maximum value. Table 2 lists the frequently used symbols in this paper.

Insertion: To achieve high accuracy and support pipeline implementation, we propose an approximate conservative update strategy as follows. To record a packet of flow f with attribute α and β ($\langle f,\alpha,\beta\rangle$, α will be accumulated and β will be compared with the maximum), we first maintain a current minimum value ω and initialize it to ∞. For each array $\mathcal{A}_i$, we select a bucket $\mathcal{A}_i[\mathcal{H}_i(f)]$ by computing the hash function $\mathcal{H}_i(f)$. For each selected bucket $\mathcal{A}_i[\mathcal{H}_i(f)]$, we check its *sum cell* $\mathcal{A}_i^{sum}[\mathcal{H}_i(f)]$ and update it as follows:

- If $\mathcal{A}_i^{sum}[\mathcal{H}_i(f)]+\alpha<\omega$, update the current minimum value $\omega=\mathcal{A}_i^{sum}[\mathcal{H}_i(f)]+\alpha$, and set the cell to ω.
- If $\mathcal{A}_i^{sum}[\mathcal{H}_i(f)]+\alpha\geqslant\omega$, and $\mathcal{A}_i^{sum}[\mathcal{H}_i(f)]<\omega$, set the cell to ω.
- If $\mathcal{A}_i^{sum}[\mathcal{H}_i(f)]\geqslant\omega$, we keep the cell unchanged.

For the *maximum cell* $\mathcal{A}_i^{max}[\mathcal{H}_i(f)]$, we just set it to $\max\{\mathcal{A}_i^{max}[\mathcal{H}_i(f)],\beta\}$. The pseudo-code of the insertion operation is shown in Algorithm 1 in Appendix A.

Query: Given a flow f, SuMax returns two results: one sum value estimation and one maximum value estimation. The sum estimation is the minimum value among $\mathcal{A}_0^{sum}[\mathcal{H}_0(f)]$ $,\cdots,$ $\mathcal{A}_{d-1}^{sum}[\mathcal{H}_{d-1}(f)]$. The maximum value estimation is the minimum value among $\mathcal{A}_0^{max}[\mathcal{H}_0(f)]$ $,\cdots,$ $\mathcal{A}_{d-1}^{max}[\mathcal{H}_{d-1}(f)]$.

Example (Figure 2): To record a packet $\langle f,\alpha=3,\beta=4\rangle$, SuMax updates the d ($d=3$) buckets $\mathcal{A}_0[\mathcal{H}_0(f)]$, $\mathcal{A}_1[\mathcal{H}_1(f)]$, $\mathcal{A}_2[\mathcal{H}_2(f)]$ as follows. For the bucket $[6,3]$, we increase 6 to 9, set ω to 9, and set 4 to $\max\{4,3\}$. For the bucket $[9,7]$, as $9\geqslant\omega$ and $7\geqslant 4$, we keep this bucket unchanged. For the bucket $[3,5]$, as $3+\alpha<\omega$, we update ω to 6 and update 3 to

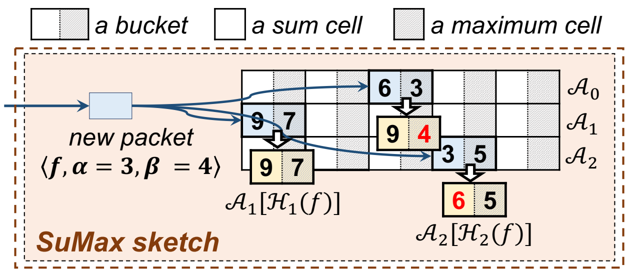


Figure 2: An example of SuMax.

$\omega=6$; as $4<5$, we do not change the maximum cell. After the insertion, when query flow f, SuMax returns $\min\{9,9,6\}=6$ as the sum estimation, and returns $\min\{4,7,5\}=4$ as the maximum value estimation.

Analysis: Our SuMax uses an approximate conservative update strategy to achieve both accuracy and pipeline friendly. Note that the conservative update strategy (CU) cannot be implemented in the pipeline because it needs the traceback operations to only increase the smallest counter(s). Our idea is to use the current minimum value to approximate the global minimum value. In each insertion process, with more and more counters accessed, the current minimum value will be closer and closer to the global minimum value, and thus the updated counter will be closer and closer to CU. Actually, the first array is updated following the rule of CM, and the last array is updated following the rule of CU. Since the counters in the last few arrays tend to have smaller values, they are more likely to be returned as query results. Therefore, SuMax can be viewed as an intermediate between CM and CU, and its error is also bounded between them, but closer to CU. As there are no tracebacks in our SuMax, it can be easily implemented in the switch pipeline.

4.4 Configuration of SuMax Sketch

In current implementation, we design each bucket as follows. Each bucket consists of four parts:

- a *flow-size cell* (sum cell) recording the flow size;
- λ_d *delay cells* (sum cells) recording the per-hop delay distribution, each one of which is associated with a predefined delay time interval;
- an *interval cell* (maximum cell) recording the maximum inter-arrival time;
- a *last-time cell* (maximum cell) recording the arrival time of the last packet of a flow.

All cells are initialized to zero. When the cells of a bucket are going to update, they should be updated as follows. Let t_{now} be the ingress timestamp of this packet, t_{last} be the value of the *last-time cell*, and $t_{interval}=t_{now}-t_{last}$. When $t_{last}=0$, we consider the current packet as the first packet of a flow, and set $t_{interval}=0$. First, we increment the *flow-size cell* by 1. Second, we select one cell from the λ_d *delay cells* according to the packet delay, and increment this cell by 1. Third, we compare the value in the *interval cell* with $t_{interval}$ and update it accordingly. Fourth, we update the *last-time cell* to t_{now}.

5 Transmission of Sketchlets

In this section, we show the transmission procedure of sketchlets in each participating network device. First, we split sketches into sketchlets. Second, we sample packets to carry sketchlets. Third, we select a sketchlet to be carried using our selection strategy and insert it into the packet.

5.1 Splitting the Sketch into Sketchlets.

LightGuardian deploys two SuMax sketches (one active and one idle) on each device. The active sketch is used to record flow-level information, while the idle sketch is split into sketchlets for transmission. After a fixed time interval (*e.g.*, 5s), we interchange these two sketches. We use an *active-bit* to indicate which sketch is active. The *active-bit* is flipped periodically, and the current interval is set to 5 seconds.

We split the idle sketch column by column, so that each sketchlet contains a column of buckets. Each sketchlet is associated with **1)** a *Sketchlet ID* indicating the column index; **2)** a *Device ID*; and **3)** the *active-bit* indicating which one of the two sketches it belongs to. The analyzer will sort the received sketchlets (bucket columns) according to the *Device ID*, *active-bit*, and *Sketchlet ID*.

5.2 Probabilistically Carrying Sketchlets.

Given an incoming packet, the device first checks the packet header: if it has already carried a sketchlet, no more sketchlets will be carried. Otherwise, the device calculates a fixed carrying probability λ_c (*e.g.*, 0.05) to determine whether this packet should carry a sketchlet. Each device samples only a part of the packets to carry sketchlets with λ_c, so that every device has a similar opportunity for packet transmission.

The packet format is shown in Figure 1. If a packet is selected to carry sketchlet, we insert the sketchlet between the TCP header and the application-layer message. First, we use a bit in the TCP header (*carry-bit*) to indicate whether this TCP packet carries a sketchlet. Second, we add a field to record the device ID (16 bits). Third, we add a field to record the sketchlet ID and the *active-bit*.

5.3 Sketchlets Selection: *K+chance Selection*.

Once the device determines the incoming packet should carry a sketchlet, we need an algorithm to choose a sketchlet. In-band telemetry solutions will lose measurement information when packet drops happen. To address this issue, we can send a sketchlet several times at the cost of more bandwidth usage. An effective solution is to use a counter array. Specifically, each counter corresponds to a sketchlet, indicating the number of times this sketchlet has been carried. For the incoming packet, we locate several counters by computing hash functions, find the smallest counter among them, and choose the corresponding sketchlet to carry. As mentioned above, similar to the CU sketch, this solution cannot be implemented in current P4-programmable switches, and thus we propose a new algorithm namely *k+chance selection*.

The *k+chance selection* uses k arrays, each of which is an N-bit array. For each array, each bit corresponds to a sketchlet. All bits are initialized to 0. Whenever we need to select one of the N sketchlets, we access the k arrays one by one. For each array, we randomly choose a bit: if it is zero, we choose the corresponding sketchlet and set this bit to 1; Otherwise, we access the next array. In the worst case, we do not find a zero bit after accessing all the k arrays, and we randomly choose one sketchlet to transmit. In this way, we only need to record an array ID in each sketchlet, which just takes $\lceil \log(k+1) \rceil$ bits (2∼3 bits). By contrast, when using the simple round-robin, we need to record the column ID (usually 32 bits) in each sketchlet. *K+chance selection* is an approximately fair selection algorithm for hardware platforms. Our experiments show that *k+chance selection* works well (§ 8.1).

6 Reconstruction and Analysis

In this section, we first describe the two modules at the end-hosts: forwarding module and reconstruction module. These two modules can work in isolation or in parallel. Then we elaborate on how to obtain device-local measurements and network-wide analysis using SuMax.

6.1 End-host Modules

Reconstruction Module. This module dynamically classifies the received sketchlets into groups according to their device IDs and *active-bit*s, and sorts the sketchlets in each group by their sketchlet IDs. In this way, the end-host reconstructs a sketch for each group. Note that the reconstructed sketches might be incomplete because some sketchlets are still in the network or missing. Fortunately, an incomplete sketch can also be used to answer queries: each query will access d buckets, some of which may not have been received yet. We consider the values in these buckets as invalid, and report the minimum value among the other valid buckets. As long as one of the d buckets is valid, we can report a valid result. Otherwise, we report the result of invalid. In this way, after some sketchlets are collected, the end-host then uses these reconstructed sketches to perform further analysis. Our experimental results (see § 8.1) show that 55% sketchlets can report 90% valid results and achieve accurate estimation ($ARE < 0.1$). The following theorem provides theoretical guarantees for the reconstruction process.

Theorem 6.1 *After receiving sketchlets with a ratio of* θ, *SuMax can report valid results with a ratio of* $(1-(1-\theta)^d)$. *Specifically, when the result is valid, the estimated flow size has the following error bounds.*

$$\Pr\left\{|\hat{n}_f - n_f| > \varepsilon\right\} < \frac{\left(\frac{m\theta}{w\varepsilon} + 1 - \theta\right)^d - (1-\theta)^d}{1-(1-\theta)^d}$$

where d and w are parameters of SuMax (see Table 2), n_f and $\hat{n}_f$ are the real and estimated flow size, and m is the number of inserted packets.

The module can reconstruct the following four widely-studied [28, 29, 54] device-local measurements:

- *Flow Size Estimation.* We return the minimum value of the d mapped flow-size cells.
- *Flow Size Distribution.* We use the MRAC [93] algorithm with the first bucket array in SuMax as input.
- *Entropy.* We compute $-\sum(n_i \cdot \frac{i}{m} \log \frac{i}{m})$ based on the flow size distribution, where n_i is the number of flows with size of i, and $m = \sum(i \cdot n_i)$.
- *Cardinality.* We calculate the number of flows using the method of linear counting [94].

We believe the main advantage of this approach is that the measurement is done in a distributed fashion, without a centralized control or management plane.

Forwarding Module. End-hosts use this module to forward sketchlets to a global analyzer for network-wide analysis. Each end-host groups the received sketchlets into batches. The end-host will send a batch of sketchlets to the analyzer when appropriate. 1) When the bandwidth usage is high, the end-host does not send sketchlets. 2) When the number of the accumulated sketchlets reaches a threshold, or the end-host has not sent any sketchlets for a certain period, it will send all the accumulated sketchlets to the network-wide analyzer. The network-wide analyzer reconstructs the sketches as the end-hosts do, and then performs the network-wide analysis.

6.2 Network-wide Analysis with SuMax

For the following four network-wide analysis tasks, we need to access different SuMax cells for different tasks. To perform network-wide analysis tasks, we have two steps. First, the network operator detects abnormal end-to-end incidents (*e.g.*, TCP duplicate ACKs, TCP timeout[8]), and report the victim flows to the network-wide control plane analyzer. Second, based on the network topology, the analyzer further investigates the sketches on the switches in the forwarding path of the victim flow as to locate the specific culprit device or link.

Locating Inflated Latency. Locating inflated latency refers to finding out the culprit switch, and the victim flow when inflated end-to-end latency occurs. First, the end-host detects abnormal incidents of inflated end-to-end latency, and reports the ID of the victim flow. Second, the analyzer queries the per-hop latency distribution of this flow by accessing the delay cells in the corresponding reconstructed sketches. In this way, it can easily locate the culprit switches with inflated latency (*e.g.*, a switch on which 80% packets have $> 10\mu s$ latency).

Locating Packet Drops. As mentioned above, there are three main packet drops behaviors: random drops, loops, and black-holes. Random drops may result from hardware failures (*e.g.*, faulty interfaces in switches). Loops may result from the mis-configuration of the forwarding table, which leads the packets of the victim flows forever loop among several switches. Blackholes may result from forwarding entries corruption in culprit switches. After detecting end-to-end packet drops from TCP re-transmission, timeout, or ping probe loss, the end-host (sender) reports the flow ID to the analyzer. To locate the culprit switch, the analyzer queries the victim flow in every sketch on the forwarding path by accessing the flow-size cells. **1)** If the flow size suddenly drops to 0 after passing a switch, we report the switch as a blackhole. For example, suppose there are five switches ($s_1 \sim s_5$) on the forwarding path. If the estimated flow sizes on the five switches are $100, 100, 100, 0, 0$, respectively, we report s_4 as a blackhole. **2)** If the flow size is abnormally large on several switches, we infer a loop happens on them. For the same example with five switches, if the flow sizes are $100, 100, 5000, 5000, 0$, respectively, we infer that s_3 and s_4 probably be involved in a loop. **3)** If the flow size slightly decreases after passing a switch, we infer that the switch suffers random packet drops. For the same example, if the flow sizes are $100, 100, 95, 95, 95$, we infer random packet drops happen on s_3.

Locating Abnormal Jitters. After detecting end-to-end variation in the packet inter-arrival time of a flow, the end-host reports the flow ID to the analyzer. The analyzer queries the maximum inter-arrival time of that flow, and finds out the culprit switches on which the result is abnormally large.

Finding Abnormal Forwarding Path. When an end-host receives a packet which carries a sketchlet not belonging to the switches on the expected forwarding path, we report this packet suffers abnormal forwarding.

7 Prototype Implementation

In this section, we first describe the workflow and difficulties we face when implementing a LightGuardian prototype on a programmable switch (Tofino-40GbE). On each switch, we develop SuMax and the sketchlet transmission mechanism using P4 [96]. Then we overview the components in the end-hosts: the kernel modules to collect and forward the sketchlets.

7.1 SuMax on Programmable Switches

All existing sketches can be implemented in the software (*e.g.* middleboxes, virtual network appliances, *etc.*), but most of them cannot be deployed on programmable switches, which limits their applicability outside of cloud networking environments. For LightGuardian, since deployability is crucial to achieving full-visibility in all network environments, we first show that SuMax can be deployed on programmable switches by implementing it on a Tofino-40GbE switch.

7.1.1 Workflow

On the switch, we design the workflow (relevant to LightGuardian) (Figure 3) as follows: we put Decision Making Stage in the ingress pipeline, and Sketching Stage and Sketchlet Generation Stage are placed in the egress pipeline.

The **Decision Making Stage** decides the following:

[8]Some tools provided by the OS (*e.g.*, ePBF [95]) can help operators to easily detect these abnormal incidents.

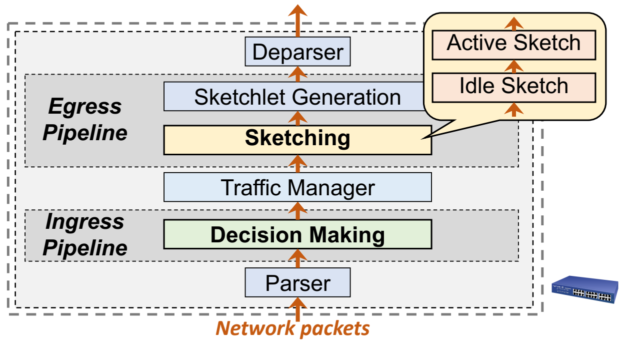


Figure 3: Workflow on an RMT switch.

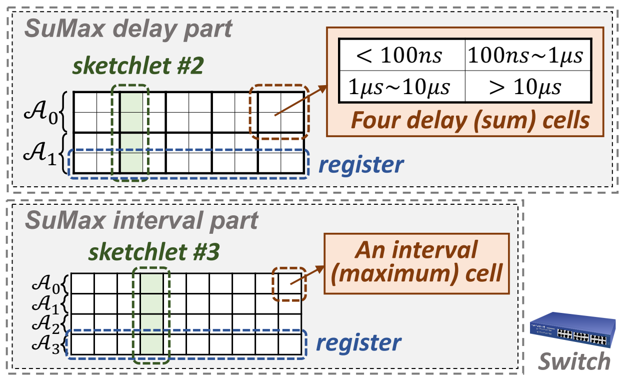


Figure 4: Sketch implementation on an RMT switch.

- **Active sketch**, *i.e.* which sketch should be inserted into. As mentioned in § 5.1, we deploy two SuMax sketches on each switch, and use the *active-bit* to identify the *active* sketch. Note that the data plane of the switch *cannot* periodically update the active-bit. Therefore, we run a process in the switch control plane to periodically flip it. As the flipping is asynchronous, we forbid carrying sketchlets in the last second in each measurement interval.
- **Fitness for sketchlets**, *i.e.* whether the packet should carry a sketchlet. The fitness conditions are: 1) the packet is not carrying a sketchlet; 2) For each packet, we use its 5-tuple and its ingress timestamp to calculate a 16-bit hash value (CRC16), and only when the value falls within $[0, \lambda_c 2^{16})$, the packet is selected to carry a sketchlet. λ_c is a pre-configured parameter, and the second condition is approximately allowing a packet to carry sketchlet with a probability of λ_c
- **Sketchlets selection.** As described in § 5.3, we use the *k+chances selection* algorithm to select a sketchlet to carry. Thus, we need to randomly select a bit for each bit array. In Tofino switches, we can only achieve pseudo-randomness: we still use CRC16 to generate approximately random numbers, and choose reasonable polynomials of CRC16 to generate multiple approximately independent random numbers. Due to limitation of Tofino switch, we set $k = 1$.

In the **Sketching Stage**, we place two sketches: one idle and one active. Their status is periodically flipped. These two sketches are two match-action tables placed in the egress pipeline, so each packet will pass them sequentially. For each packet, the sketch table checks the active-bit. If the active-bit indicates the current sketch is active, we hash the flow ID to update the corresponding cells to record packet information. The update procedures of SuMax are challenging on Tofino, and we highlight the difficulties below (§ 7.1.2).

The **Sketch Generation Stage** reads the selected sketchlet and writes it into the metadata if the packet is selected.

7.1.2 Challenge of Sketching Stage

SuMax records multiple packet attributes (*e.g.*, flow size, delay distribution, last arrival time, maximum inter-arrival time). This requires multiple cells in each bucket. In Tofino switches, the cells in SuMax are stored in *registers*. A switch has 12 *Match-Action Units* (MAU), each of which contains up to two 256KB registers. Since 6 MAUs are used in other stages, only 6 MAUs (12 registers) can be used in the Sketching stage. The main challenge is that, each incoming packet can only access each register exactly once, and each access can only read/write up to 64 consecutive bytes.

Thus, we have to assign the cells in a single bucket to multiple registers. In other words, we need to divide SuMax into parts. We use two examples, the measurement of latency distribution (sum) and that of maximum packet inter-arrival time, to illustrate our solution.

Latency distribution. We use the delay part of SuMax to perform this task. As shown in Figure 4, this part consists of $d = 2$ bucket arrays, each of which has $w = 2^{15}$ buckets. Each bucket has $\lambda_d = 4$ sum cells (32-bit), each of which corresponds to a predefined delay range. To make full use of the registers, we observe that:

- The four cells in each bucket should not be assigned to a single register. Since each 256KB register stores up to 2^{16} 32-bit cells, using a single register will limit the size of the sketch (up to 2^{14}), which compromises the accuracy.
- Using four registers to store the four cells in each bucket cannot be implemented on Tofino switch. As each switch has two sketches, each of which contains at least two bucket arrays, so we need at least 16 registers, while at most 12 registers are available in the Sketching stage.

Thus, we propose to use one register to store two cells in each bucket, as shown in Figure 4. We divide each bucket array into two registers, the first contains the first two cells of each bucket, and the second contains the remaining two cells. We group 4 cells in the same column into a sketchlet. In this way, either the active or the idle sketch is updated, each register is accessed *only once* for a packet.

Packet inter-arrival time is a task of measuring the maximum value, and its implementation is much easier. As shown in Figure 4, we set $d = 4$ and $w = 2^{16}$. For each bucket array, all 32-bit interval-cells are assigned to one register. We still group the 4 interval-cells in the same column into a sketchlet.

7.2 End-host Components

LightGuardian needs to implement three functions on the end-hosts: sending packets, receiving packets, reconstruction and performing analysis.

Sending packets: In the current implementation, LightGuardian inserts sketchlets between the Ethernet header and the IP header. However, we should emphasize that this is mainly due to hardware limitation: TCP checksum recalculation on Tofino is unreliable currently.

We also add the *carry-bit* after the Ethernet header, because there is no space in the Ethernet header. To implement this design, we program a Linux kernel module on the end-host, which registers a new packet type `ETH_P_SKETCHLET` in the Layer-3 protocol stack and modifies the Ethernet type of each packet to be sent to `ETH_P_SKETCHLET`, and allocates extra space for the *carry-bit*.

Receiving packets: We implement another Linux kernel module to handle `ETH_P_SKETCHLET` packets. This module decides whether the packet carries a sketchlet by checking the *carry-bit*, and records the sketchlet in the `stderr`.

Reconstruction and Forwarding We implement a forwarding module for end-hosts to forward the sketchlets to a centralized analyzer. It reads `stderr` every 1 millisecond. When the process finds the number of sketchlets in the log exceeds a threshold (dependant on Maximum Transmission Unit (MTU) of the network), or when a timeout is reached, the module generates a packet containing all the received sketchlets of the current interval, and sends it to the central analyzer. For example, when MTU is 9KB, the threshold is set to 350 packets ($\sim$8.4KB). We set the timeout to 100 milliseconds.

Finally, analysis can be performed on the end-host or the centralized analyzer with the same sketch reconstruction algorithm described in § 6.1.

8 Experimental Results

We conduct extensive experiments on a testbed and using mininet [40]. We focus on the following four key issues.

- **How accurate can our SuMax sketch measure per-flow statistics?** We implement our SuMax sketch using C++, and use the CAIDA datasets to evaluate the accuracy of SuMax for seven measurement tasks.
- **How much is the overhead of sending and aggregating sketchlets?** We generate network traffic following the widely used traffic distributions (WEB [97] and DCTCP [98]). We evaluate the aggregation time, the bandwidth overhead, and the impact on network performance (*e.g.*, RTT, FCT).
- **How accurate can LightGuardian detect network anomalies?** We use mininet to simulate a network, and evaluate the accuracy of LightGuardian in locating blackholes, loops, and abnormal jitters.
- **Is LightGuardian resilient to network failures?** We evaluate the performance of LightGuardian when end-hosts fail, or some sketchlets are missing.

We conduct the experiments using the following metrics: **ARE**, **RR**, **PR**, F_1 **Score**, **RE**, and **WMRE**. We explain the details of these metrics in Appendix C.

8.1 Experiments on SuMax

We use the anonymized IP traces collected in 2018 from CAIDA [99]. The dataset contains 6M packets belonging to 0.9M different flows. We set $d = 3$ by default, which means there are 3 bucket arrays in SuMax.

Flow size estimation (Figure 5a): We find that the accuracy of SuMax is higher than CM and close to CU. When using 96KB of memory, the ARE of SuMax is 6.78 times lower than CM, and 1.75 times higher than CU. We further study how the flow sizes affect the accuracy (see Figure 11a in Appendix D.1), and find that the results hold for both large and small flows.

Robustness (Figure 5b-5c): We find that partially reconstructed SuMax can provide accurate estimation. We set the memory to 768KB and measure the valid query rate and the ARE of the largest 1K flows. The results show that 55% reconstructed SuMax can report >90% valid results with <0.1 ARE, and 80% reconstructed SuMax can report >99% valid results with <0.01 ARE.

Other device-local tasks (Figure 5d-5e): We find that besides flow size estimation, SuMax also achieves good performance in other device-local measurement tasks, including estimating cardinality, flow size distribution (see Figure 11c in Appendix D.1), and entropy.

Delay distribution (Figure 5f): We find that the accuracy of SuMax is higher than CM and close to CU. We generate the delay of each packet according to the *chi-square distribution*. We set $\lambda_d = 8$ and vary w from 2^{10} to 2^{17}. For other delay distribution, please refer to Figure 12a-12e in Appendix D.1.

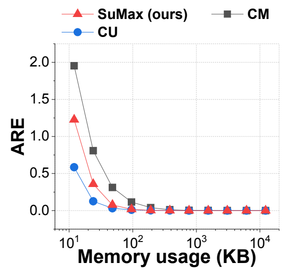
Maximum inter-arrival time (Figure 5g): We find that SuMax achieves <10 ARE when using more than 6MB of memory, and <0.3 ARE when using more than 12MB of memory. Since when abnormal incidents happen, the maximum inter-arrival time will rapidly increase dozens or hundreds times, <10 ARE is accurate enough to locate problems. We further study how the flow sizes affect the accuracy (see Figure 12h in Appendix D.1), and find that the results hold for both large and small flows.

***k+chance* Selection (Figure 5h):** We find that *k+chance Selection* can effectively reduce the number of packets required by the reconstruction process. According to the results, a larger k goes with fewer required packets, which demonstrates the effectiveness of our algorithm. We also find that the larger the w, the better the optimization effect.

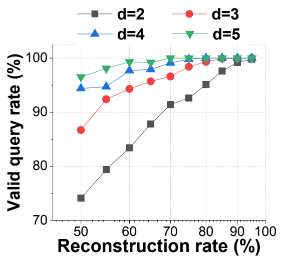
We further study the memory overhead of SuMax, and find that its memory overhead grows sub-linearly with the network scale, which guarantees the scalability of LightGuardian (see Figure 11b and Table 3 in Appendix D.1).

8.2 Testbed Experiments

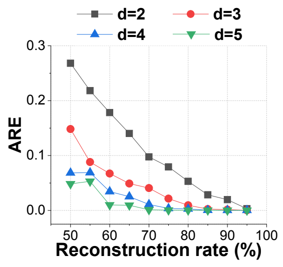
We evaluate LightGuardian on the testbed described in § 7. Take the delay distribution measurement task as an instance, the SuMax sketch we used contains $d = 2$ bucket arrays, each of which has $w = 2^{15}$ buckets, and each bucket contains 4



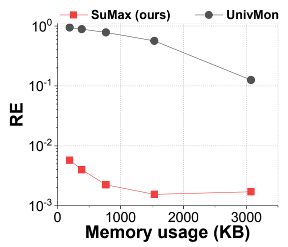
(a) Flow size estimation.



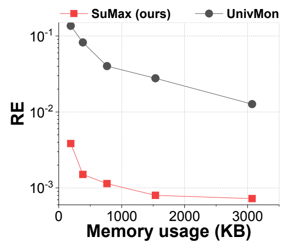
(b) Valid query rate.



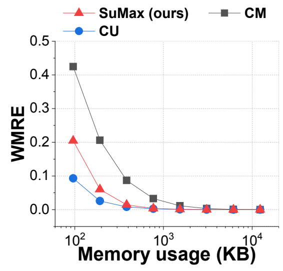
(c) ARE.



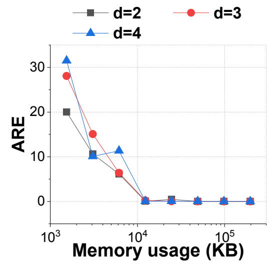
(d) Cardinality.



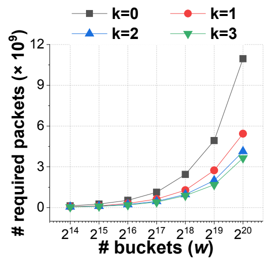
(e) Entropy.



(f) Delay distribution.



(g) Max inter-arrival time.



(h) *K+chance Selection.*

Figure 5: Experimental results on SuMax and the selection algorithm.

delay cells. This sketch also supports locating packet drops and all the mentioned local measurement tasks. Similarly, we can use another SuMax to locate abnormal jitters. By default, we set the carrying probability λ_c to $\frac{1}{16}$, and set $k = 1$ for *k+chance selection*. We use two traffic distributions W1 (DCTCP [98]) and W2 (WEB [97]), which are widely used in existing works [8, 100–102]. On each switch, the number of the sketchlets is 2^{16}, and each sketchlet is 24 bytes.

Bandwidth overhead *v.s.* traffic load (Figure 6a): We find that our system saves substantial bandwidth than INT. We compare LightGuardian with a kind of INT that inserts 20-bytes per-packet information into the packet headers at each hop. The results show that the bandwidth overhead of LightGuardian ranges from 13.8Mbps to 25.7Mbps, which is only about 0.07% of the total bandwidth. The bandwidth overhead of INT ranges from 211Mbps to 394Mbps. Compared with INT, our LightGuardian saves more than 93.5% bandwidth. We also study how the carrying probability λ_c affects the bandwidth usage (see Figure 13a-13c in Appendix D.3).

FCT *v.s.* traffic load (Figure 6b): We find that LightGuardian has little impact on the network. We vary the bandwidth usage from 50% to 90%, and measure the average Flow Completion Time (FCT) before and after deploying LightGuardian. Under workload W1, after deploying LightGuardian, the average FCT increases by 8.3% at 50% traffic load, and 1.8% at 90% traffic load. Under workload W2, after deploying LightGuardian, the average FCT increases by 16.3% at 50% traffic load, and 5.6% at 90% traffic load. Even under 90% traffic load, LightGuardian still achieves <5ms FCT. We also study the impact of the flow size on the average FCT (see Figure 14b in Appendix D.3).

Per-hop latency *v.s.* traffic load (Figure 6c): We find that LightGuardian has little impact on the network. We test the per-hop latency before and after deploying LightGuardian in the network. We vary the bandwidth usage from 0% to 90%, and measure the average per-hop latency of 10^4 packets using the `ping -f` instruction. The results show that at 0% traffic load, after deploying LightGuardian, per-hop latency increases $1.6\mu s$. At 90% load, per-hop latency increases at most $3.1\mu s$.

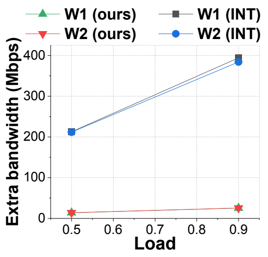
Reconstruction rate *v.s.* time (Figure 6d): We find that the sketches in LightGuardian can be quickly reconstructed. We use 90% of the total bandwidth and measure the reconstruction rate on each switch over time. The results show that under workload W1, the analyzer aggregates 90% sketchlets on the *edge switches*, the *aggregation switches*, and the *core switches* in 1.3, 1.7 and 2.1 seconds, respectively; and it aggregates 99% sketchlets in 2.1, 2.8 and 3.6 seconds, respectively. The results under workload W2 (as shown in Figure 14a in Appendix D.3) are similar. Other results related to sketch reconstruction are shown in Figure 14c-14d in Appendix D.3.

8.3 Simulations

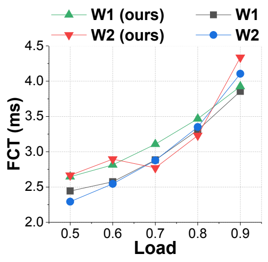
8.3.1 Simulations on Mininet

We evaluate LightGuardian's performance in locating blackholes, loops, and abnormal jitters through Mininet case studies. Our setup in Mininet consists of 16 hosts, 20 switches, and 48 links in a Fat-Tree topology. We only show the results as F1 scores. For more specific PR and RR results, please refer to Appendix D.2.

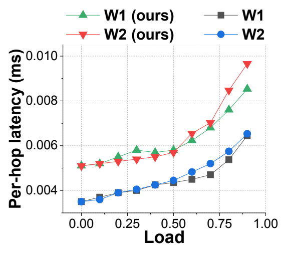
Locating blackholes (Figure 7a): We find that LightGuardian achieves high accuracy in locating blackholes. We randomly generate 10M packets belonging to 0.1M different flows. We create two blackholes by shutting down two links. And we reconstruct the sketchlets in a fixed time interval (5s) into sketches. For each flow, we query it in the reconstructed sketches to locate the culprit switches where the ratio $\frac{\mathcal{P}}{\mathcal{L}}$ is below a threshold. Here, for any switch, $\mathcal{P}$ is the estimated flow size, and $\mathcal{L}$ is the estimated flow size in the last-hop switch. The results show that when using 0.8MB of memory (2^{16} buckets), F1 score can reach 0.99.



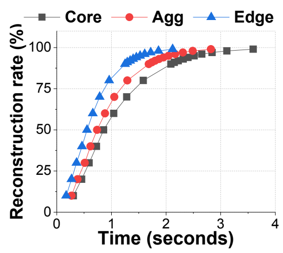
(a) Bandwidth *v.s.* load.



(b) FCT *v.s.* load.

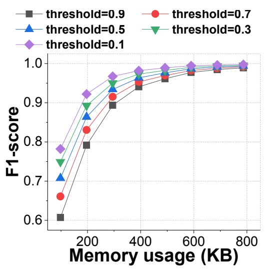


(c) Per-hop latency *v.s.* load.

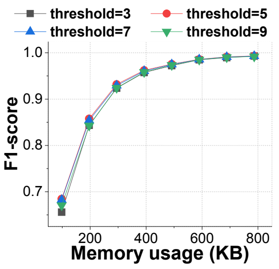


(d) Reconstruction *v.s.* time.

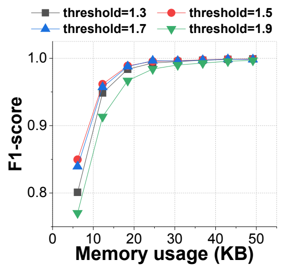
Figure 6: Experimental results on the testbed.



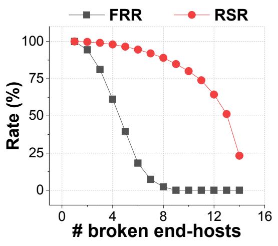
(a) Blackhole detection.



(b) Loop detection.



(c) Jitter detection.



(d) FRR/RSR *v.s.* λ_b.

Figure 7: Simulations results on Mininet.

Locating loops (Figure 7b): We find that LightGuardian achieves high accuracy in locating loops. We randomly generate 10M packets belonging to 0.1M different flows, and let 10% flows loop between two randomly selected adjacent switches. For each flow, we query it in the reconstructed sketches and locate the switches where $\frac{\varphi}{L}$ exceeds a threshold. The results show that when using 0.8MB of memory, F1 score reaches about 0.99.

Locating abnormal jitters (Figure 7c): We find that LightGuardian achieves high accuracy in locating abnormal jitters. We randomly generate 10M packets belonging to 10K different flows. To simulate jitters on the switch, we randomly choose two links, and split each of them into two parallel links with different speed. In this way, the flows passing through the slow link will suffer jitters, which leads to a sharp increase in their inter-arrival time in the next-hop switch. The results show that when using more than 20KB of memory, the F1 score is close to 1.

8.3.2 Simulations for Robustness

Next, we focus on the robustness of LightGuardian. The network topology here is the same as Mininet. We set $w = 2^{16}$. And in each experiment, we randomly select λ_b end-hosts and shut them down[9]. Then we observe how many sketches can be fully-reconstructed (recovered) in the global analyzer. The metrics we used here are: 1) *Full-Recovery Rate (FRR)*: the probability of recovering all sketches; 2) *Recovering-Sketch Rate (RSR)*: the ratio of the number of recovered sketches to the number of all sketches; From Figure 7d, we find that our system is robust to survive several device failures. When $\lambda_b = 4$, the FRR is still >60%. Even if half of the end-hosts break down ($\lambda_b = 8$), the analyzer still stands a chance of recovering all sketches (FRR > 0). And the RSR slowly decreases as λ_b increases. When $\lambda_b = 7$, the analyzer can reconstruct more than 90% sketches.

[9]Normal end-hosts still send packets to broken end-hosts, but broken end-hosts cannot send packets to others.

9 Conclusion and Future Work

In this paper, we present LightGuardian, a full-visibility, lightweight, in-band network telemetry system. LightGuardian designs the SuMax sketch to capture per-flow per-hop statistics on the programmable data plane, and use the constant-sized sketchlet to aggregate the statistics to any end-host, which can then perform both the device-local and the network-wide analysis. Experiments on a testbed and mininet simulations show that our system is able to perform 4 local measurement tasks, 3 network-wide tasks, and 3 anomalies locating tasks with high accuracy and consistently low overhead.

In the future work, we plan to design a mechanism to automatically adjust the system parameters according to the current traffic characteristics; we plan to conduct large-scale simulations; we plan to design and evaluate other methods of transferring sketches; we plan to offload the reconstruction and forwarding modules in end-host to smart NIC; we plan to deploy our system in cloud networking; and we also plan to use our measurement results to further improve the performance of congestion control, load balancing, and traffic scheduling.

Acknowledgment

We would like to thank the anonymous reviewers and our shepherd, Z. Morley Mao, for their thoughtful suggestions. This work is supported by National Natural Science Foundation of China (NSFC) (No. U20A20179), and the project of "FANet: PCL Future Greater-Bay Area Network Facilities for Large-scale Experiments and Applications" (No. LZC0019).

References

[1] Minlan Yu. Network telemetry: towards a top-down approach. *ACM SIGCOMM Computer Communication Review*, 49(1):11–17, 2019.

[2] Guohan Lu, Chuanxiong Guo, Yulong Li, Zhiqiang Zhou, Tong Yuan, Haitao Wu, Yongqiang Xiong, Rui Gao, and Yongguang Zhang. Serverswitch: a programmable and high performance platform for data center networks. In *Nsdi*, volume 11, pages 2–2, 2011.

[3] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Klaus-Tycho Förster, Arvind Krishnamurthy, and Thomas Anderson. Understanding and mitigating packet corruption in data center networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 362–375, 2017.

[4] Yunpeng James Liu, Peter Xiang Gao, Bernard Wong, and Srinivasan Keshav. Quartz: a new design element for low-latency dcns. *ACM SIGCOMM Computer Communication Review*, 44(4):283–294, 2014.

[5] Siva Kesava Reddy Kakarla, Alan Tang, Ryan Beckett, Karthick Jayaraman, Todd Millstein, Yuval Tamir, and George Varghese. Finding network misconfigurations by automatic template inference. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 999–1013, 2020.

[6] Cheng Tan, Ze Jin, Chuanxiong Guo, Tianrong Zhang, Haitao Wu, Karl Deng, Dongming Bi, and Dong Xiang. Netbouncer: Active device and link failure localization in data center networks. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 599–614, 2019.

[7] Teng Ma, Tao Ma, Zhuo Song, Jingxuan Li, Huaixin Chang, Kang Chen, Hai Jiang, and Yongwei Wu. X-rdma: Effective rdma middleware in large-scale production environments. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–12. IEEE, 2019.

[8] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Weicheng Sun. Pias: Practical information-agnostic flow scheduling for data center networks. In *Proceedings of the 13th ACM workshop on hot topics in networks*, pages 1–7, 2014.

[9] Li Chen, Justinas Lingys, Kai Chen, and Feng Liu. Auto: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 191–205, 2018.

[10] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM 2011 conference*, pages 350–361, 2011.

[11] Xin Wu, Daniel Turner, Chao-Chih Chen, David A Maltz, Xiaowei Yang, Lihua Yuan, and Ming Zhang. Netpilot: automating datacenter network failure mitigation. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 419–430, 2012.

[12] Qishi Wu, Sajjan Shiva, Sankardas Roy, Charles Ellis, and Vivek Datla. On modeling and simulation of game theory-based defense mechanisms against dos and ddos attacks. In *Proceedings of the 2010 spring simulation multiconference*, pages 1–8, 2010.

[13] JD Case, Mark Fedor, Martin Lee Schoffstall, and James Davin. Rfc1157: Simple network management protocol (snmp), 1990.

[14] Nick G Duffield and Matthias Grossglauser. Trajectory sampling for direct traffic observation. *IEEE/ACM transactions on networking*, 9(3):280–292, 2001.

[15] Vyas Sekar, Michael K Reiter, Walter Willinger, Hui Zhang, Ramana Rao Kompella, and David G Andersen. Csamp: a system for network-wide flow monitoring. 2008.

[16] Vyas Sekar, Michael K Reiter, and Hui Zhang. Revisiting the case for a minimalist approach for network flow monitoring. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 328–341, 2010.

[17] Junho Suh, Ted Taekyoung Kwon, Colin Dixon, Wes Felter, and John Carter. Opensample: A low-latency, sampling-based measurement platform for commodity sdn. In *2014 IEEE 34th International Conference on Distributed Computing Systems*, pages 228–237. IEEE, 2014.

[18] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[19] Sajad Shirali-Shahreza and Yashar Ganjali. Flexam: flexible sampling extension for monitoring and security applications in openflow. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 167–168, 2013.

[20] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y Zhao, et al. Packet-level telemetry in large datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 479–491, 2015.

[21] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 139–152. ACM, 2015.

[22] Hongyi Zeng, Ratul Mahajan, Nick McKeown, George Varghese, Lihua Yuan, and Ming Zhang. Measuring and troubleshooting large operational multipath networks with gray box testing. *Mountain Safety Res., Seattle, WA, USA, Rep. MSR-TR-2015-55*, 2015.

[23] Petr Lapukhov and Aijay Adams. Netnorad: Troubleshooting networks via end-to-end probing. https://engineering.fb.com/networking-traffic/netnorad-troubleshooting-networks-via-end-to-end-probing/, Febrary 2016.

[24] Amogh Dhamdhere, David D Clark, Alexander Gamero-Garrido, Matthew Luckie, Ricky KP Mok, Gautam Akiwate, Kabir Gogia, Vaibhav Bajpai, Alex C Snoeren, and Kc Claffy. Inferring persistent interdomain congestion. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 1–15, 2018.

[25] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1), 2005.

[26] Cristian Estan and George Varghese. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Transactions on Computer Systems (TOCS)*, 21(3), 2003.

[27] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *International Colloquium on Automata, Languages, and Programming*, pages 693–703. Springer, 2002.

[28] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*. ACM, 2016.

[29] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM SIGCOMM*, pages 561–575. ACM, 2018.

[30] Qun Huang, Patrick PC Lee, and Yungang Bao. Sketchlearn: relieving user burdens in approximate measurement with automated statistical inference. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 576–590. ACM, 2018.

[31] Pratanu Roy, Arijit Khan, and Gustavo Alonso. Augmented sketch: Faster and more accurate stream processing. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1449–1463, 2016.

[32] Tong Yang, Yang Zhou, Hao Jin, Shigang Chen, and Xiaoming Li. Pyramid sketch: A sketch framework for frequency estimation of data streams. *Proceedings of the VLDB Endowment*, 10(11):1442–1453, 2017.

[33] Xiaoqi Chen, Shir Landau-Feibish, Mark Braverman, and Jennifer Rexford. Beaucoup: Answering many network traffic queries, one memory update at a time. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 226–239, 2020.

[34] Jizhou Li, Zikun Li, Yifei Xu, Shiqi Jiang, Tong Yang, Bin Cui, Yafei Dai, and Gong Zhang. Wavingsketch: An unbiased and generic sketch for finding top-k items in data streams. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1574–1584, 2020.

[35] Tal Mizrahi, Gidi Navon, Giuseppe Fioccola, Mauro Cociglio, Mach Chen, and Greg Mirsky. Am-pm: Efficient network telemetry using alternate marking. *IEEE Network*, 33(4):155–161, 2019.

[36] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. Inband network telemetry via programmable dataplanes. In *ACM SIGCOMM*, 2015.

[37] Vimalkumar Jeyakumar, Mohammad Alizadeh, Yilong Geng, Changhoon Kim, and David Mazières. Millions of little minions: Using packets for low latency network programming and visibility. *ACM SIGCOMM Computer Communication Review*, 44(4):3–14, 2014.

[38] Cisco Nexus 9000 Series NX-OS Programmability Guide. https://www.cisco.com/c/en/us/td/docs/switches/datacenter/nexus9000/sw/92x/programmability/guide/b-cisco-nexus-9000-series-nx-os-programmability-guide-92x/b-cisco-nexus-9000-series-nx-os-programmability-guide-92x_chapter_0100001.html#id_95566.

[39] John Sonchack, Oliver Michel, Adam J Aviv, Eric Keller, and Jonathan M Smith. Scaling hardware accelerated network monitoring to concurrent and dynamic queries with* flow. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 823–835, 2018.

[40] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIG-*

COMM Workshop on Hot Topics in Networks, pages 1–6, 2010.

[41] P4 behavior model. https://github.com/p4lang/behavioral-model.

[42] Qun Huang, Xin Jin, Patrick PC Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. Sketchvisor: Robust network measurement for software packet processing. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 113–126. ACM, 2017.

[43] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, Shan Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research*, pages 164–176, 2017.

[44] Qun Huang and Patrick PC Lee. Ld-sketch: A distributed sketching design for accurate and scalable anomaly detection in network data streams. In *IEEE INFOCOM 2014-IEEE Conference on Computer Communications*, pages 1420–1428. IEEE, 2014.

[45] Yi Lu, Andrea Montanari, Balaji Prabhakar, Sarang Dharmapurikar, and Abdul Kabbani. Counter braids: a novel counter architecture for per-flow measurement. *ACM SIGMETRICS Performance Evaluation Review*, 36(1):121–132, 2008.

[46] Minlan Yu, Lavanya Jose, and Rui Miao. Software defined traffic measurement with opensketch. In *NSDI*, volume 13, 2013.

[47] Kai Sheng Tai, Vatsal Sharan, Peter Bailis, and Gregory Valiant. Sketching linear classifiers over data streams. In *Proceedings of the 2018 International Conference on Management of Data*, pages 757–772, 2018.

[48] Qun Huang, Siyuan Sheng, Xiang Chen, Yungang Bao, Rui Zhang, Yanwei Xu, and Gong Zhang. Toward nearly-zero-error sketching via compressive sensing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, April 2021.

[49] Zaoxing Liu, Samson Zhou, Ori Rottenstreich, Vladimir Braverman, and Jennifer Rexford. Memory-efficient performance monitoring on programmable switches with lean algorithms. In *Symposium on Algorithmic Principles of Computer Systems*, pages 31–44. SIAM, 2020.

[50] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Lossradar: Fast detection of lost packets in data center networks. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pages 481–495, 2016.

[51] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[52] Michael T Goodrich and Michael Mitzenmacher. Invertible bloom lookup tables. In *2011 49th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 792–799. IEEE, 2011.

[53] Junzhi Gong, Yuliang Li, Bilal Anwer, Aman Shaikh, and Minlan Yu. Microscope: Queue-based performance diagnosis for network functions. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 390–403, 2020.

[54] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Flowradar: A better netflow for data centers. In *NSDI*, 2016.

[55] Leo Cloutier. Apparatus and method for correcting jitter in data packets, August 4 1998. US Patent 5,790,543.

[56] Siu-Ping Chan, C-W Kok, and Albert K Wong. Multimedia streaming gateway with jitter detection. *IEEE Transactions on Multimedia*, 7(3):585–592, 2005.

[57] Tom McBeath. Method and apparatus for monitoring latency, jitter, packet throughput and packet loss ratio between two points on a network, June 14 2011. US Patent 7,961,637.

[58] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Distributed network monitoring and debugging with switchpointer. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 453–456, 2018.

[59] Ivan Morandi, Francesco Bronzino, Renata Teixeira, and Srikanth Sundaresan. Service traceroute: tracing paths of application flows. In *International Conference on Passive and Active Network Measurement*, pages 116–128. Springer, 2019.

[60] Feng Wang, Zhuoqing Morley Mao, Jia Wang, Lixin Gao, and Randy Bush. A measurement study on the impact of routing events on end-to-end internet path performance. *ACM SIGCOMM Computer Communication Review*, 36(4):375–386, 2006.

[61] Ying Zhang, Zhuoqing Morley Mao, and Jia Wang. A framework for measuring and predicting the impact of routing changes. In *IEEE INFOCOM 2007-26th IEEE International Conference on Computer Communications*, pages 339–347. IEEE, 2007.

[62] Kanak Agarwal, Eric Rozner, Colin Dixon, and John Carter. Sdn traceroute: Tracing sdn forwarding without changing network behavior. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 145–150, 2014.

[63] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Simplifying datacenter network debugging with pathdump. In *12th {USENIX} Symposium on Operat-*

ing Systems Design and Implementation ({OSDI} 16), pages 233–248, 2016.

[64] Jeff Rasley, Brent Stephens, Colin Dixon, Eric Rozner, Wes Felter, Kanak Agarwal, John Carter, and Rodrigo Fonseca. Planck: Millisecond-scale monitoring and control for commodity networks. *ACM SIGCOMM Computer Communication Review*, 44(4):407–418, 2014.

[65] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, pages 71–85, 2014.

[66] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. Confluo: Distributed monitoring and diagnosis stack for high-speed networks. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 421–436, 2019.

[67] Haoyu Song, Sarang Dharmapurikar, Jonathan Turner, and John Lockwood. Fast hash table lookup using extended bloom filter: an aid to network processing. *ACM SIGCOMM Computer Communication Review*, 35(4):181–192, 2005.

[68] Yifei Yuan, Dong Lin, Ankit Mishra, Sajal Marwaha, Rajeev Alur, and Boon Thau Loo. Quantitative network monitoring with netqre. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 99–112, 2017.

[69] Omid Alipourfard, Masoud Moshref, and Minlan Yu. Re-evaluating measurement algorithms in software. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, pages 1–7, 2015.

[70] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Trumpet: Timely and precise triggers in data centers. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 129–143, 2016.

[71] Xuemei Liu, Meral Shirazipour, Minlan Yu, and Ying Zhang. Mozart: Temporal coordination of measurement. In *Proceedings of the Symposium on SDN Research*, pages 1–12, 2016.

[72] Olivier Tilmans, Tobias Bühler, Ingmar Poese, Stefano Vissicchio, and Laurent Vanbever. Stroboscope: Declarative network monitoring on a budget. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 467–482, 2018.

[73] Benoit Claise. Cisco systems netflow version 9. Technical report, 2004.

[74] Peter Phaal, Sonia Panchen, and Neil McKee. Inmon corporation's sflow: A method for monitoring traffic in switched and routed networks. Technical report, 2001.

[75] Pavlos Nikolopoulos, Christos Pappas, Katerina Argyraki, and Adrian Perrig. Retroactive packet sampling for traffic receipts. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 3(1):1–39, 2019.

[76] Fangfan Li, Arian Akhavan Niaki, David Choffnes, Phillipa Gill, and Alan Mislove. A large-scale analysis of deployed traffic differentiation practices. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 130–144. 2019.

[77] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. Nitrosketch: Robust and general sketch-based monitoring in software switches. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 334–350. 2019.

[78] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *ACM SIGCOMM Computer Communication Review*, volume 45. ACM, 2015.

[79] Pietro Marchetta, Alessio Botta, Ethan Katz-Bassett, and Antonio Pescapé. Dissecting round trip time on the slow path with a single packet. In *International Conference on Passive and Active Network Measurement*, pages 88–97. Springer, 2014.

[80] Andreas Wundsam, Dan Levin, Srini Seetharaman, and Anja Feldmann. Ofrewind: Enabling record and replay troubleshooting for networks. In *USENIX Annual Technical Conference*, pages 327–340. USENIX Association, 2011.

[81] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, and Alex C Snoeren. Passive realtime datacenter fault detection and localization. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 595–612, 2017.

[82] Srinivasan Seshan, Mark Stemm, and Randy H Katz. Spand: Shared passive network performance discovery. In *USENIX Symposium on Internet Technologies and Systems*, pages 1–13, 1997.

[83] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 85–98, 2017.

[84] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 Conference of the ACM Special*

Interest Group on Data Communication, pages 357–371, 2018.

[85] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Dream: dynamic resource allocation for software-defined measurement. *ACM SIGCOMM Computer Communication Review*, 44(4), 2015.

[86] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Scream: Sketch resource allocation for software-defined measurement. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, pages 1–13, 2015.

[87] Qun Huang, Haifeng Sun, Patrick PC Lee, Wei Bai, Feng Zhu, and Yungang Bao. Omnimon: Re-architecting network telemetry with resource efficiency and full accuracy. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 404–421, 2020.

[88] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. Pint: Probabilistic in-band network telemetry. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 662–680, 2020.

[89] The open virtual switch website. `http://openvswitch.org`.

[90] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research*. ACM, 2017.

[91] Ran Ben-Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. Efficient measurement on programmable switches using probabilistic recirculation. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, pages 313–323. IEEE, 2018.

[92] Graham Cormode. Sketch techniques for approximate query processing. *Foundations and Trends in Databases. NOW publishers*, 2011.

[93] Abhishek Kumar, Minho Sung, Jun Xu, and Jia Wang. Data streaming algorithms for efficient and accurate estimation of flow size distribution. *ACM SIGMETRICS Performance Evaluation Review*, 32(1):177–188, 2004.

[94] Kyu-Young Whang, Brad T Vander-Zanden, and Howard M Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Transactions on Database Systems (TODS)*, 15(2):208–229, 1990.

[95] eBPF - Introduction, Tutorials Community Resources. `https://ebpf.io`.

[96] P4-16 Language Specification. `https://p4.org/p4-spec/docs/P4-16-v1.2.1.html#sec-checksums`.

[97] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. Inside the social network's (datacenter) network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 123–137, 2015.

[98] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 conference*, pages 63–74, 2010.

[99] The CAIDA Anonymized Internet Traces. `http://www.caida.org/data/overview/`.

[100] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, et al. Flow event telemetry on programmable data plane. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 76–89, 2020.

[101] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: Minimal near-optimal datacenter transport. *ACM SIGCOMM Computer Communication Review*, 43(4):435–446, 2013.

[102] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 221–235, 2018.

APPENDIX

A Algorithm

Algorithm 1: Insertion of SuMax sketch

```
Input: A new packet $\langle f, \alpha, \beta \rangle$.
$\omega \leftarrow +\infty$;
for $i = 0 \rightarrow d-1$ do
    if $\mathcal{A}_i^{sum}[\mathcal{H}_i(f)] + \alpha < \omega$ then
        $\omega \leftarrow \mathcal{A}_i^{sum}[\mathcal{H}_i(f)] + \alpha$ ;
        $\mathcal{A}_i^{sum}[\mathcal{H}_i(f)] \leftarrow \omega$;
    else if $\mathcal{A}_i^{sum}[\mathcal{H}_i(f)] < \omega$ then
        $\mathcal{A}_i^{sum}[\mathcal{H}_i(f)] \leftarrow \omega$;
    end
    $\mathcal{A}_i^{max}[\mathcal{H}_i(f)] \leftarrow \max\{\beta, \mathcal{A}_i^{max}[\mathcal{H}_i(f)]\}$;
end
```

B Proof of Theorem 6.1

Theorem B.1 *After receiving sketchlets with a ratio of θ, SuMax can report valid results with a ratio of $(1-(1-\theta)^d)$. Specifically, when the result is valid, the estimated flow size has the following error bounds.*

$$\Pr\left\{|\hat{n}_f - n_f| > \varepsilon\right\} < \frac{\left(\frac{m\theta}{w\varepsilon} + 1 - \theta\right)^d - (1-\theta)^d}{1-(1-\theta)^d},$$

where d and w are parameters of SuMax (see Table 2), n_f and $\hat{n}_f$ are the real and estimated flow size, and m is the number of inserted packets.

Proof B.1 *Let $\mathcal{A}_1[\mathcal{H}_1(f)], \cdots \mathcal{A}_d[\mathcal{H}_d(f)]$ be the values of d mapped buckets. Let $\mathcal{V}_1, \cdots, \mathcal{V}_d$ be d indicating random variables, where $\mathcal{V}_i$ indicates whether the i-th mapped bucket is received. Since the reported result of SuMax is valid as long as at least one buckets is received, the probability of acquiring a valid result is:*

$$\Pr\left\{\texttt{valid}\right\} = \Pr\left\{\mathcal{V}_1 = 1 \vee \cdots \vee \mathcal{V}_d = 1\right\}$$
$$= 1 - \prod_{i=1}^{d} \Pr\left\{\mathcal{V}_i = 0\right\} = 1-(1-\theta)^d$$

The expected number of packets mapped to each bucket is $\frac{m}{w} + n_f$. Since SuMax uses a conservative update method, not every packet increments the value of the bucket. Thus the value of each bucket satisfies:

$$\mathbb{E}\left(\mathcal{A}_i[\mathcal{H}_i(f)]\right) < \frac{m}{w} + n_f$$

According to the Markov inequality, we can derive that:

$$\Pr\left\{\left|\mathcal{A}_i[\mathcal{H}_i(f)] - n_f\right| > \varepsilon\right\} < \frac{\mathbb{E}\left(\mathcal{A}_i[\mathcal{H}_i(f)] - n_f\right)}{\varepsilon} < \frac{m}{w\varepsilon}$$

According to the total probability rule, we have

$$\Pr\left\{|\hat{n}_f - n_f| > \varepsilon\right\} = \sum_{i=1}^{d} \Pr\left\{\zeta_i\right\} \cdot \Pr\left\{|\hat{n}_f - n_f| > \varepsilon \mid \zeta_i\right\}$$
$$< \sum_{i=1}^{d} \frac{\binom{d}{i}\theta^i(1-\theta)^{d-i}}{1-(1-\theta)^d} \cdot \left(\frac{m}{w\varepsilon}\right)^i = \frac{\left(\frac{m\theta}{w\varepsilon} + 1 - \theta\right)^d - (1-\theta)^d}{1-(1-\theta)^d}$$

where ζ_i indicates that there are i valid buckets after the reconstruction process.

C Evaluation Metrics

1) Average Relative Error (ARE): $\frac{1}{|\Psi|}\sum_{f_i \in \Psi} \frac{|n_i - \hat{n}_i|}{n_i}$, where n_i is the real statistics of flow f_i, $\hat{n}_i$ is the estimated statistics of flow f_i, and Ψ is the flow set.
2) Recall Rate (RR): The ratio of the number of correctly reported instances to the number of all correct instances.
3) Precision Rate (PR): The ratio of the number of correctly reported instances to the number of all reported instances.
4) F_1 Score: $\frac{2 \times PR \times RR}{PR + RR}$.
5) Relative Error (RE): $\frac{|Est. - True|}{True}$, where $Est.$ and $True$ are the estimated and true statistics, respectively.
6) Weighted Mean Relative Error (WMRE) [42, 93]: $\frac{\sum_{i=1}^{z}|n_i - \hat{n}_i|}{\sum_{i=1}^{z}\frac{n_i + \hat{n}_i}{2}}$, where n_i and $\hat{n}_i$ are the real and estimated event probabilities respectively, and z is the number of events.

D Additional Experimental Results

D.1 Experiments on SuMax

Flow size estimation (Figure 11a): We find that the ARE of SuMax is higher to CM and close to CU. When using 768KB of memory, for the largest 500 flows, the ARE of SuMax is 17.1 times lower than CM and only 0.1 times higher than CU.
Memory overhead (Figure 11b): We find that the memory overhead of SuMax grows sub-linearly with the number of generated packets, which guarantees the scalability of LightGuardian. We conduct this experiment in the flow size estimation task. We vary the number of generated packets in the network, and record how much memory SuMax needs to achieve 0.01 ARE. Table 3 further studies the memory overhead of SuMax in various tasks.
Delay distribution (Figure 12a-12e): We find that for different datasets, SuMax always achieves performance similar to CU. Figure 12a-12b show that the WMRE of SuMax is lower than CM and close to CU, which means SuMax has a stable performance. Figure 12c-12e show that when using 6MB of memory and varying the top-k flows, the WMRE of SuMax is similar to CU and lower than CM.
Last arrival time (Figure 12f-12g): We find that when using 768KB of memory, the Average Absolute Error (AAE) is less than 12ms; and when using 3MB of memory, the AAE is less than 1.5ms. Figure 12g further illustrated that when using

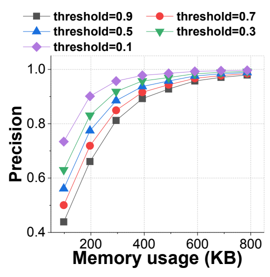
768KB of memory, the estimated results are absolutely correct for 87% packets.

Table 3: Memory usage of SuMax in various tasks.

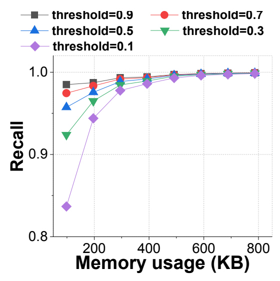
Task	Target error	Memory (MB)
Flow size estimation	0.01	~ 0.1
Flow size distribution	0.05	~ 0.4
Cardinality	0.005	~ 0.2
Entropy	0.001	~ 0.8
Delay distribution	0.05	~ 0.8
Max inter-arrival	0.01	~ 50
Last arrival time	0.01	~ 0.8

D.2 Simulations on Mininet

We demonstrate the specific PR and RR experimental results in § 8.3.



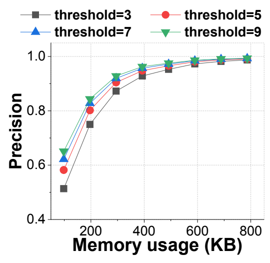
(a) PR *v.s.* memory usage.



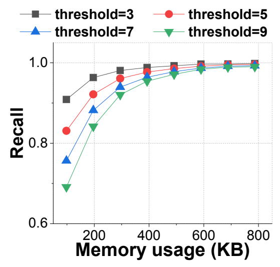
(b) RR *v.s.* memory usage.

Figure 8: Accuracy of locating blackholes.

Locating blackholes (Figure 8a-8b): We find that LightGuardian achieve high accuracy in locating blackholes. The results show that higher threshold goes with lower PR and higher RR. When using 800KB of memory (2^{16} buckets), the PR and RR reach $0.982 \sim 0.997$ and $0.998 \sim 0.999$ respectively.



(a) PR *v.s.* memory usage.

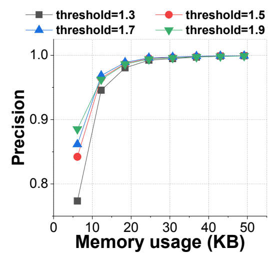


(b) RR *v.s.* memory usage.

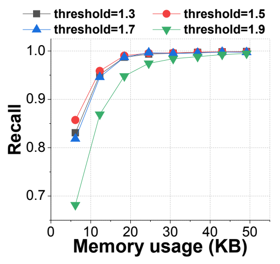
Figure 9: Accuracy of locating loops.

Locating loops (Figure 9a-9b): We find that LightGuardian achieve high accuracy in locating loops. The results show that higher threshold goes with higher PR and lower RR. When using 800KB of memory, the PR and RR reaches $0.988 \sim 0.994$ and $0.993 \sim 0.998$ respectively.

Locating abnormal jitters (Figure 10a-10b): We find that LightGuardian achieve high accuracy in locating jitters. The results show that higher threshold goes with higher PR and



(a) PR *v.s.* memory usage.



(b) RR *v.s.* memory usage.

Figure 10: Accuracy of locating abnormal jitters.

lower RR. When using more than 500KB of memory, both PR and RR are close to 1.0. When using 50KB of memory (2^{12} buckets), the PR and RR reach $0.998 \sim 0.999$ and $0.994 \sim 0.999$ respectively.

D.3 Testbed Experiments

We further extend the experiments in § 8.2.

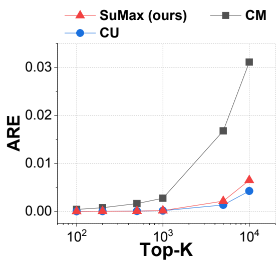
Bandwidth overhead *v.s.* λ_c (Figure 13a): We find that the bandwidth overhead of LightGuardian can be dynamically adjusted by the carrying probability λ_c. We vary the carrying probability λ_c from $\frac{1}{64}$ to $\frac{8}{64}$, and measure the bandwidth overhead. The results show that compared with INT, LightGuardian only uses 1.5% to 12.4% bandwidth. When the average packet size becomes smaller (*e.g.*, when encountering DDos attacks) and the number of the packets increases, our LightGuardian can adjust the bandwidth overhead by reducing λ_c. INT does not have this ability.

Required time (RT) *v.s.* λ_c (Figure 13b): We find that the time required to construct the sketches can be dynamically adjusted by the carrying probability λ_c. We generate 36Gbps traffic between two end-hosts in the same rack, and vary λ_c from $\frac{1}{64}$ to $\frac{8}{64}$. The results show that as λ_c increases, the required time decreases.

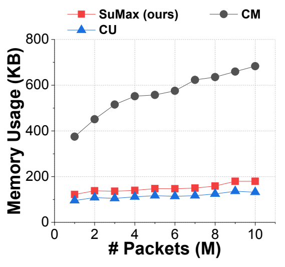
Required packets (RP) *v.s.* λ_c (Figure 13c): We find that the packets required to reconstruct the sketches can be dynamically adjusted by the carrying probability λ. The results show that the packets required to aggregate 90% and 99% sketchlets is negatively correlated to λ_c.

FCT *v.s.* flow size (Figure 14b): We find that LightGuardian has little impact on the FCT for the flows of any size. We measure the average FCT of flows of different sizes under 90% traffic load. We divide the flows into five groups according to their sizes: (0, 0.01MB), (0.01, 0.1MB), (0.1MB, 1MB), (1MB, 10MB) and (10MB, 100MB), and calculate the average FCT of each group. The results show that even for the flows of 100MB, the average FCT is no more than 40ms.

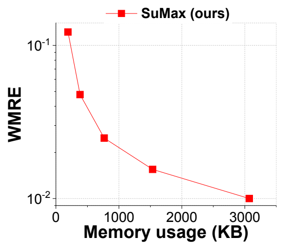
RP/RT *v.s.* w (Figure 14c-14d): We find that the required packets and the required time to reconstruct the sketches can be dynamically adjusted by w. We vary the number of sketchlets on the TOR switch from 2^{12} to 2^{16} and measure the number of the required packets and the required time to achieve certain reconstruction rates. The results show that both the required time and the required packets grow linearly with the number of sketchlets.



(a) Flow size estimation.

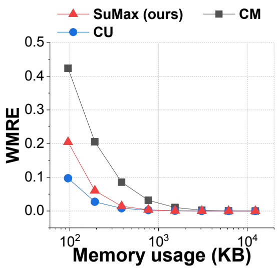


(b) Memory overhead.

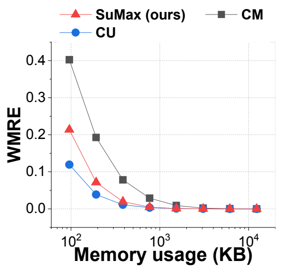


(c) Flow size distribution.

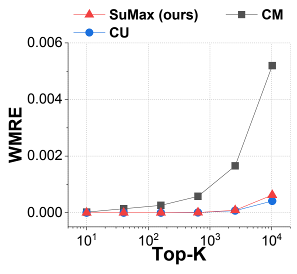
Figure 11: Experimental results of the SuMax sketch in device-local tasks.



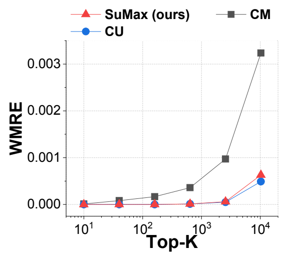
(a) Distribution (Union).



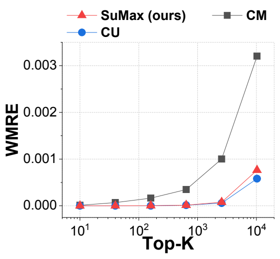
(b) Distribution (Mixed).



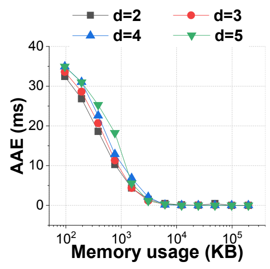
(c) Distribution (χ^2).



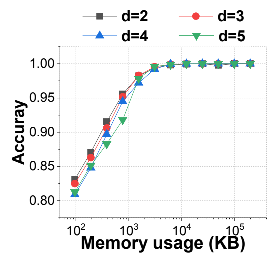
(d) Distribution (Union).



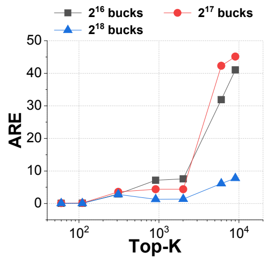
(e) Distribution (Mixed).



(f) Last arrival time.

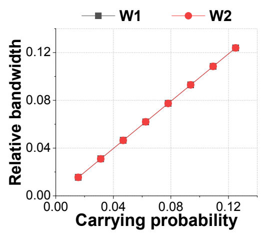


(g) Last arrival time.

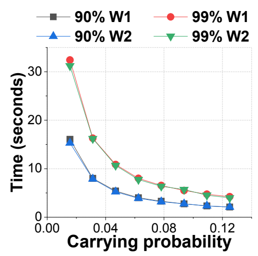


(h) Max inter-arrival time.

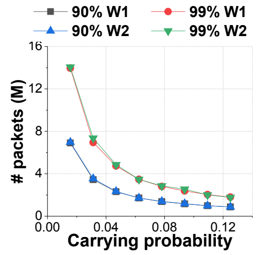
Figure 12: Experimental results of the SuMax sketch in network-wide tasks.



(a) Bandwidth *v.s.* λ_c.

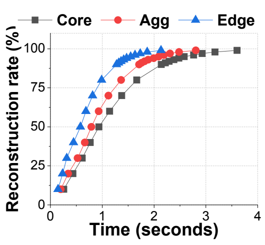


(b) RT *v.s.* λ_c.

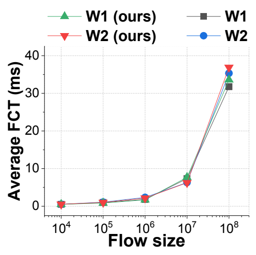


(c) RP *v.s.* λ_c.

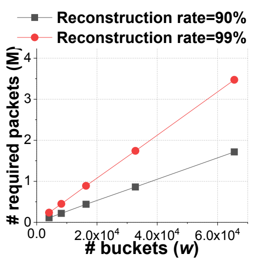
Figure 13: Impact of carrying probability λ_c.



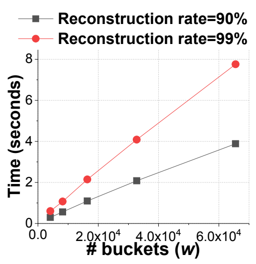
(a) Reconstruction (W2).



(b) FCT *v.s.* flow size.



(c) RP *v.s.* *w*.



(d) RT *v.s.* *w*.

Figure 14: Experimental results on the testbed.

Fast and Light Bandwidth Testing for Internet Users

Xinlei Yang[1*], Xianlong Wang[1*], Zhenhua Li[1], Yunhao Liu[1]
Feng Qian[2], Liangyi Gong[1], Rui Miao[3], Tianyin Xu[4]
[1]*Tsinghua University* [2]*University of Minnesota* [3]*Alibaba Group* [4]*UIUC*

Abstract

Bandwidth testing measures the access bandwidth of end hosts, which is crucial to emerging Internet applications for network-aware content delivery. However, today's bandwidth testing services (BTSes) are slow and costly—the tests take a long time to run, consume excessive data usage at the client side, and/or require large-scale test server deployments. The inefficiency and high cost of BTSes root in their methodologies that use excessive temporal and spatial redundancies for combating noises in Internet measurement.

This paper presents FastBTS to make BTS fast and cheap while maintaining high accuracy. The key idea of FastBTS is to accommodate and exploit the noise rather than repetitively and exhaustively suppress the impact of noise. This is achieved by a novel statistical sampling framework (termed *fuzzy rejection sampling*). We build FastBTS as an end-to-end BTS that implements fuzzy rejection sampling based on elastic bandwidth probing and denoised sampling from high-fidelity windows, together with server selection and multi-homing support. Our evaluation shows that with only 30 test servers, FastBTS achieves the same level of accuracy compared to the state-of-the-art BTS (`SpeedTest.net`) that deploys ∼12,000 servers. Most importantly, FastBTS makes bandwidth tests 5.6× faster and 10.7× more data-efficient.

1 Introduction

Access link bandwidth of Internet users commonly constitutes the bottleneck of Internet content delivery, especially for emerging applications like AR/VR. In traditional residential broadband networks, the access bandwidth is largely stable and matches ISPs' service plans [9, 14, 15]. In recent years, however, it becomes less transparent and more dynamic, driven by virtual network operators (VNOs), user mobility, and infrastructure dynamics [21].

To effectively measure the access bandwidth, bandwidth testing services (BTSes) have been widely developed and deployed. BTSes serve as a core component of many applications that conduct network-aware content delivery [1, 31]. BTSes' data are cited in government reports, trade press [37], and ISPs' advertisements [29]; they play a key role in ISP customers' decision making [39]. During COVID-19, BTSes are top "home networking tips" to support telework [11, 12]. The following lists a few common use cases of BTSes:

[*] Co-primary authors. Zhenhua Li is the corresponding author.

- VNO has been a popular operation model that resells network services from base carrier(s). The shared nature of VNOs and their complex interactions with the base carriers make it challenging to ensure service qualities [69, 72, 78]. Many ISPs and VNOs today either build their own BTSes [2], or recommend end users to use public BTSes. For example, `SpeedTest.net`, a popular BTS, serves more than 500M unique visitors per year [4].
- Wireless access is becoming ubiquitous, exhibiting heterogeneous and dynamic performance. To assist users to locate good coverage areas, cellular carriers offer "performance maps" [16], and several commercial products (*e.g.,* WiFiMaster used by 800M mobile devices [31]) employ crowd-sourced measurements to probe bandwidth.
- Emerging bandwidth-hungry apps (*e.g.,* UHD videos and VR/AR), together with bandwidth-fluctuating access networks (*e.g.,* 5G), make BTSes an integral component of modern mobile platforms. For example, the newly released Android 11 provides 5G apps with a bandwidth estimation API that offers "a rough guide of the expected peak bandwidth for the first hop of the given transport [20]."

Most of today's BTSes work in three steps: (1) setup, (2) bandwidth probing, and (3) bandwidth estimation. During the setup process, the user client measures its latency to a number of candidate test servers and selects one or more servers with low latency. Then, it probes the available bandwidth by uploading and downloading large files to and from the test server(s) and records the measured throughput as samples. Finally, it estimates the overall downlink/uplink bandwidth.

The key challenge of BTSes is to deal with *noises* of Internet measurements incurred by congestion control, link sharing, *etc.* Spatially, the noise inflates as the distance (the routing hop count) increases between the user client and test server. Temporally, the throughput samples may be constantly fluctuating over time—the shorter the test duration is, the severer impact on throughput samples the noise can induce. An effective BTS needs to accurately and efficiently measure the access bandwidth from noisy throughput samples.

Today's BTSes are slow and costly. For example, a 5G bandwidth test using `SpeedTest.net` for a 1.15 Gbps downlink takes 15 seconds of time and incurs 1.94 GB of data usage on end users in order to achieve satisfying test accuracy. To deploy an effective BTS, hundreds to thousands of test servers are typically needed. Such a level of cost (both at the client and server sides) and long test duration prevent BTSes from being a foundational, ubiquitous Internet service for high-speed, metered networks. Based on our measurements and reverse engineering of 20 commercial BTSes (§2), we find that the inefficiency and cost of these BTSes fundamentally root in their methodology of relying on temporal and/or spatial redundancy to deal with noises:

- Temporally, most BTSes rely on a *flooding-based* bandwidth probing approach, which simply injects an excessive number of packets to ensure that the bottleneck link is saturated by test data rather than noise data. Also, their test processes often intentionally last for a long time to ensure the convergence of the probing algorithm.
- Spatially, many BTSes deploy dense, redundant test servers close to the probing client, in order to avoid "long-distance" noises. For example, `FAST.com` and `SpeedTest.net` deploy $\sim$1,000 and $\sim$12,000 geo-distributed servers, respectively, while WiFiMaster controversially exploits a large Internet content provider's CDN server pool.

In this paper, we present FastBTS to make BTS fast and cheap while maintaining high accuracy. Our key idea is to accommodate and exploit the noise through a novel statistical sampling framework, which eliminates the need for long test duration and exhaustive resource usage for suppressing the impact of noise. Our insight is that the workflow of BTS can be modeled as a process of *acceptance-rejection sampling* [43] (or *rejection sampling* for short). During a test, a sequence of throughput samples are generated by bandwidth probing and exhibit a measured distribution $P(x)$, where x denotes the throughput value of a sample. They are filtered by the bandwidth estimation algorithm, in the form of an acceptance-rejection function (ARF), which retains the accepted samples and discards the rejected samples to model the target distribution $T(x)$ for calculating the final test result.

The key challenge of FastBTS is that $T(x)$ cannot be known beforehand. Hence, we cannot apply traditional rejection sampling algorithm that assumes a $T(x)$ and uses it as an input. In practice, our extensive measurement results show that, while the noise samples are scattered across a wide throughput interval, the true samples tend to concentrate within a narrow throughput interval (termed as a *crucial interval*). Therefore, one can reasonably model $T(x)$ using the crucial interval, as long as $T(x)$ is persistently covered by $P(x)$. We name the above-described technique *fuzzy rejection sampling*.

FastBTS implements fuzzy rejection sampling with the architecture shown in Figure 1. First, it narrows down $P(x)$ as the boundary of $T(x)$ to bootstrap $T(x)$ modeling. This is done by an Elastic Bandwidth Probing (EBP) mechanism to tune the transport-layer data probing rate based on its deviation from the currently-estimated bandwidth. Second, we design a Crucial Interval Sampling (CIS) algorithm, acting as the ARF, to efficiently calculate the optimal crucial interval with throughput samples (*i.e.,* performing denoised sampling from high-fidelity throughput windows). Also, the Data-driven Server Selection (DSS) and Adaptive Multi-Homing (AMH) mechanisms are used to establish multiple parallel connections with different test servers when necessary. DSS and AMH together can help saturate the access link, so that $T(x)$ can be accurately modeled in a short time, even when the access bandwidth exceeds the capability of each test server.

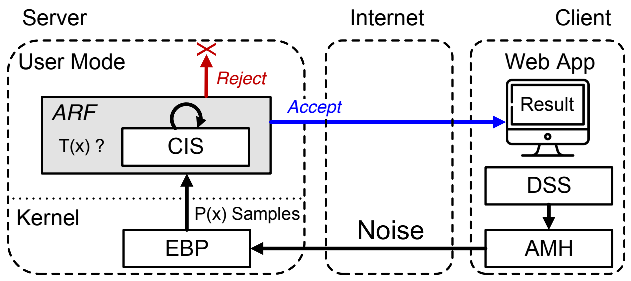


Figure 1: An architectural overview of FastBTS. The arrows show the workflows of a bandwidth test in FastBTS.

We have built FastBTS as an end-to-end BTS, consisting of the FastBTS app for clients, and a Linux kernel module for test servers. We deploy the FastBTS backend using 30 geo-distributed budget servers, and the FastBTS app on 100+ diverse client hosts. Our key evaluation results are[1]:

- On the same testbed, FastBTS yields 5%–72% higher average accuracy than the other BTSes under diverse network scenarios (including 5G), while incurring 2.3–8.5$\times$ shorter test duration and 3.7–14.2$\times$ less data usage.
- Employing only 30 test servers, FastBTS achieves comparable accuracy compared with the production system of `SpeedTest.net` with $\sim$12,000 test servers, while incurring 5.6$\times$ shorter test duration and 10.7$\times$ less data usage.
- FastBTS flows incur little ($<$6%) interference to concurrent non-BTS flows—EBP only ramps up fast when the data rate is well below the available bandwidth; it slowly grows the data rate when it is about to hit the bottleneck bandwidth.

To benefit the community, we have released all the source code at `https://FastBTS.github.io` and an online prototype system at `http://FastBTS.thucloud.com`.

2 Understanding State-of-The-Art BTSes

2.1 Methodology

We measure a BTS using the following metrics: (1) *Test Accuracy* measures how well the result (r) reported by a BTS matches the ground-truth bandwidth R. We calculate the accuracy as $\frac{r}{R}$. In practice, we observe that all BTSes (including FastBTS) tend to underestimate the bottleneck bandwidth due to factors like TCP slow start and congestion control, so the accuracy values are less than 1.0. (2) *Test Duration* measures

[1]In this work, we focus on the downlink bandwidth test due to its importance to a typical Internet user compared to the uplink.

the time needed to perform a bandwidth test—from starting a bandwidth test to returning the test result. (3) *Data Usage* measures the consumed network traffic for a test. This metric is of particular importance to metered LTE and 5G links.

Obtaining ground truth. Measuring test accuracy requires ground-truth data. However, it is challenging to know all the ground-truth bandwidths for large measurements. We use best possible estimations for different types of access links:

- *Wired LANs for in-lab experiments.* We regard the (known) physical link bandwidth, with the impact of (our injected) cross traffic properly considered, as the ground truth.
- *Commercial residential broadband and cloud networks.* We collect the bandwidth claimed by the ISPs or cloud service providers from the service contract, denoted as T_C. We then verify T_C by conducting long-lived bulk data transfers (average value denoted as T_B) before and after a bandwidth test. In more than 90% of our experiments, T_B and T_C match, with their difference being less than 5%; thus, we regard T_C as the ground truth. Otherwise, we choose to use T_B.
- *Cellular networks (LTE and 5G).* Due to a lack of T_C and the high dynamics of cellular links, we leverage the results provided by SpeedTest.net as a baseline reference. Being the state-of-the-art BTS that owns a massive number of (∼12,000) test servers across the globe, SpeedTest.net's results are widely considered as a close approximation to the ground-truth bandwidth [33, 36, 38, 41, 50, 73].

2.2 Analyzing Deployed BTSes

We study 20 deployed BTSes, including 18 widely-used, web-based BTSes and 2 Android 11 BTS APIs.[2] We run the 20 BTSes on three different PCs and four different smartphones listed in Table 1 (WiFiMaster and Android APIs are only run on smartphones). To understand the implementation of these BTSes, we jointly analyze: (1) the network traffic (recorded during each test), (2) the client-side code, and (3) vendors' documentation. A typical analysis workflow is as follows. We first examine the network traffic to reveal which server(s) the client interacts with during the test, as well as their interaction durations. We then inspect the captured HTTP(S) transactions to interpret the client's interactions with the server(s) such as server selection and file transfer. We also inspect client-side code (typically in JavaScript). However, this attempt may not always succeed due to code obfuscation used by some BTSes like SpeedTest. In this case, we use the Chrome developer tool to monitor the entire test process in the debug mode.

[2]The 18 web-based BTSes are ATTtest [2], BWP [5], CenturyLink [6], Cox [7], DSLReports [8], FAST [10], NYSbroadband [17], Optimum [19], SFtest [13], SpeakEasy [22], Spectrum [23], SpeedOf [24], SpeedTest [25], ThinkBroadband [28], Verizon [30], Xfinity [32], XYZtest [26], and WiFi-Master [31]. They are selected based on Alexa ranks and Google page ranks. In addition, we also study two BTS APIs in Android 11: `getLinkDownstreamBandwidthKbps` and `testMobileDownload`.

Device	Location	Network	Ground Truth
PC-1	U.S.	Residential broadband	100 Mbps
PC-2	Germany	Residential broadband	100 Mbps
PC-3	China	Residential broadband	100 Mbps
Samsung GS9	U.S.	LTE (60Mhz/1.9Ghz)	60–100 Mbps
Xiaomi XM8	China	LTE (40Mhz/1.8Ghz)	58–89 Mbps
Samsung GS10	U.S.	5G (400Mhz/28Ghz)	0.9–1.2 Gbps
Huawei HV30	China	5G (160Mhz/2.6Ghz)	0.4–0.7 Gbps

Table 1: Client devices used for testing the 20 BTSes. The test results are obtained from SpeedTest.net.

With the above efforts, we are able to "reverse engineer" the implementations of all the 20 BTSes.

Our analysis shows that a bandwidth test in these BTSes is typically done in three phases: (1) setup, (2) bandwidth probing, and (3) bandwidth estimation. In the setup phase, the BTS sends a list of candidate servers (based on the client's IP address or geo-location) to the client who then PINGs each candidate server over HTTP(S). Next, based on the servers' PING latency, the client selects one or more candidate servers to perform file transfer(s) to collect throughput samples. The BTS processes the samples and returns the result to the user.

2.3 Measurement Results

We select 9 (out of 20) representative BTSes for more in-depth characterizations, as listed in Table 2. These 9 selected BTSes well cover different designs (in terms of the key bandwidth test logic) of the remaining 11 ones. We deploy a large-scale testbed to comprehensively profile 8 representative BTSes, except Android API-A (we will discuss it separately). Our testbed is deployed on 108 geo-distributed VMs from multiple public cloud services providers (CSPs, including Azure, AWS, Ali Cloud, Digital Ocean, Vultr, and Tencent Cloud) as the client hosts. Note that we mainly employ VMs as client hosts because they are globally distributed and easy to deploy. Per their service agreements, the CSPs offer three types of access link bandwidths: 1 Mbps, 10 Mbps, and 100 Mbps (36 VMs each). The ground truth in Figure 2c is obtained according to the methodology in §2.1. We denote one *test group* as using one VM to run back-to-back bandwidth tests across all the 8 BTSes in a random order. We perform in one day 3,240 groups of tests, *i.e.,* 108 VMs × 3 different time-of-day (0:00, 8:00, and 16:00) × 10 repetitions.

We summarize our results in Table 2. We discover that all but one of the BTSes adopt *flooding-based* approaches to combat the test noises from a temporal perspective, leading to enormous data usage. Meanwhile, they differ in many aspects: (1) bandwidth probing mechanism, (2) bandwidth estimation algorithm, (3) connection management strategy, (4) server selection policy, and (5) server pool size.

2.4 Case Studies

We present our case studies of five major BTSes with the largest user bases selected from Table 2.

BTS	# Servers	Bandwidth Test Logic	Duration	Accuracy (Testbed / 5G)	Data Usage (Testbed / 5G)
TBB*	12	average throughput in all connections	8 s	0.59 / 0.31	42 MB / 481 MB
SpeedOf	116	average throughput in the last connection	8–230 s	0.76 / 0.22	61 MB / 256 MB
BWP	18	average throughput in the fastest connection	13 s	0.81 / 0.35	74 MB / 524 MB
SFtest	19	average throughput in all connections	20 s	0.89 / 0.81	194 MB / 2,013 MB
ATTtest	75	average throughput in all connections	15–30 s	0.86 / 0.53	122 MB / 663 MB
Xfinity	28	average all throughput samples	12 s	0.82 / 0.67	107 MB / 835 MB
FAST	∼1,000	average stable throughput samples	8–30 s	0.80 / 0.72	45 MB / 903 MB
SpeedTest	∼12,000	average refined throughput samples	15 s	0.96 / 0.92	150 MB / 1,972 MB
Android API-A	0	directly calculate using system configs	< 10 ms	NA / 0.09	0 / 0

Table 2: A brief summary of the 9 representative BTSes. “Testbed” and “5G” denote the large-scale cloud-based testbed and the 5G scenario, respectively. * means that WiFiMaster and Andriod API-B share the similar bandwidth test logic with ThinkBroadBand (TBB).

ThinkBroadBand [28]. The ThinkBroadBand BTS first selects a test server with the lowest latency to the client among its server pool. Then, it starts an 8-second bandwidth test by delivering a 20-MB file towards the client; if the file transfer takes less than 8 seconds, the test is repeated to collect more data points. After the 8 seconds, it calculates the average throughput (*i.e.,* data transfer rate) during the whole test process as the estimated bandwidth.

WiFiMaster [31]. WiFiMaster’s BTS is largely the same as that of ThinkBroadBand. The main difference lies in the test server pool. Instead of deploying a dedicated test server pool, WiFiMaster exploits the CDN servers of a large Internet content provider (Tencent) for frequent bandwidth tests. It directly downloads fixed-size (∼47 MB) software packages as the test files and measures the average download speed as the estimated bandwidth.

Our measurements show that the accuracy of ThinkBroadBand and WiFiMaster is low. The accuracy is merely 0.59, because the single HTTP connection during the test can easily be affected by network spikes and link congestion which lead to significant underestimation. In addition, using the average throughput for bandwidth estimation cannot rule out the impact of slow start and thus requires a long test duration.

Android APIs [1,3]. To cater to the needs of bandwidth estimation for bandwidth-hungry apps (*e.g.,* UHD videos and VR/AR) over 5G, Android 11 offers two “Bandwidth Estimator” APIs to “make it easier to check bandwidth for uploading and downloading content [1]”.

API-A, `getLinkDownstreamBandwidthKbps`, statically calculates the access bandwidth by “taking into account link parameters (radio technology, allocated channels, *etc.*) [20]”. It uses a pre-defined dictionary (`KEY_BANDWIDTH_STRING_ARRAY`) to map device hardware information to bandwidth values. For example, if the end-user’s device is connected to the new-radio non-standalone mmWave 5G network, API-A searches the dictionary which records `NR_NSA_MMWAVE:145000,60000`, indicating that the downlink bandwidth is 145,000 Kbps and the uplink bandwidth is 60,000 Kbps. This API provides a static “start-up on idle” estimation [1]. We test the performance of API-A in a similar manner as introduced in §2.3 with the 5G phones in Table 1. The results show that API-A bears rather poor accuracy (0.09) in realistic scenarios.

API-B, `testMobileDownload`, works in a similar way as ThinkBroadBand. It requires the app developer to provide the test servers and the test files.

FAST [10] is an advanced BTS with a pool of about 1,000 test servers. It employs a two-step server selection process: the client first picks five nearby servers based on its IP address, and then PINGs these five candidates to select the latency-wise nearest server for the bandwidth probing phase.

FAST progressively increases the concurrency according to the client network condition during the test. The client starts with a 25-MB file over a single connection. When the throughput reaches 0.5 Mbps, a new connection is created to transfer another 25-MB file. Similarly, at 1 Mbps, a third connection is established. For each connection, when the file transfer completes, it repeatedly requests another 25-MB file (the concurrency level never decreases).

FAST estimates the bandwidth as follows. As shown in Figure 2a, it collects a throughput sample every 200 ms, and maintains a 2-second window consisting of 10 most recent samples. After 5 seconds, FAST checks whether the in-window throughput samples are stable: $S_{max} - S_{min} \leq 3\% \cdot S_{avg}$, where S_{max}, S_{min}, and S_{avg} correspond to the maximum, minimum, and average value across all samples in the window, respectively. If the above inequality holds, FAST terminates the test and returns S_{avg}. Otherwise, the test will continue until reaching a time limit of 30 seconds; at that time, the last 2-second window’s S_{avg} will be returned to the user.

Unfortunately, our results show that the accuracy of FAST is still unsatisfactory. The average accuracy is 0.80, as shown in Table 2 and Figure 2c. We ascribe this to two reasons: (1) Though FAST owns ∼1,000 servers, they are mostly located in the US and Canada. Thus, FAST can hardly assign a nearby server to clients outside North America. In fact, FAST achieves relatively high average accuracy (0.92) when serving the clients in North America; however, it has quite low accuracy (0.74) when measuring the access link bandwidth of the clients in other places around the world. (2) We observe that FAST’s window-based mechanism for early generation of the test result is vulnerable to throughput fluctuations. Under unstable network conditions, FAST can only use throughput samples in the last two seconds (rather than the entire

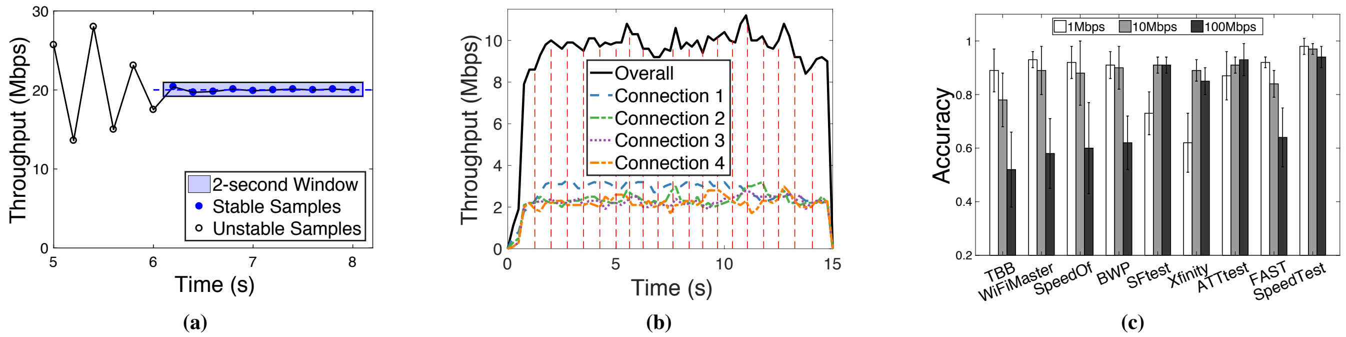


Figure 2: (a) Test logic of FAST. (b) Test logic of SpeedTest. (c) Test accuracy of nine commercial BTSes.

30-second samples) to calculate the test result.

SpeedTest [25] is considered the most advanced industrial BTS [33, 36, 41, 50, 73]. It deploys a pool of ∼12,000 servers. Similar to FAST, it also employs the two-step server selection process: it identifies 10 candidate servers based on the client's IP address, and then selects the latency-wise nearest from them. It also progressively increases the concurrency level: it begins with 4 parallel connections for quickly saturating the available bandwidth, and establishes a new connection at 25 Mbps and 35 Mbps, respectively. It uses a fixed file size of 25 MB and a fixed test duration of 15 seconds.

SpeedTest's bandwidth estimation algorithm is different from FAST's. During the bandwidth probing phase, it collects a throughput sample every 100 ms. Since the test duration is fixed to 15 seconds, all the 150 samples are used to construct 20 slices, each covering the same traffic volume, illustrated as the area under the throughput curve in Figure 2b. Then, 5 slices with the lowest average throughput and 2 slices with the highest average throughput are discarded. This leaves 13 slices remaining, whose average throughput is returned as the final test result. This method may help mitigate the impact of throughput fluctuations, but the two fixed thresholds for noise filtering could be deficient under diverse network conditions.

Overall, SpeedTest exhibits the highest accuracy (0.96) among the measured BTSes. A key contributing factor is its large server pool, as shown in §5.2.

3 Design of FastBTS

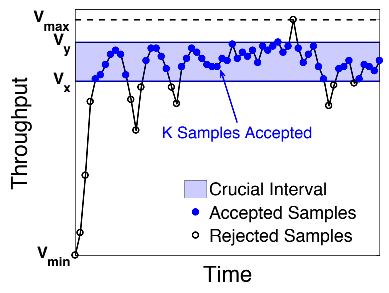
FastBTS is a fast and lightweight BTS with a fundamentally new design. FastBTS *accommodates and exploits* noises (instead of suppressing them) to significantly reduce the resource footprint and accelerate the tests, while retaining high test accuracy. The key technique of FastBTS is *fuzzy rejection sampling* which automatically identifies true samples that represent the target distribution and filters out false samples due to measurement noises, without apriori knowledge of the target distribution. Figure 1 shows the main components of FastBTS and the workflow of a bandwidth test.

- *Crucial Interval Sampling (CIS)* implements the acceptance rejection function of fuzzy rejection sampling. CIS is built upon a key observation based on our measurement study (see §2.3 and §2.4): *while the noise samples may be widely scattered, the desired bandwidth samples tend to concentrate within a narrow throughput interval*. CIS searches for a *dense* and *narrow* interval that covers the majority of the desirable samples, and uses computational geometry to drastically reduce the searching complexity.
- *Elastic Bandwidth Probing (EBP)* generates throughput samples that persistently[3] obey the distribution of the target bandwidth. We design EBP by optimizing BBR's bandwidth estimation algorithm [42] – different from BBR's static bandwidth probing policy, EBP reaches the target bandwidth much faster, while being non-disruptive.
- *Data-driven Server Selection (DSS)* selects the server(s) with the highest bandwidth estimation(s) through a data-driven model. We show that a simple model can significantly improve server selection results compared to the de-facto approach that ranks servers by round-trip time.
- *Adaptive Multi-Homing (AMH)* adaptively establishes multiple parallel connections with different test servers. AMH is important for saturating the access link when the last-mile access link is not the bottleneck, *e.g.,* 5G [67].

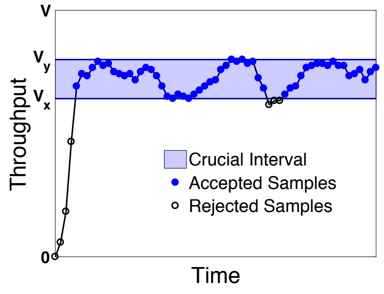
3.1 Crucial Interval Sampling (CIS)

CIS is designed based on the key observation: while noise samples are scattered across a wide throughput interval, the desirable samples tend to concentrate within a narrow interval, referred to as the *crucial interval*. As shown in Figure 3, in each subfigure, although the crucial interval is narrow, it can cover the vast majority of the desirable samples. Thus, while the target distribution $T(x)$ is unknown, we can approximate $T(x)$ with the crucial interval. Also, as more noise samples accumulate, the test accuracy would typically increase as

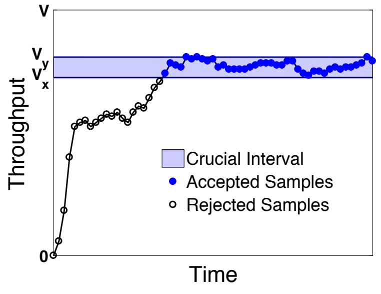
[3]Here "persistently" means that a certain set (or range) of samples constantly recur to the measurement data during the test process [45].



(a) SpeedTest

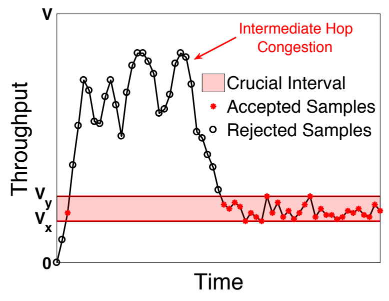


(b) ATTtest, BWP, XFinity, Android API-B

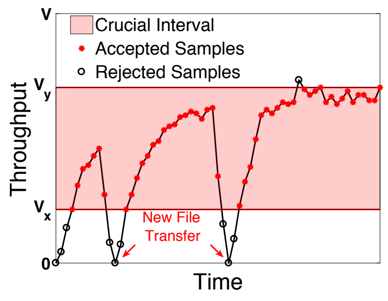


(c) FAST

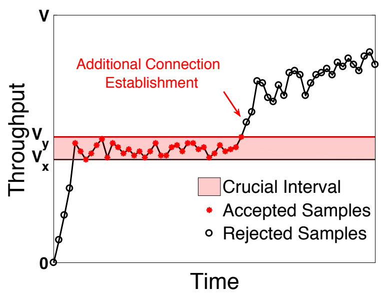
Figure 3: Common scenarios where the true samples fall in a crucial interval. In our measurement, > 96% cases fall into the three patterns.



(a) Slow start effect of sequential file transfer in SpeedOf and TBB



(b) WiFiMaster when an intermediate hop suffers a temporary congestion



(c) SpeedTest when another connection is established to saturate the access link

Figure 4: Pathological scenarios where the true samples are not persistently covered by the crucial interval (less than 4% in our measurements).

randomly scattered noise samples help better "contrast" the crucial interval, leading to its improved approximation.

Crucial Interval Algorithm. Based on the above insights, our designed bandwidth estimation approach for FastBTS aims at finding this crucial interval ($[V_x, V_y]$) that has both *a high sample density* and *a large sample size*. Assuming there are N throughput samples ranging from V_{min} to V_{max}, our aim is formulated as maximizing the *product of density and size*. We denote the size as $K(V_x, V_y)$, *i.e.,* the number of samples that fall into $[V_x, V_y]$. The density can be calculated as the ratio between $K(V_x, V_y)$ and $N' = N(V_y - V_x)/(V_{max} - V_{min})$, where N' is the "baseline" corresponding to the number of samples falling into $[V_x, V_y]$ *if* all N samples are uniformly distributed in $[V_{min}, V_{max}]$. To prevent a pathological case where the density is too high, we enforce a lower bound of the interval: $V_y - V_x$ should be at least L_{min}, which is empirically set to $(V_{max} - V_{min})/(N-1)$. Given the above, the objective function to be maximized is:

$$F(V_x, V_y) = Density \times Size = C \cdot \frac{K^2(V_x, V_y)}{V_y - V_x}, \qquad (1)$$

where $C = (V_{max} - V_{min})/N$ is a constant. Once the optimal $[V_x, V_y]$ is calculated, we can derive the bandwidth estimation by averaging all the samples falling into this interval.

FastBTS computes the crucial interval as bandwidth probing (§3.2) is in progress, which serves as the acceptance-rejection function (ARF) of rejection sampling. When a new sample is available, the server computes a crucial interval by maximizing Equation (1). It thus produces a series of intervals $[V_{x3}, V_{y3}]$, $[V_{x4}, V_{y4}]$, $\cdots$ where $[V_{xi}, V_{yi}]$ corresponds to the interval generated when the i-th sample is available.

Searching Crucial Interval with Convex Hull. We now consider how to actually solve the maximization problem in Equation (1). To enhance the readability, we use L to denote $V_y - V_x$, use K to denote $K(V_x, V_y)$, and let the maximum value of $F(V_x, V_y)$ be F_{max}, which lies in $(0, \frac{C \cdot N^2}{L_{min}}]$.

Clearly, a naïve exhaustive search takes $O(N^2)$ time. Our key result is that this can be done much more efficiently in $O(N \log N)$ by strategically searching on a convex hull dynamically constructed from the samples. Our high-level approach is to perform a binary search for F_{max}. The initial midpoint is set to $\lfloor \frac{C \cdot N^2}{2 \cdot L_{min}} \rfloor$. In each binary search iteration, we examine whether the inequality $\frac{C \cdot K^2}{L} - m \geq 0$ holds for any interval(s), where $0 < m \leq F_{max}$ is the current midpoint. Based on the result, we adjust the midpoint and continue with the next iteration.

We next see how each iteration is performed exactly. Without loss of generality, we assume that the throughput samples are sorted in ascending order. Suppose we choose the i-th and j-th samples ($i < j$) from the N sorted samples as the end-

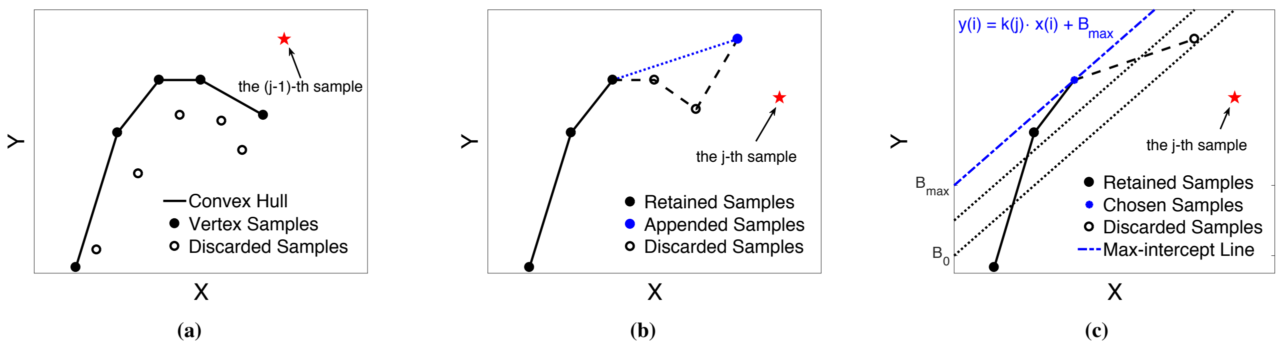


Figure 5: (a) The current convex hull under transformed coordinates, where the axes X and Y correspond to $x(i)$ and $y(i)$ in Equation (4) respectively. (b) Updating the convex hull with the $(j-1)$-th sample. (c) Searching for the max-intercept line.

points of the interval $[V_i, V_j]$. Then the inequality $\frac{C \cdot K^2}{L} - m \geq 0$ can be transformed as:

$$\frac{C \cdot (j-i+1)^2}{V_j - V_i} - m \geq 0. \tag{2}$$

We further rearrange it as:

$$i^2 - 2i + \frac{m}{C}V_i - 2ij \geq \frac{m}{C}V_j - 2j - j^2 - 1. \tag{3}$$

It is not difficult to discover that the right side of the inequality is only associated with the variable j, while the left side just relates to the variable i except the term $-2ij$. Therefore, we adopt the following coordinate conversion:

$$\begin{cases} k(j) = 2j \\ b(j) = \frac{m}{C}V_j - 2j - j^2 - 1 \\ x(i) = i \\ y(i) = i^2 - 2i + \frac{m}{C}V_i \end{cases} \tag{4}$$

With the above-mentioned coordinate conversion, the inequality (3) can be transformed as: $y(i) - k(j) \cdot x(i) \geq b(j)$. Then, determining whether the inequality holds for at least one pair of (i, j) is equivalent to finding the maximum of $f(i) = y(i) - k(j) \cdot x(i)$ for each $1 < j \leq N$.

As depicted in Figure 5a, we regard $\{x(i)\}$ and $\{y(i)\}$ as coordinates of N points on a two-dimensional plane (these points do not depend on j). It can be shown using the linear programming theory that for any given j, the largest value of $f(i)$ always occurs at a point that is on the convex hull formed by $(x(i), y(i))$. This dictates an algorithm where for each $1 < j \leq N$, we check the points on the convex hull to find the maximum of $f(i)$.

Since i must be less than j, each time we increment j (the outer loop), we progressively add one point $(x(j-1), y(j-1))$ to the (partial) convex hull, which is shown in Figure 5b. Then among all existing points on the convex hull, we search backward from the point with the largest $x(i)$ value to the smallest $x(i)$ to find the maximum of $f(i)$, and stops searching when $f(i)$ starts to decrease since the points are on the convex hull (the inner loop).

As demonstrated in Figure 5c, an analytic geometry explanation of this procedure is to determine a line with a fixed slope $y = k(j)x + B$, *s.t.* the line intersects with a point on the convex and the intercept B is maximized, and the maximized intercept corresponds to the maximum of $f(i)$.

Also, once the maximum of $f(i)$ is found at $(x(i'), y(i'))$ for a given j, all points that are to the right of $(x(i'), y(i'))$ can be removed from the convex hull – they must not correspond to the maximum of $f(i)$ for all $j' > j$. This is because (1) the slope of the convex hull's edge decreases as i increases, and (2) $k(j)$ increases as j increases. Therefore, using amortized analysis, we can show that in each binary search iteration, the overall processing time for all points is $O(N)$ as j grows from 1 to N. This leads to an overall complexity of $O(N \log N)$ for the whole algorithm.

Fast Result Generation. FastBTS selects a group of samples that well fit $T(x)$ as soon as possible while ensuring data reliability. Given two intervals $[V_{xi}, V_{yi}]$ and $[V_{xj}, V_{yj}]$, we regard their similarity as the Jaccard Coefficient [60]. FastBTS then keeps track of the similarity values of consecutive interval pairs *i.e.,* $S_{3,4}, S_{4,5}, \ldots$ If the test result stabilizes, the consecutive interval pairs' similarity value will keep growing from a certain value β, satisfying $\beta \leq S_{i,i+1} \leq \cdots \leq S_{i+k,i+k+1} \leq 1$. If the above sequence is observed, FastBTS determines that the result has stabilized and reports the bottleneck bandwidth as the average value of the throughput samples belonging to the most recent interval. The parameters β and k pose a tradeoff between accuracy and cost in terms of test duration and data traffic. Specifically, increasing β and k can yield a higher test accuracy while incurring a longer test duration and more data usage. Currently, we empirically set β=0.9 and k=2, which are found to well balance the tradeoff between the test duration and accuracy. Nevertheless, when dealing with those relatively rare cases that are not covered by this paper, BTS providers are recommended to do pre-tests in order to find the suitable parameter settings before putting CIS mechanism into actual use.

Solutions to Caveats. There do exist some "irregular" bandwidth graphs in prior work [37, 48, 56] where CIS may lose efficacy. For instance, due to in-network mechanisms like data aggregation and multi-path scheduling, the millisecond-level throughput samples can vary dramatically (*i.e.,* sometimes the throughput is about to reach the link capacity, and sometimes the throughput approaches zero). To mitigate this issue, we learn from some BTSes (*e.g.,* SpeedTest) and use a relatively large time interval (50 ms) to smooth the gathered throughput samples. However, even with smoothed samples, it is still possible that CIS may be inaccurate if the actual access bandwidth is outside the crucial interval, or the interval becomes too wide to give a meaningful bandwidth estimation. In our experiences, such cases are rare (less than 4% in our measurements in §2.3). However, to ensure that our tests are stable in all scenarios, we design solutions to those pathological cases.

Figure 4 shows all types of the pathological cases of CIS we observe, where $P(x)$ deviates from $T(x)$ over time. FastBTS leverages three mechanisms to resolve these cases: (1) elastic bandwidth probing (§3.2) reaches bottleneck bandwidth in a short time, effectively alleviating the impact of slow-start effect in Figure 4a; (2) data-driven server selection (§3.3) picks the expected highest-throughput server(s) for bandwidth tests, minimizing the requirement of additional connection(s) in Figure 4c; (3) adaptive multi-homing (§3.4) establishes concurrent connections with different servers, avoiding the underestimations in Figures 4b and 4c. We will discuss these mechanisms in §3.2 – §3.4.

3.2 Elastic Bandwidth Probing (EBP)

In rejection sampling, $P(x)$ determines the boundary of $T(x)$. A bandwidth test measures $P(x)$ using bandwidth probing based on network conditions. It shares a similar principle as congestion control at the test server's transport layer—the goal is to accommodate diverse noises over the live Internet, while saturating the bandwidth of the access link. FastBTS employs BBR [42], an advanced congestion control algorithm, as a starting point for probing design. Specifically, FastBTS uses BBR's built-in bandwidth probing for bootstrapping.

On the other hand, bandwidth tests have different requirements compared with congestion control. For example, congestion control emphasizes stable data transfers over a long period, while a BTS focuses on obtaining accurate link capacity as early as possible with the lowest data usage. Therefore, we modify and optimize BBR to support bandwidth tests.

BBR Prime. BBR is featured by two key metrics: bottleneck bandwidth $BtlBw$ and round-trip propagation time RT_{prop}. It works in four phases: Startup, Drain, ProbeBW (probing the bandwidth), and ProbeRTT. A key parameter *pacing_gain* (PG) controls TCP pacing so that the capacity of a network path can be fully utilized while the queuing delay is minimized. BBR multiplies its measured throughput by PG to determine the data sending rate in the subsequent RTT. After a connection is established, BBR enters the Startup phase and exponentially increases the sending rate (*i.e.,* $PG = \frac{2}{ln2}$) until the measured throughput does not increase further, as shown in Figure 6. At this point, the measured throughput is denoted as B_0 and a queue is already formed at the bottleneck of the network path. Then, BBR tries to Drain it by reducing PG to $\frac{ln2}{2} < 1$ until there is expected to be no excess in-flight data. Afterwards, BBR enters a (by default) 10-second ProbeBW phase to gradually probe $BtlBw$ in a number of cycles, each consisting of 8 RTTs with $PGs = \{\frac{5}{4}, \frac{3}{4}, 1, 1, 1, 1, 1, 1\}$. We plot in Figure 6 four such cycles tagged as ①②③④. Finally (10 seconds later), the *maximum* value of the measured throughput samples is taken as the network path's $BtlBw$ and BBR enters a 200-ms ProbeRTT phase to estimate RT_{prop}.

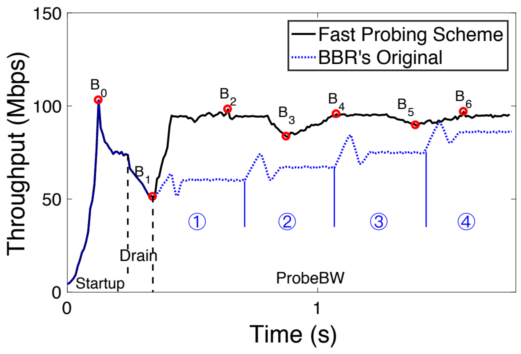


Figure 6: Elastic bandwidth probing vs. BBR's original scheme.

Limitations of BBR. Directly applying BBR's $BtlBw$-based probing method to BTSes is inefficient. First, as illustrated in Figure 6 (where the true $BtlBw$ is 100 Mbps), BBR's $BtlBw$ probing is conservative, making the probing process unnecessarily slow. A straightforward idea is to remove the 6 RTTs with $PG = 1$ in each cycle. Even with that, the probing process is still inefficient when the data (sending) rate is low. Second, when the current data rate (*e.g.,* 95 Mbps) is close to the true $BtlBw$ (*e.g.,* 100 Mbps), using the fixed PG of $\frac{5}{4}$ causes the data rate to far overshoot its limit (*e.g.,* to 118.75 Mbps). This may not be a severe issue for data transfers, but may significantly slow down the convergence of $BtlBw$ and thus lengthen the test duration. Third, BBR takes the maximum of all throughput samples in each cycle as the estimated $BtlBw$. The simple maximization operation is vulnerable to outliers and noises (this is addressed by CIS in §3.1).

Elastic Data-rate Pacing. We design *elastic pacing* to make bandwidth probing faster, more accurate, and more adaptive. Intuitively, when the data rate is low, it ramps up quickly to reduce the probing time; when the data rate approaches the estimated bottleneck bandwidth, it performs fine-grained probing by reducing the step size, towards a smooth convergence. This is in contrast to BBR's static probing policy.

We now detail our method. As depicted in Figure 6, once entering the ProbeBW phase, we have recorded B_0 and B where B_0 is the peak data rate measured during the Startup phase and B is the current data rate. Let the ground-truth bottleneck bandwidth be B_T. Typically, B_0 is slightly higher than B_T due to queuing at the bottleneck link in the end of the Startup phase (otherwise it will not exit Startup); also, B is lower than B_T due to the Drain phase. We adjust the value of B by controlling the *pacing gain* (PG) of the data sending rate, but the pivotal question here is the adjustment policy, *i.e.,* how to make B approach B_T quickly and precisely.

Our idea is inspired by the spring system [51] in physics where the restoring force of a helical spring is proportional to its elongation. We thus regard PG as the restoring force, and B's deviation from B_0 as the elongation (we do not know B_T so we approximate it using B_0). Therefore, the initial PG is expected to grow to:

$$PG_{grow-0} = (PG_m - PG_0) \times \left(1 - \frac{B}{B_0}\right) + PG_0, \quad (5)$$

where $1 - \frac{B}{B_0}$ denotes the normalized distance between B_0 and B, and PG_0 represents the default value (1.0) of PG. We set the upper bound of PG (PG_m) as $\frac{2}{ln2}$ that matches BBR's PG in the (most aggressive) Startup phase. As Equation (5) indicates, the spring system indeed realizes our idea: it increases the data rate rapidly when B is well below B_0, and cautiously reduces the probing step as B grows.

To accommodate a corner scenario where B_0 is lower than B_T ($< 1\%$ cases in our experiments), we slightly modify Equation (5) to allow B to overshoot B_0 marginally:

$$PG_{grow-0} = max\{(PG_m - PG_0') \times \left(1 - \frac{B}{B_0}\right) + PG_0', PG_0'\}, \quad (6)$$

where PG_0' equals $1 + \varepsilon$, with ε being empirically set to 0.05.

When the data rate overshoots the bottleneck bandwidth (*i.e.,* the number of in-flight bytes exceeds the bandwidth-delay product), we reduce the data rate to suppress the excessive in-flight bytes. This is realized by inverting PG to $PG_{drop-0} = \frac{1}{PG_{grow-0}}$. This process continues until no excessive in-flight data is present. At that moment, B will again drop below B_T, so we start a new cycle to repeat the aforementioned data rate growth process.

To put things together, in our design, the ProbeBW phase consists of a series of cycles each consisting of only two stages: *growth* and *drop*. Each stage has a variable number of RTTs, and the six RTTs with $PG = 1$ in the original BBR algorithm are removed. The transitions between the two stages are triggered by the formation and disappearance of excessive in-flight bytes (*i.e.,* the queueing delay). In the i-th cycle, the PGs for the two stages are:

$$\begin{cases} PG_{grow-i} = max\{(PG_m - PG_0') \times \left(1 - \frac{B}{B_i}\right) + PG_0', PG_0'\}, \\ PG_{drop-i} = \frac{1}{PG_{grow-i}}, \end{cases} \quad (7)$$

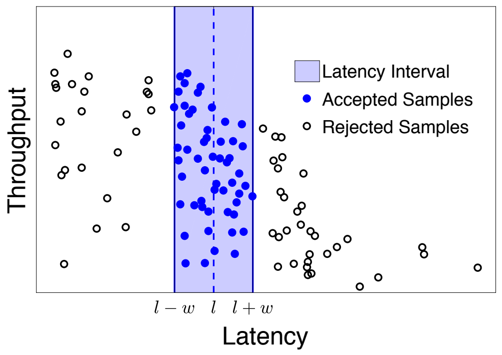


Figure 7: Data-driven server selection that takes both historical latency and throughput information into account.

where B is the data rate at the beginning of a growth stage, and B_i is the peak rate in the previous cycle's growth stage. Interestingly, by setting $B_0 = +\infty$, we can make the growth and drop stages identical to BBR's Startup and Drain phases, respectively. Our final design thus only consists of a single phase (ProbeBW) with two stages. The ProbeRTT phase is removed because it does not help our bandwidth probing.

Compared with traditional bandwidth probing mechanisms, EBP can saturate available bandwidth more quickly as it ramps up the sending rate when the current rate is much lower than the estimated bandwidth. Meanwhile, when the sending rate is about to reach the estimated bandwidth, EBP carefully increases the rate in order to be less aggressive to other flows along the path than other bandwidth probing mechanisms.

3.3 Data-driven Server Selection (DSS)

FastBTS includes a new server selection method. We find that selecting the test server(s) with the lowest PING latency, widely used in existing BTSes, is ineffective. Our measurement shows that latency and the available bandwidth are not highly correlated—the servers yielding the highest throughput may not always be those with the lowest PING latency.

FastBTS takes a *data-driven approach* for server selection (DSS): each test server maintains a database (model) containing {latency, throughput} pairs obtained from the setup and bandwidth probing phases of past tests. Then in a new setup phase, the client still PINGs the test servers, while each server returns an expected throughput value based on the PING latency by looking up the database. The client will then rank the selected server(s) based on their expected throughput values.

As demonstrated in Figure 7, the actual DSS algorithm is conceptually similar to CIS introduced in §3.1, whereas we empirically observe that only considering the density can yield decent results. Specifically, given a latency measurement l, the server searches for a width w that maximizes the density defined as $K(l,w)/2w$, where $K(l,w)$ denotes the number of latency samples falling in the latency interval $[l-w, l+w]$. The expected throughput is calculated as an average of all samples in $[l-w, l+w]$. In addition, the server also returns the maximum throughput (using the 99-percentile value) belonging to $[l-w, l+w]$ to the client. Both values will be used in the bandwidth probing phase (§3.4). During bootstrapping when servers have not yet accumulated enough samples, the client can fallback to the traditional latency-based selection strategy. To keep their databases up-to-date, servers can maintain only most recent samples.

3.4 Adaptive Multi-Homing (AMH)

For high-speed access networks like 5G, the last-mile access link may not always be the bottleneck. To saturate the access link, we design an adaptive multi-homing (AMH) mechanism to dynamically adjust the concurrency level, *i.e.,* the number of concurrent connections between the servers and client.

AMH starts with a single connection to cope with possibly low-speed access links. For this single connection C_1, when CIS (§3.1) has accomplished using the server S_1 (the highest-ranking server, see §3.3), the reported bottleneck bandwidth is denoted as BW_1. At this time, the client establishes another connection C_2 with the second highest-ranking server S_2 while retaining C_1. C_2 also works as described in §3.2. Note we require S_1 and S_2 to be in different ASes to minimize the likelihood that S_1 and S_2 share the same Internet-side bottleneck. Moreover, we pick the server with the second highest bandwidth estimation as S_2 to saturate the client's access link bandwidth with the fewest test servers. After that, we view C_1 and C_2 together as an "aggregated" connection, with its throughput being $BW_2 = BW_{2,1} + BW_{2,2}$, where $BW_{2,1}$ and $BW_{2,2}$ are the real-time throughput of C_1 and C_2 respectively.

By monitoring $BW_{2,1}$, $BW_{2,2}$, and BW_2, FastBTS applies intelligent throughput sampling and fast result generation (§3.1) to judge whether BW_2 has become stable. Once BW_2 stabilizes, AMH determines whether the whole bandwidth test process should be terminated based on the relationship between BW_1 and $BW_{2,1}$. If for C_1 the bottleneck link is not the access link, $BW_{2,1}$ should have a value similar to or higher than BW_1 (assuming the unlikeliness of C_1 and C_2 sharing the same Internet-side bottleneck [53]). In this case, the client establishes another connection with the third highest-ranking server S_3 (with a different AS), and repeats the above process (comparing $BW_{3,1}$ and BW_1 to decide whether to launch the fourth connection, and so on). Otherwise, if $BW_{2,1}$ exhibits a noticeable decline (empirically set to $> 5\%$) compared to BW_1, we regard that C_1 and C_2 saturate the access link and incur cross-flow contention. In this case, the client stops probing and reports the access bandwidth as $max(BW_1, BW_2)$.

4 Implementation

As shown in Figure 1, we implement *elastic bandwidth probing* (EBP) and *crucial interval sampling* (CIS) on the server side, because EBP works at the transport layer and thus requires OS kernel modifications, and CIS needs to get fine-grained throughput samples from EBP in real time. We implement EBP and CIS in C and Node.js, respectively.

We implement *data-driven server selection* (DSS) and *adaptive multi-homing* (AMH) on the client side. End users can access the FastBTS service through REST APIs. We implement DSS and AMH in JavaScript to make them easy to integrate with web pages or mobile apps.

The test server is built on CentOS 7.6 with the Linux kernel version of 5.0.1. As mentioned in §3.2, we develop EBP by using BBR as the starting point. Specifically, we implement the calculation of *pacing_gain* according to Equation (7) by modifying the `bbr_update_bw` function; we also modify `bbr_set_state` and `bbr_check_drain` to alter BBR's original cycles in the ProbeBW phase, so as to realize EBP's two-stage cycles. EBP is implemented as a loadable kernel module. CIS is a user-space program. To efficiently send in-situ performance statistics including throughput samples, *Btlbw*, and RT_{prop} from EBP to CIS (both EBP and CIS are implemented on the server side in C and Node.js), we use the Linux Netlink Interface in `netlink.h` to add a raw socket and a new packet structure `bbr_info` that carries the above performance information. The performance statistics are also sent to the client by piggybacking with probing traffic, allowing users to examine the real-time bandwidth test progress.

5 Evaluation

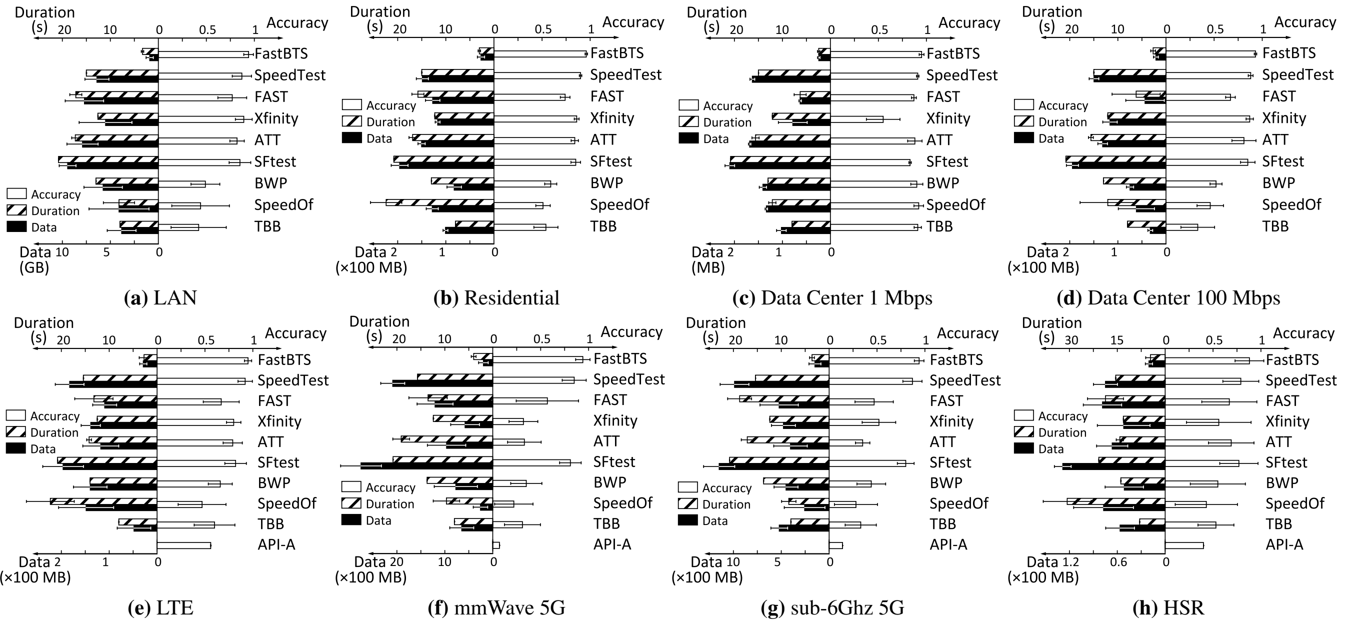
5.1 Experiment Setup

We compare FastBTS with the 9 state-of-the-art BTSes studied in §2. For fair comparisons, we re-implement all the BTSes based on our reverse engineering efforts and use the same setup for all these re-implemented BTSes. To do so, we built the following testbeds.

Large-scale Testbed. We deploy a total of 30 test servers on 30 VMs across the globe (North America, South America, Asia, Europe, Australia, and Africa) with the same configurations (dual-core Intel CPU@2.5 GHz, 8-GB DDR memory, and 1.5+ Gbps outgoing bandwidth). The size of the server pool (30) is on par with 5 out of the 9 BTSes but is smaller than those of FAST and SpeedTest (Table 2), which we assume is a representative server pool size adopted by today's commercial BTSes. We deploy 100+ clients including 3 PCs, 4 smartphones, and 108 VMs (the same as those adopted in §2). For a fair comparison with FastBTS, we replicate the 9 other popular BTSes: SpeedOf, BWP, SFtest, ATTtest, Xfinity, FAST, SpeedTest, TBB, and Android API-A (see §2.3) and deploy them on the 30 test servers and the 100+ clients. We deploy API-A (Android specific) on 4 phones.

Tested Networks. We conduct extensive evaluations under heterogeneous networks. We detail their setups below.

- *Residential Broadband.* We deploy three PCs located in China, U.S., and Germany (Table 1). All the PCs' access links are 100 Mbps residential broadband. The three clients



(a) LAN **(b)** Residential **(c)** Data Center 1 Mbps **(d)** Data Center 100 Mbps

(e) LTE **(f)** mmWave 5G **(g)** sub-6Ghz 5G **(h)** HSR

Figure 8: Duration and test accuracy of FastBTS, compared with 9 BTSes under various networks. "API-A" refers to Android API-A (§2.4).

communicate with (a subset of) the aforementioned 30 test servers to perform bandwidth tests. We perform in one day 90 groups of tests, consisting of 3 clients $\times$ 3 different time-of-day (0:00, 8:00, and 16:00) $\times$ 10 repetitions.

- *Data Center Networks.* We deploy 108 VMs belonging to different commercial cloud providers as the clients (§2.3). We perform a total number of 108 VMs $\times$ 3 time-of-day $\times$ 10 repetitions = 3,240 groups of tests.
- *mmWave 5G* experiments were conducted at a downtown street in a large U.S. city, with a distance from the phone (Samsung GS10, see Table 1) to the base station of 30m. This is a typical 5G usage scenario due to the small coverage of 5G base stations. The phone typically has line-of-sight to the base station unless being blocked by passing vehicles. The typical downlink throughput is between 0.9 and 1.2 Gbps. We perform in one day 120 groups of tests, consisting of 4 clients $\times$ 3 time-of-day $\times$ 10 repetitions.
- *Sub-6Ghz 5G* experiments were conducted in a Chinese city using an HV30 phone over China Mobile. The setup is similar to that of mmWave. We run 120 groups of tests.
- *LTE* experiments were conducted in both China (a university campus) and U.S. (a large city's downtown area) using XM8 and GS9, respectively, each with 120 groups of tests.
- *HSR Cellular Access.* We also perform tests on high-speed rail (HSR) trains. We take the Beijing-Shanghai HSR line (peak speed of 350 km/h) with two HV30 phones. We measure the LTE bandwidth from the train. We run 2 clients $\times$ 50 repetitions = 100 groups of tests.
- *LAN.* Besides the deployment of 30 VMs, we also create an in-lab LAN testbed to perform controlled experiments, where we can craft background traffic. The testbed consists of two test servers (S_1, S_2) and two clients (C_1, C_2), each equipping a 10 Gbps NIC. They are connected by a commodity switch with a 5 Gbps forwarding capability, thus being the bottleneck. When running bandwidth tests on this testbed, we maintain two parallel flows: one 1 Gbps background flow between S_1 and C_1, and a bandwidth test flow between S_2 and C_2.

We use the three metrics described in §2.1 to assess BTSes: test duration, data usage, and accuracy. Also, the methodology for obtaining the ground truth is described in §2.1.

5.2 End-to-End Performance

LAN and Residential Networks. As shown in Figure 8a and 8b, FastBTS yields the highest accuracy (0.94 for LAN and 0.96 for residential network) among the 9 BTSes, whose accuracy lies within 0.44–0.89 for LAN, and 0.51–0.9 for residential network. The average test duration of FastBTS for LAN and residential network is 3.4 and 3.0 seconds respectively, which are 2.4–7.4$\times$ shorter than the other BTSes. The average data usage of FastBTS is 0.9 GB for LAN and 27 MB for residential network, which are 3.1–10.5$\times$ less than the other BTSes. The short test duration and small data usage are attributed to EBP (§3.2), which allows a rapid data rate increase when the current data rate is far lower than the bottleneck bandwidth, as well as fast result generation, which strategically trades off accuracy for a shorter test duration.

Data Center Networks. Figure 8c and 8d show the performance of different BTSes in CSPs' data center networks with the bandwidth of $\{1, 100\}$ Mbps (per the CSPs' service agreements). FastBTS outperforms the other BTSes by yielding the highest accuracy (0.94 on average), the shortest test duration (2.67 seconds on average), and the smallest data usage (21 MB on average for 100-Mbps network). In contrast, the other BTSes' accuracy ranges between 0.46 and 0.91; their test duration is also much longer, from 6.2 to 20.8 seconds, and they consume much more data (from 44 to 194 MB) compared to FastBTS. In particular, we find that on low-speed network (1 Mbps), some BTSes such as Xfinity and SFtest establish too many parallel connections. This leads to poor performance due to the excessive contention across the connections. FastBTS addresses this issue through AMH (§3.4) that adaptively adjusts the concurrency level according to the network condition. The results of networks with 10 Mbps bandwidth are similar to those of 100 Mbps networks.

LTE and 5G Networks. We evaluate the BTSes' performance on commercial LTE and 5G networks (both mmWave and sub-6Ghz for 5G). Over LTE, as plotted in Figure 8e, FastBTS owns the highest accuracy (0.95 on average), the smallest data usage (28.23 MB on average), and the shortest test duration (2.73 seconds on average). The other 9 BTSes are far less efficient: 0.62–0.92 for average accuracy, 41.8 to 179.3 MB for data usage, and 7.1 to 20.8 seconds for test duration. For instance, we discover that FAST bears a quite low accuracy (0.67) because its window-based mechanism is very vulnerable to throughput fluctuations in LTE. SpeedTest, despite having a decent accuracy (0.92), incurs quite high data usage (166.3 MB) since it fixes the bandwidth test duration to 15 seconds regardless of the stability of the network.

Figure 8f shows the results for mmWave 5G. It is also encouraging to see that FastBTS outperforms the 9 other BTSes across all three metrics (0.94 vs. 0.07–0.87 for average accuracy, 194.7 vs. 101–2,749 MB for data usage, and 4.0 vs. 8.9–26.2 seconds for test duration). Most of the BTSes have low accuracy (< 0.6); Speedtest and SFtest bear relatively high accuracy (0.81 and 0.85). However, the high data usage issue due to their flooding nature is drastically amplified in mmWave 5G. For example, Speedtest incurs very high data usage—up to 2,087 MB per test. The data usage for FAST is even as high as 2.75 GB. FastBTS addresses this issue through the synergy of its key features for fuzzy rejection sampling such as EBP and CIS. We observe similar results in the sub-6Ghz 5G experiments as shown in Figure 8g.

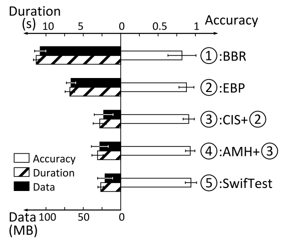
HSR Cellular Access. We also benchmark the BTSes on an HSR train running at a peak speed of 350km/h from Beijing to Shanghai. As shown in Figure 8h, the accuracy of all 10 BTSes decreases. This is attributed to two reasons. First, on HSR trains, the LTE bandwidth is highly fluctuating because of, *e.g.,* frequent handovers caused by high mobility and the contention traffic from other passengers. Second, given such fluctuations, performing bulk transfer before and after a bandwidth test can hardly capture the ground truth bandwidth, which varies significantly during the test. Nevertheless, compared to the other 9 BTSes, FastBTS still achieves the best performance (0.88 vs. 0.26–0.84 for average accuracy, 20.3 vs. 16–155 MB for data usage, and 4.6 vs. 10.6–32.4 seconds for test duration). The test duration is longer than the stationary scenarios because under high mobility, network condition fluctuation makes crucial intervals converge slower.

5.3 Individual Components

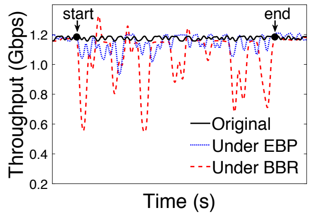
We evaluate the benefits of each component of FastBTS by incrementally enabling one at a time. When EBP is not enabled, we use BBR. When CIS is not enabled, we average the throughput samples to calculate the test result. When DSS is not enabled, test server(s) are selected based on PING latency. When AMH is not enabled, we apply SpeedTest's (single-homing) connection management logic.

Bandwidth Probing Schemes. We compare BBR-based FastBTS and SpeedTest under mmWave 5G. The average data usage of BBR per test (735 MB) is 65% less than that of our replicated SpeedTest (2,087 MB). Meanwhile, the accuracy of BBR slightly reduces from 0.85 to 0.81. The results indicate that BBR's *BtlBw* estimation mechanism better balances the tradeoff between accuracy and data usage compared to flooding-based methods. We next compare BBR and EBP. We find that our EBP brings further improvements over BBR: under mmWave, EBP achieves an average data usage of 419 MB (compared 735 MB in BBR, a 42% reduction) and an average accuracy of 0.87 (compared to 0.81 in BBR, a 7% improvement). The advantages of EBP come from its elastic PG setting mechanism that is critical for adaptive bandwidth probing. Next, we compare BBR and EBP under data center networks. As shown in Figure 9a, compared to BBR, EBP reduces the average test duration by 40% (from 11.3 to 6.6 seconds) and the data usage by 38% (from 107 to 66 MB), while improving the average accuracy by 6%.

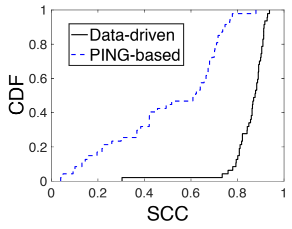
Probing Intrusiveness. We evaluate the probing intrusiveness of vanilla BBR and EBP using the LAN testbed (§5.1). Recall that we simultaneously run a 1 Gbps background flow and the bandwidth test flow that shares a 5 Gbps bottleneck at the switch. Ideally, a BTS should measure the bottleneck bandwidth to be 4 Gbps without interfering with the background flow, whose average/stdev throughput is thus used as a metric to assess the intrusiveness of BBR and EBP. Our test procedure is as follows. We first run the background flow alone for 1 minute and measure its throughput as R_{Origin}= 1 Gbps. We then run BBR and EBP with the background flow and measure the average (standard deviation) of the background flow throughput as R_{BBR} (S_{BBR}) and R_{EBP} (S_{EBP}), respectively, during the test. We demonstrate the three groups' throughput samples with their timestamps normalized by BBR's test duration in Figure 9b. EBP incurs a much smaller impact on the background flow compared to BBR, with R_{EBP} and R_{BBR} measured to be 0.97 Gbps and 0.90 Gbps, respectively.



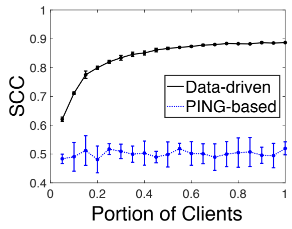
(a)



(b)



(c)



(d)

Figure 9: (a) Impact of individual modules of FastBTS in 100-Mbps data center networks. (b) Comparing intrusiveness between EBP and BBR. (c) Distributions of SCC_{gp} (PING-based) and SCC_{gd} (Data-driven) when P=20%; (d) P (portion of clients) vs. SCC_{gp} and SCC_{gd}.

Also, under EBP, the background flow's throughput variation is lower than under BBR: S_{EBP}/R_{EBP} and S_{BBR}/R_{BBR} are calculated to be 0.03 and 0.15, respectively. This suggests that when probing the bandwidth, EBP is less intrusive than BBR. We repeat the above test procedure under other settings, with the background flow's bandwidth varying from 0.5 to 4 Gbps, and observe consistent results. The lower intrusiveness of EBP compared with vanilla BBR probably lies in that when the sending rate is about to hit the access link bandwidth, EBP tends to carefully reclaim the available bandwidth; however, vanilla BBR still increases the sending rate with a fixed step, which is more aggressive than EBP in this scenario.

Crucial Interval Sampling (CIS). We further enable CIS. As shown in Figure 9a, by strategically removing outliers, CIS increases the average accuracy from 0.87 (EBP only) to 0.91. Due to the fast result generation mechanism, the test duration is reduced from 6.8 seconds to 2.8 seconds and the data usage is reduced by 2.8× (compared with EBP only).

We next compare CIS with the sampling approaches used by the other 9 BTSes (§5.1), which use a total of five bandwidth sampling algorithms because SFtest, ATTtest, and Xfinity employ the same trivial approach of simply averaging the throughput samples. To fairly compare them with CIS, we take a replay-based approach. Specifically, we select one "template" BTS from which we collect the network traces during the bandwidth probing phase; the time series of the aggregated throughput across all connections is then obtained from the traces and fed to all the sampling algorithms. We exclude SpeedOf and BWP from this experiment because they calculate the bandwidth based on the last or fastest connection that cannot be precisely reconstructed by our replay approach. We next show the results by using SpeedTest as the template BTS. The simplest algorithm (averaging) bears the lowest accuracy (0.81) because it is poor at eliminating the noises caused by, for example, TCP congestion control; the accuracy values of FAST, and SpeedTest are 0.82 and 0.84, respectively. In contrast, CIS owns the highest accuracy (0.91). This confirms the effectiveness of CIS's sampling approach. The efficiency and effectiveness of CIS lies in that, instead of incurring much redundancy in test duration and data usage to achieve a decent test accuracy, CIS keeps calculating the crucial interval of the gathered throughput samples. Once the crucial interval stabilizes, CIS immediately stops the test, thus significantly saving test duration and data usage.

Adaptive Multi-Homing (AMH). When AMH is further enabled, the test accuracy increases from 0.91 (EBP+CIS) to 0.93 (EBP+CIS+AMH), as shown in Figure 9a. Meanwhile, since testing over more connections takes additional time, AMH slightly lengthens the average test duration from 2.8 to 3.1 seconds, with the average data usage increased from 23 MB to 28 MB. We repeat the above experiments over mmWave 5G networks where the bottleneck is more likely to shift to the Internet side. The results show that AMH improves the average accuracy from 0.84 to 0.91, while incurring moderate overhead by increasing the average data usage from 148 MB to 206 MB and the average test duration from 3.3 to 4.1 seconds. The results suggest that AMH is essential for high-speed networks such as mmWave 5G.

Data-driven Server Selection (DSS). We employ *cross-validation* for a fair comparison between the PING-based method and DSS in three steps: (1) We do file transfers between every server and a randomly selected portion (P) of all clients to gather throughput samples. (2) Each client C runs a bandwidth test towards every server. In each test, the clients' historical test records (excluding the record of C) gathered in the previous step is utilized by each server to calculate the expected bandwidth, which is then returned to C. (3) Each client calculates three rankings of the servers based on the server-returned expected bandwidth: $Rank_g$, $Rank_p$, and $Rank_d$. $Rank_g$ refers to the server ranking based on the ground truth. $Rank_p$ is the ranking calculated based on PING latency; and $Rank_d$ is the ranking computed by DSS. We use the *Spearman Correlation Coefficient* (SCC [59]) to calculate the similarity SCC_{gp} between $Rank_g$ and $Rank_p$, as well as the similarity SCC_{gd} between $Rank_g$ and $Rank_d$.

The distributions of SCC_{gp} and SCC_{gd} when $P = 20\%$ are shown in Figure 9c. We find that SCC_{gd} is much higher than SCC_{gp} in terms of the median (0.81 *vs.* 0.63), average (0.80 *vs.* 0.50), and maximum (0.93 *vs.* 0.88) values. Further, Figure 9d shows that SCC_{gd} drops as P decreases; however, even when P decreases to 5%, SCC_{gd} (0.62) is still 24% larger than SCC_{gp} (0.5). These results show that even with limited historical data, DSS works reasonably well. We enable DSS in our experiments of Figure 9a with $P = 20\%$. Compared to EBP+CIS+AMH, enabling DSS improves the average accuracy from 0.93 to 0.94; the test duration slightly reduces.

Overall Runtime Overhead. The client-side overhead of FastBTS is negligible based on our measurement on Samsung Galaxy S9, S10, Xiaomi M8, and Huawei Honor V30. On the server side, the incurred overhead is also low. When *Btlbw* is 100 Mbps, the CPU overhead is measured to be lower than 5% (single core, tested on Intel CPU@2.5 GHz, 8-GB memory). The CPU overhead is only 12% when *Btlbw* is 5 Gbps.

6 Related Work

Bandwidth Measurement. Bandwidth measurement is an essential component for many networked systems that empower many important applications and use cases [46, 61, 62, 75]. Apart from the BTSes described in §2, other bandwidth measurement methods mostly target specific types of networks (*e.g.,* datacenter [44], LTE [56, 71], and wireless [74, 77]) and require special support from the deployed infrastructure. For example, AuTO [44] conducts bandwidth estimation over DCTCP in data centers; it needs switch support to tag ECN marks on the data packets, and thus is challenging to be applied in WAN. Huang et al. [56] propose to deploy monitors inside the cellular core network for bandwidth measurement. Dischinger et al. [47] devise a bandwidth measurement tool which concurrently leverages multiple packet trains with different sending rates to measure the link bandwidth of residential broadband network.

While almost all commercial BTSes employ flooding-based methods to combat measurement noises, there exists quite a few non-flooding methods [55, 65, 68, 70] in academia, which indirectly infer the available bandwidth based on timing information of crafted packets (including packet pairs and packet trains). Unfortunately, these methods are highly sensitive to timing information, and thus can be easily disrupted by many factors like packet loss [54, 64], queueing [54], and data/ACK aggregation [64], especially in high-speed networks.

Designed as a generic network service for Internet users, FastBTS differs from and complements the above work. FastBTS targets at conducting fast and light bandwidth tests especially for high-speed wide-area networks (*e.g.,* 5G), significantly reducing data usage and test duration for clients. It does not require any hardware support at the client side. On the server side, we show that FastBTS requires a much smaller deployment to achieve the same level of effectiveness of existing large-scale commercial BTSes (*e.g.,* SpeedTest).

Congestion Control. FastBTS's elastic bandwidth probing is inspired by congestion control algorithms [63, 66, 79]. We categorize congestion control algorithms based on the congestion indicators: (1) *Loss-based CCs* (*e.g.,* BIC-TCP [76] and CUBIC [52]) which take packet loss as the indicator. They are vulnerable to bufferbloat and random losses [34]. (2) *Delay-based CCs* (*e.g.,* TCP FAST [58] and TCP Vegas [40]) which take transmission delay as the indicator. They are known to under-utilize the available bandwidth as the Internet latency is inherently noisy and fluctuating. (3) *Rate-based CCs* (*e.g.,* BBR [42], PCC [48] and PCC Vivace [49]) which directly estimate the available bandwidth and accordingly adjust data sending rate, typically via a feedback loop. We choose to design elastic bandwidth probing based on BBR, because BBR is mature with large-scale deployment on WAN [18], edge [27, 57], and cellular networks [35].

7 Concluding Remarks

We present FastBTS, a novel bandwidth testing system, to make bandwidth testing fast and light as well as accurate. By accommodating and exploiting the test noises, FastBTS achieves the highest level of accuracy among commercial BTSes, while significantly reducing data usage and test duration. Further, FastBTS only employs 30 servers, 2–3 orders of magnitude fewer than the state of the arts.

Despite the above merits, FastBTS still bears several limitations at the moment. First, when testing a client's uplink bandwidth, FastBTS requires extra deployment efforts (in particular a kernel module of EBP, as demonstrated in Figure 1) at the client side. Second, the performance of the data-driven server selection (DSS) mechanism can be affected by its cold start phase as well as the specific deployment of test servers. Third, when the selected test servers cannot saturate the client's downlink bandwidth, the adaptive multi-homing (AMH) mechanism may need several rounds to make the bandwidth probing process converge, thus leading to a relatively long test duration. We have been exploring practical ways to overcome these limitations.

Acknowledgements

We sincerely thank the anonymous reviewers for their valuable comments, and our shepherd Prof. Andreas Haeberlen for guiding us through the revision process. Also, we appreciate the generous help from Hongzhe Yang and Jiaxing Qiu in system deployment and text proofreading. This work is supported in part by the National Key R&D Program of China under grant 2018YFB1004700, the National Natural Science Foundation of China (NSFC) under grants 61822205, 61902211, 61632020 and 61632013, and the Beijing National Research Center for Information Science and Technology (BNRist).

References

[1] Add 5G capabilities to your app. https://developer.android.com/about/versions/11/features/5g.

[2] AT&T BTS. http://speedtest.att.com/speedtest/.

[3] BandwidthTest API in Android. https://cs.android.com/android/platform/superproject/+/master:frameworks/base/core/tests/bandwidthtests/src/com/android/bandwidthtest/BandwidthTest.java.

[4] BTS Insights. https://www.speedtest.net/insights.

[5] BWP BTS. https://www.bandwidthplace.com/.

[6] Centurylink BTS. https://www.centurylink.com/home/help/internet/internet-speed-test.html/.

[7] Cox BTS. https://www.cox.com/residential/support/internet/speedtest.html.

[8] DSLReports. http://www.dslreports.com/speedtest/.

[9] Eighth Measuring Broadband America Fixed Broadband Report: A Report on Consumer Fixed Broadband Performance in the United States by (2018). Technical report, Federal Communications Commission.

[10] FAST BTS. https://fast.com/.

[11] Home Network Tips for the Coronavirus Pandemic. https://www.fcc.gov/home-network-tips-coronavirus-pandemic.

[12] How Coronavirus Affects Internet Usage and What You Can Do to Make Your Wi-Fi Faster. https://www.nbcnewyork.com/news/local/how-coronavirus-affects-internet-usage-and-what-you-can-do-to-make-your-wi-fi-faster/2332117/.

[13] HTML5 Speed Test by SourceForge. https://sourceforge.net/speedtest/.

[14] Measuring Broadband America Fixed Broadband Report (2016). Technical report, Federal Communications Commission.

[15] Measuring Broadband America Fixed Broadband Report: A Report on Consumer Fixed Broadband Performance in the US by (2014). Technical report, Federal Communications Commission.

[16] Nperf BTS. https://www.nperf.com/en/map/US/-/2420.ATT-Mobility/signal/?ll=37.59682400108367&lg=-109.44030761718751&zoom=8.

[17] NYSbroadband BTS. http://nysbroadband.speedtestcustom.com/.

[18] Optimizing HTTP/2 Prioritization with BBR and Tcp_notsent_lowat. https://blog.cloudflare.com/http-2-prioritization-with-nginx/.

[19] Optimum BTS. https://www.optimum.net/pages/speedtest.html.

[20] Source of Android / NetworkCapabilities.java. https://cs.android.com/android/platform/superproject/+/master:frameworks/base/packages/Connectivity/framework/src/android/net/NetworkCapabilities.java.

[21] SpaceX Starlink Speeds Revealed as Beta Users Get Downloads of 11 to 60Mbps. https://arstechnica.com/information-technology/2020/08/spacex-starlink-beta-tests-show-speeds-up-to-60mbps-latency-as-low-as-31ms/.

[22] Speakeasy. https://www.speakeasy.net/speedtest/.

[23] Spectrum BTS. https://www.spectrum.com/internet/speedtest-only/.

[24] Speedof.me BTS. https://www.speedof.me/.

[25] SpeedTest BTS. https://www.speedtest.net.

[26] Speedtest.xyz BTS. https://speedtest.xyz/.

[27] TCP BBR Congestion Control Comes to GCP – Your Internet Just Got Faster. https://cloud.google.com/blog/products/gcp/tcp-bbr-congestion-control-comes-to-gcp-your-internet-just-got-faster.

[28] ThinkBroadband BTS. https://www.thinkbroadband.com/speedtest/.

[29] Understanding Internet Speeds. https://www.att.com/support/article/u-verse-high-speed-internet/KM1010095.

[30] Verizon BTS. https://www.verizon.com/speedtest/.

[31] WiFiMaster. https://en.wifi.com/wifimaster/.

[32] Xfinity BTS. http://speedtest.xfinity.com/.

[33] E. Alimpertis, A. Markopoulou, and U. Irvine. A System for Crowdsourcing Passive Mobile Network Measurements. In *Proc. of NSDI (2017)*. USENIX.

[34] V. Arun and H. Balakrishnan. Copa: Practical Delay-based Congestion Control for the Internet. In *Proc. of NSDI (2018)*, pages 329–342. USENIX.

[35] E. Atxutegi, F. Liberal, H. K. Haile, et al. On the Use of TCP BBR in Cellular Networks. *IEEE Communications Magazine (2018)*, 56(3):172–179.

[36] V. Bajpai and J. Schönwälder. A Survey on Internet Performance Measurement Platforms and Related Standardization Efforts. *IEEE Communications Surveys & Tutorials (2015)*, 17(3):1313–1341.

[37] S. Bauer, D. Clark, and W. Lehr. Understanding Broadband Speed Measurements. *MIT Computer Science & Artificial Intelligence Lab, Tech. Rep. (2010)*.

[38] S. Bauer et al. Improving the Measurement and Analysis of Gigabit Broadband Networks. *SSRN 2757050 (2016)*.

[39] Z. S. Bischof, J. S. Otto, M. A. Sánchez, et al. Crowdsourcing isp characterization to the network edge. In *Proc. of SIGCOMM W-MUST workshop (2011)*, pages 61–66. ACM.

[40] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *Proc. of SIGCOMM (1994)*, pages 24–35. ACM.

[41] I. Canadi, P. Barford, and J. Sommers. Revisiting Broadband Performance. In *Proc. of IMC (2012)*, pages 273–286. ACM.

[42] N. Cardwell, Y. Cheng, C. S. Gunn, et al. BBR: Congestion-based Congestion Control. *Communications of the ACM (2017)*, 60(2):58–66.

[43] G. Casella, C. P. Robert, M. T. Wells, et al. Generalized Accept-reject Sampling Schemes. *A Festschrift for Herman Rubin (2004)*, pages 342–347.

[44] L. Chen et al. Auto: Scaling Deep Reinforcement Learning for Datacenter-scale Automatic Traffic Optimization. In *Proc. of SIGCOMM (2018)*, pages 191–205. USENIX.

[45] H. Dai, M. Shahzad, A. X. Liu, et al. Finding Persistent Items in Data Streams. In *Proc. of VLDB (2016)*, pages 289–300. VLDB Endowment.

[46] H. Deng, C. Peng, A. Fida, et al. Mobility Support in Cellular Networks: A Measurement Study on Its Configurations and Implications. In *Proc. of IMC (2018)*, pages 147–160. ACM.

[47] M. Dischinger, A. Haeberlen, K. P. Gummadi, et al. Characterizing Residential Broadband Networks. In *Proc. of IMC (2007)*, pages 43–56. ACM.

[48] M. Dong, Q. Li, D. Zarchy, et al. PCC: Re-architecting Congestion Control for Consistent High Performance. In *Proc. of NSDI (2015)*, pages 395–408. USENIX.

[49] M. Dong, T. Meng, D. Zarchy, et al. PCC Vivace: Online-Learning Congestion Control. In *Proc. of NSDI (2018)*, pages 343–356. USENIX.

[50] O. Goga et al. Speed Measurements of Residential Internet Access. In *Proc. of PAM (2012)*, pages 168–178. Springer.

[51] H. Goldstein, C. Poole, and J. Safko. *Classical mechanics*. American Association of Physics Teachers, 2002.

[52] S. Ha et al. CUBIC: a New TCP-friendly High-speed TCP Variant. In *Proc. of SIGOPS (2008)*, pages 64–74. ACM.

[53] N. Hu, L. Li, Z. M. Mao, et al. A Measurement Study of Internet Bottlenecks. In *Proc. of INFOCOM (2005)*, pages 1689–1700. IEEE.

[54] N. Hu, L. E. Li, Z. Mao, et al. Locating Internet Bottlenecks: Algorithms, Measurements, and Implications. In *Proc. of SIGCOMM (2004)*, pages 41–54. ACM.

[55] N. Hu and P. Steenkiste. Evaluation and Characterization of Available Bandwidth Probing Techniques. *IEEE Journal on Selected Areas in Communications (2003)*, 21(6):879–894.

[56] J. Huang et al. An In-depth Study of LTE: Effect of Network Protocol and Application Behavior on Performance. In *Proc. of SIGCOMM (2013)*, pages 363–374. ACM.

[57] A. Ivanov. Evaluating BBRv2 on the Dropbox Edge Network. *arXiv preprint arXiv:2008.07699 (2020)*.

[58] C. Jin, D. X. Wei, and S. H. Low. FAST TCP: Motivation, Architecture, Algorithms, Performance. In *Proc. of INFOCOM (2004)*, pages 2490–2501. IEEE.

[59] A. Lehman. *JMP for Basic Univariate and Multivariate Statistics: a Step-by-step Guide*. SAS Institute, 2005.

[60] M. Levandowsky and D. Winter. Distance Between Sets. *Nature*, 234(5323):34–35, 1971.

[61] F. Li, A. A. Niaki, D. Choffnes, et al. A Large-scale Analysis of Deployed Traffic Differentiation Practices. In *Proc. of SIGCOMM (2019)*, pages 130–144. ACM.

[62] Y. Li, H. Deng, C. Peng, et al. icellular: Device-customized Cellular Network Access on Commodity Smartphones. In *Proc. of NSDI (2016)*, pages 643–656. USENIX.

[63] Y. Li et al. HPCC: High Precision Congestion Control. In *Proc. of SIGCOMM (2019)*, pages 44–58. ACM.

[64] B. Melander, M. Bjorkman, and P. Gunningberg. Regression-based Available Bandwidth Measurements. In *Proc. of International Symposium on Performance Evaluation of Computer & Telecommunication Systems Conference (SPECTS) (2002)*, pages 14–19. IEEE.

[65] B. Melander et al. A New End-to-End Probing and Analysis Method for Estimating Bandwidth Bottlenecks. In *Proc. of GlobeCom (2000)*, pages 415–420. IEEE.

[66] R. Mittal, V. T. Lam, N. Dukkipati, et al. TIMELY: RTT-based Congestion Control for the Datacenter. In *Proc. of SIGCOMM (2015)*, pages 537–550. ACM.

[67] A. Narayanan, J. Carpenter, E. Ramadan, et al. A First Measurement Study of Commercial mmWave 5G Performance on Smartphones. *arXiv preprint arXiv:1909.07532 (2019)*.

[68] J. Navratil and R. L. Cottrell. ABwE: A Practical Approach to Available Bandwidth Estimation. In *Proc. of PAM (2003)*, pages 14–19. Springer.

[69] T. Oshiba. Accurate Available Bandwidth Estimation Robust against Traffic Differentiation in Operational MVNO Networks. In *Proc. of ISCC (2018)*, pages 694–700. IEEE.

[70] V. Ribeiro, R. Riedi, R. Baraniuk, et al. pathChirp: Efficient Available Bandwidth Estimation for Network Paths. In *Proc. of PAM workshop (2003)*. Springer.

[71] N. Sato, T. Oshiba, K. Nogami, et al. Experimental Comparison of Machine Learning-based Available Bandwidth Estimation Methods over Operational LTE Networks. In *Proc. of ISCC (2017)*, pages 339–346. IEEE.

[72] P. Schmitt et al. A Study of MVNO Data Paths and Performance. In *Proc. of PAM (2016)*, pages 83–94. Springer.

[73] J. Sommers and P. Barford. Cell vs. WiFi: On the Performance of Metro area Mobile Connections. In *Proc. of IMC (2012)*, pages 301–314. ACM.

[74] L. Song and A. Striegel. Leveraging Frame Aggregation for Estimating Wifi Available Bandwidth. In *Proc. of SECON (2017)*, pages 1–9. IEEE.

[75] S. Sundaresan, X. Deng, Y. Feng, et al. Challenges in Inferring Internet Congestion using Throughput Measurements. In *Proc. of IMC (2017)*, pages 43–56. ACM.

[76] L. Xu, K. Harfoush, and I. Rhee. Binary Increase Congestion Control (BIC) for Fast Long-distance Networks. In *Proc. of INFOCOM (2004)*, pages 2514–2524. IEEE.

[77] T. Yang, Y. Jin, Y. Chen, et al. RT-WABest: A Novel End-to-end Bandwidth Estimation Tool in IEEE 802.11 Wireless Network. *International Journal of Distributed Sensor Networks (2017)*, 13(2):1550147717694889.

[78] F. Zarinni, A. Chakraborty, V. Sekar, et al. A First Look at Performance in Mobile Virtual Network Operators. In *Proc. of IMC (2014)*, pages 165–172. ACM.

[79] Y. Zhu, H. Eran, D. Firestone, et al. Congestion Control for Large-scale RDMA Deployments. In *Proc. of SIGCOMM (2015)*, pages 523–536. ACM.

Toward Nearly-Zero-Error Sketching via Compressive Sensing

Qun Huang[1,2] Siyuan Sheng[3] Xiang Chen[1,2,4] Yungang Bao[3] Rui Zhang[5] Yanwei Xu[5] Gong Zhang[5]
[1]Peking University [2]Pengcheng Lab [3]Institute of Computing Technology, CAS
[4]Fuzhou University [5]Huawei Theory Department

Abstract

Sketch algorithms have been extensively studied in the area of network measurement, given their limited resource usage and theoretically bounded errors. However, error bounds provided by existing algorithms remain too coarse-grained: in practice, only a small number of flows (e.g., heavy hitters) actually benefit from the bounds, while the remaining flows still suffer from serious errors. In this paper, we aim to design a nearly-zero-error sketch that achieves negligible per-flow error for almost all flows. We base our study on a technique named compressive sensing. We exploit compressive sensing in two aspects. First, we incorporate the near-perfect recovery of compressive sensing to boost sketch accuracy. Second, we leverage compressive sensing as a novel and uniform methodology to analyze various design choices of sketch algorithms. Guided by the analysis, we propose two sketch algorithms that seamlessly embrace compressive sensing to reach nearly zero errors. We implement our algorithms in OpenVSwitch and P4. Experimental results show that the two algorithms incur less than 0.1% per-flow error for more than 99.72% flows, while preserving the resource efficiency of sketch algorithms. The efficiency demonstrates the power of our new methodology for sketch analysis and design.

1 Introduction

Sketch algorithms have been widely adopted in flow-level monitoring. They maintain compact data structures that sacrifice a small portion of accuracy to be readily deployable in commodity network devices. Given their limited overheads and provable high accuracy, numerous sketch algorithms are designed to monitor various flow statistics, such as per-flow counting [49], heavy hitters [19, 25], denial-of-service victims [26,84] and traffic distributions [46]. These flow statistics form essential building blocks for network management.

Despite the sound theoretical bounds on the errors, existing sketch algorithms remain far from perfect for providing comprehensive guarantees for all flows. Ideally, it is expected to monitor *every* flow with *minimum* errors, which empowers various fine-grained network management operations such as responsive diagnosis [17, 51, 67] and precise failure localization [3, 50]. However, the bounds in existing algorithms are designed for specific traffic statistics such as heavy hitters or flow distributions. They are too coarse-grained when applied to all flows. As a result, *only a small portion of flows actually benefit from the provable error bounds*. For instance, for byte counting, many sketch algorithms guarantee an upper bound of per-flow error. Heavy hitters whose size is much larger than the bound can certainly achieve high accuracy as the maximum possible error is limited compared to their size. Nonetheless, such a bound is still unacceptable for most small flows that still suffer from poor accuracy.

In this paper, our goal is to explore nearly-zero-error (NZE) per-flow monitoring. We aim to achieve a negligibly small error (e.g., >99.99% flows are reported, and the estimated size of any reported flow has a <0.1% relative error compared to the true size). We base our study on a signal processing technique named *compressive sensing*. Our key insight is that: (1) compressive sensing provides near-perfect signal recovery with limited resources, which inspires us to apply it to flow monitoring; (2) compressive sensing is built on various matrix properties such as sparsity, which provides a powerful tool to study sketch algorithms, given that most sketch algorithms exhibit the same mathematical form as compressive sensing [21]. Even though some telemetry solutions also adopt compressive sensing [7, 16, 21, 35, 44, 83], our work addresses the design of NZE sketch, which is never studied.

In particular, we exploit compressive sensing in two lines. In the *first line*, we incorporate the near-perfect recovery technique of compressive sensing by regarding flow statistics as signals. However, our preliminary experiments show that it is non-trivial to adopt compressive sensing directly. This motivates the *second line* of our work that examines the suitability of compressive sensing for sketch algorithms and then designs new algorithms accordingly. Specifically, we leverage compressive sensing to propose a novel and uniform methodology to study sketch techniques: we formulate various sketch algorithms in forms of matrices and then quantitatively analyze their suitability to compressive sensing. Thus, instead of designing from scratch, we use the analysis results as a guideline for the algorithm design.

In summary, we not only propose new algorithms but also provide a new methodology to study sketch techniques from a perspective of compressive sensing. We make the following contributions:

- We investigate the feasibility of applying compressive sensing to flow monitoring. We evaluate two simple methods and show that simple utilization either suffers from poor scalability or fails to reach the expected accuracy level.
- We dissect existing sketch algorithms based on compres-

sive sensing theory. We formulate each sketch algorithm by inducing a matrix for it. We examine a fundamental matrix property namely *orthonormality* that ensures the correctness of compressive sensing. We find that induced matrices of existing algorithms fail to be orthonormal.

- We study the common approaches to build sketch algorithms from a perspective of matrix analysis. We analyze the impact of these approaches on the orthonormality of their induced matrices. We reveal the limitations of existing algorithms when combining with compressive sensing.
- We design two new algorithms that efficiently utilize the common approaches to embrace compressive sensing seamlessly. The two algorithms target suitability to compressive sensing to achieve nearly-zero errors, while prior algorithms provide only coarse-grained error bounds. Further, their design choices can be interpreted by matrix analysis, while existing algorithms are built on statistical analysis or empirical observations on hash conflicts. To our best knowledge, both the two aspects are never explored before.
- We implement our proposed algorithms atop both OpenVSwitch [62] and P4 [63]. Our evaluation results demonstrate that our algorithms achieve less than 0.1% relative error for more than 99.72% flows, while incurring zero false negatives and zero false positives, while consuming limited resources compared to state-of-the-art algorithms. We release our source code at **`https://github.com/N2-Sys/NZE-Sketch`**.

2 Problem

2.1 Sketch-based Flow Monitoring

We follow the line of approximate flow-level measurement [4, 35, 37, 49, 53, 54, 80, 82]. Flow-level monitoring defines a flow as a sequence of packets with the same *flow ID*, and computes its *flow values* based on the packet sequence. We focus on sketch algorithms that outperform sampling in accuracy [49] and hence have been extensively used in flow monitoring. A sketch algorithm records information of *every* packet in a *compact* data structure, so as to achieve high accuracy yet be readily deployed in commodity measurement points.

In a nutshell, a sketch algorithm comprises a collection of counters. It supports two operations: *update* and *query*. The update operation is performed in the data plane. For each packet, it selects several counters with hash functions and updates the selected counters to reflect the changes of flow values. The query operation is invoked by the control plane. The control plane periodically collects sketch structures from each measurement point, and performs the query operation to extract flow IDs and flow values from the structure.

Sketch algorithms allow a small but bounded accuracy drop to reduce resource overheads. Specifically, in a sketch algorithm, a counter is typically shared by multiple flows, which inevitably incurs some errors due to flow conflicts. Each sketch algorithm mitigates the errors with its specific algorithmic design. Backed by sound mathematical analysis, sketch algorithms usually provide theoretical bounds on the errors.

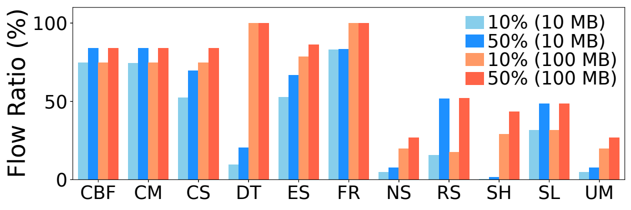


Figure 1: Fractions of flows that reach <10% and <50% per-flow errors in existing sketch algorithms.

2.2 Limitation

However, the theoretical guarantees provisioned by existing sketch algorithms are limited. Existing algorithms are typically designed to provide guarantees for specific flows (e.g., heavy hitters [8, 19, 25, 68] or super-spreaders [84]) and/or aggregated flow statistics (e.g., cardinality [28] or traffic distribution [46]). With regard to the specific scope, it is sufficient for a sketch algorithm to mitigate the overall hash conflicts only because bounding per-flow error is not a primary goal. Nonetheless, when extending an algorithm to the entire network traffic, the derived bounds are too coarse-grained to work for all flows. This leads to a considerable gap between theoretical analysis and practical results: only a small portion of flows actually benefit from the theoretical bounds, while the remaining flows still exhibit poor accuracy.

We consider an example of CountMin [20] to illustrate it. A CountMin sketch consists of r rows, each of which has w counters. When applying it to count per-flow bytes, it guarantees that the per-flow counting error is at most $\frac{2U}{w}$ with a high probability $1-\frac{1}{2^r}$, where U is the total byte count of all flows. Now we consider an interval with U =10 GB traffic, and configure $w = 10^5$ and a sufficiently large r such that the probability $1-\frac{1}{2^r}$ is close to one. In this case, the error bound is around 210 KB. For extremely large flows, such a bound guarantees a small error (e.g., <2% relative error for a flow of 10 MB). However, the error is awfully huge for small flows whose byte counts are below the bound. Given the heavy-tailed traffic distribution, most flows are small. Thus, most flows suffer from low accuracy due to the loose bound.

We justify this observation via trace-driven experiments. We consider 11 sketch algorithms for per-flow packet counting: Counting Bloom Filter (CBF) [27], CountMin (CM) [20], CountSketch (CS) [15], Deltoid (DT) [19], ElasticSketch (ES) [80], FlowRadar (FR) [49], NitroSketch (NS) [53], RevSketch (RS) [68], SeqHash (SH) [8], SketchLearn (SL) [37], and UnivMon (UM) [54]. We use Caida [9] traces and partition our traces in two 2-second intervals where each interval contains 100 K flows. We employ two configurations for each algorithm: one with 10 MB memory that is around the maximum available memory in commodity switches [43, 56], and the other with 100 MB that indicates an ideal scenario that has plenty of memory resources. We set parameters as suggested in the original papers. Figure 1 presents the fractions of flows whose per-flow error is below 10% and 50%. With 10 MB, less than half flows can reach a per-flow error below 10% in most algorithms. The low accuracy is caused by the serious hash conflicts in these sketches. With 100 MB, the overall accuracy is improved. However, such huge memory consumption is not affordable in commodity switches.

3 Overview

Goals. We explore the methodology to design NZE sketch algorithms. Specifically, we expect that: (1) flow IDs are extracted with a negligible error probability (e.g., both false positive rate and false negative rate are below 0.01%), and (2) per-flow error is small (e.g., <0.1%) for almost all (e.g., >99%) flows. At the same time, we also aim to limit the resource usage such that the algorithms can be readily deployed.

The NZE monitoring forms the basis for various flow statistics, such as flow cardinality [28], super-spreaders and DDoS victims [30, 86], heavy hitters/changes [45], flow distributions [46], and entropy [34]. For each type of statistics, a lot of specific algorithms have been proposed. However, to our best knowledge, none of existing algorithms provide comprehensive and strict accuracy guarantees for all flows. Prior studies advocate that: (1) it is sufficient to address large flows, and (2) approximate monitoring is acceptable. Nevertheless, NZE monitoring for even small flows greatly benefits network management. For example, single-packet TCP flows typically indicate unsuccessful connection attempts, caused by DDoS attacks, service crashes, or software bugs. Without accurate monitoring for small flows, it is difficult to rapidly react to such events. On the other hand, NZE monitoring allows administrators to deal with the reported anomalies without concerns on false alarms or undetected events.

Key idea. Our study addresses three questions. (1) Is NZE monitoring theoretically feasible? (2) What are the key factors to achieve NZE monitoring? (3) How do the key factors be efficiently realized in practice?

To answer these questions, we base our work on compressive sensing [11–13, 24]. Compressive sensing is a signal processing technique that acquires high-dimensional signals with limited resources. Classical compressive sensing has two procedures. The *sensing procedure* records a signal by multiplying the signal with a matrix, while the *recovery procedure* reconstructs the signal with an optimization-based approach. We exploit compressive sensing in two aspects:

- We aim to incorporate the optimization-based recovery of compressive sensing to achieve near-zero errors. This is motivated by the sound theoretical guarantees provisioned by compressive sensing on its overheads and correctness. It has been demonstrated that the optimization-based approach can recover signals nearly perfectly in many areas such as image compression [11, 12]. Thus, similar results are expected if we regard flow values as signals.
- We also leverage compressive sensing to guide the design of NZE sketch. Here, compressive sensing serves as a general framework to study various sketch algorithms. In particular, compressive sensing exhibits the same mathematical form as sketch algorithms: prior works [18, 21, 55] show that sketch algorithms can be viewed as variants of compressive sensing. Even though each sketch algorithm exhibits its unique design that is quite different from classical compressive sensing, it can be formulated by a matrix (§4.3) and analyzed via matrix analysis.

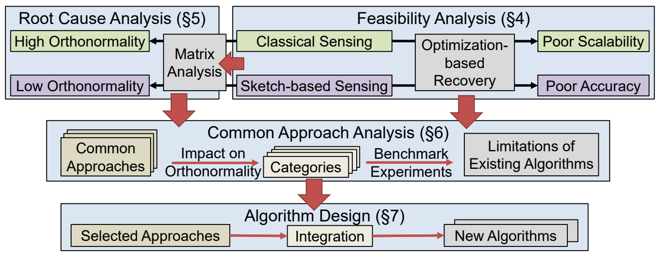


Figure 2: Workflow.

Assumptions. Our study makes two assumptions. First, network traffic is sparse. By sparsity, we mean that even though there are enormous possible flows (e.g., 2^{64} possible 2-tuple flows), the number of active flows is much smaller. This assumption has been justified in many measurement studies [5, 66] and utilized in various recent works [35, 83]. Second, we assume that a sketch algorithm contains a linear part in which each counter is updated linearly by a packet. Previous studies [18, 21] show that basic sketch algorithms, including CM, CS, and CBF, are linear structures; while we observe that many other sketches (e.g., UM, FR, ES, and SL) are built atop these basic sketches. We discuss how to handle the non-linear portion of a sketch in §4.3.

Workflow. Figure 2 outlines the workflow of our study.

- **Feasibility analysis (§4):** We investigate the feasibility of applying compressive sensing to flow monitoring. We consider two methods. The first method directly adopts classical compressive sensing, including its sensing and recovery procedures. The second method employs sketch algorithms to record per-packet information (referred to as *sketch-based sensing*). Then it formulates each sketch algorithm as a matrix and invokes the optimization-based recovery of classical compressive sensing. However, we find that the first method suffers from a scalability problem, while the second method has very low accuracy. This motivates us to dive into the fundamental theory of compressive sensing and design new algorithms.
- **Root cause analysis (§5):** We examine the root cause that leads to the poor accuracy when combining sketch-based

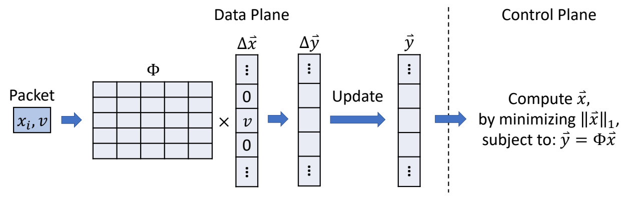


Figure 3: Utilization of compressive sensing for network monitoring.

sensing with the optimization-based recovery. Our study compares the matrices formulated by sketch algorithms with those in classical compressive sensing. We address a critical matrix property *orthonormality* (§1) that ensures the correctness of classical compressive sensing. Our benchmark experiments show that the matrices formulated by sketch algorithms lack sufficient orthonormality.

- **Common approach analysis (§6):** We identify common approaches to improve matrix orthonormality for sketch algorithms. Even though the approaches are also used in existing algorithms, we revisit these approaches in a novel methodology of matrix analysis. We group the approaches into four classes. For each class, we theoretically analyze the impact on matrix orthonormality. We also perform benchmark experiments to validate our analysis. Our analysis points out that the current utilization of these approaches is not efficient to combine with the optimization-based recovery of compressive sensing.
- **Algorithm design (§7):** We propose two algorithms that seamlessly combine sketch with the compressive sensing recovery. Based on the results of common approach analysis, we select appropriate approaches and efficiently integrate them to form the data plane of the two algorithms. Each design choice can be fully interpreted by matrix analysis. In the control plane, the two algorithms recover flow IDs and flow values by solving an optimization problem.

Discussion. Some recent studies have also applied compressive sensing in network measurement [7, 16, 21, 35, 44, 83]. However, they focus on recovering missing values in specific scenarios such as traffic matrices [16, 83] or network tomography [7]. In contrast, we utilize compressive sensing to (i) comprehensively dissect sketch algorithms, and (ii) guide the full design of NZE sketch.

Note that there are numerous variants of compressive sensing that reconstruct signals in different manners (e.g., LASSO [47] or using L_0 norm). In this paper, we focus on the original reconstruction approach that is based on matrix orthonormality [11, 12], given their simplicity and sound guarantees.

4 Feasibility Analysis

We introduce the fundamental concepts of compressive sensing in §4.1. Then we study two methods that apply compressive sensing to flow monitoring in §4.2 and §4.3, respectively.

4.1 Preliminary

Compressive sensing represents a signal as a *signal vector* $\vec{x}$ of length n. It includes two procedures to acquire $\vec{x}$.

Sensing procedure. The sensing procedure is responsible for recording $\vec{x}$ in a lightweight manner. Since the length n is usually a large number, compressive sensing linearly maps $\vec{x}$ into a *measurement vector* $\vec{y}$ of length m, where m is much smaller than n. Formally, the mapping can be represented as an $m \times n$ *sensing matrix* Φ, while $\vec{y}$ is computed as:

$$\vec{y} = \Phi \times \vec{x} \tag{1}$$

Recovery procedure. The recovery procedure is to reconstruct the signal vector $\vec{x}$ with Φ and $\vec{y}$. However, Equation (1) is an underdetermined system. It includes m linear equations for the n unknown variables in $\vec{x}$: the i-th element in $\vec{y}$ (denoted by y_i) and the i-row of Φ form a linear equation: $\sum_{j=1}^{n} \Phi_{i,j} \cdot x_j = y_i$. Since the number of variables n is much larger than the number of equations m, the number of possible solutions is infinite.

Compressive sensing addresses the underdetermined problem by introducing some prior knowledge. It assumes that $\vec{x}$ is sparse[1]. Then compressive sensing formulates an optimization problem:

$$\begin{aligned} \text{minimize:} \quad & \|\vec{x}\|_1, \\ \text{subject to:} \quad & \vec{y} = \Phi\vec{x} \end{aligned} \tag{2}$$

Here, compressive sensing chooses to minimize the L_1 norm of $\vec{x}$ because L_1 norm penalizes against the lack of sparsity [11, 12]. Therefore, a sparse vector satisfying Equation (1) is obtained. Theoretical analysis shows that the solution is close to the true $\vec{x}$ if some specific properties hold in Φ [11–13, 24]. We will study the properties in §5.

Utilization. Figure 3 depicts how to map the concepts of compressive sensing to those in network monitoring. Let n be the number of possible flows. For each type of flow statistic, all flow values form a vector $\vec{x}$ of length n. An element x_i indicates the value of the flow i. A measurement point maintains $\vec{y}$ in its memory. For each packet, it identifies the flowkey i such that the packet can be considered as a change to $\vec{x}$ denoted by $\Delta\vec{x}$. The measurement point multiplies $\Delta\vec{x}$ with the matrix Φ to form the update to $\vec{y}$ (denoted by $\Delta\vec{y}$). Then it applies the update to $\vec{y}$. The control plane collects $\vec{y}$ and invokes the optimization-based recovery in Equation (2) to reconstruct $\vec{x}$. Note that we do not need to explicitly maintain Φ, $\vec{x}$, $\Delta\vec{x}$, and $\Delta\vec{y}$ in memory. Instead, we compute their elements on demand. For example, we compute an element $\Phi_{k,i}$ when we update the k-th counter in $\vec{y}$ with flow indexed by i.

In practice, network administrators can directly utilize a classical matrix Φ [12, 75]. They can also propose their own method and formulate it in the form of Equation (1). We study

[1]For non-sparse $\vec{x}$, it needs to be transformed to another sparse vector first. We omit this case because network traffic exhibits high sparsity (§3).

the classical methods in §4.2 and formulate sketch algorithms using compressive sensing in §4.3.

4.2 Method 1: Classical Sensing

Accuracy. We first consider a method that direct utilizes classical sensing matrix Φ. We evaluate the accuracy of the classical sensing method via experiments with the same setup as that in §2.2. We employ four types of commonly used sensing matrices Φ: (1) Gaussian Matrix (GM) [75], (2) Bernoulli Matrix (BM) [12], (3) Incoherence Matrix (IM) [12], and (4) Fourier Matrix (FM) [12]. To reconstruct $\vec{x}$, we leverage two algorithms: the L_1 minimization approach that solves the optimization problem with the simplex method [22], and a greedy algorithm named Orthogonal Matching Pursuit (OMP) [64]. The four types of sensing matrices and two recovery algorithms produce eight approaches in total. The results show that all the eight approaches can recover flow IDs and flow values perfectly: zero false positives, zero false negatives, and zero per-flow error. Our results show that 400 KB memory is sufficient to achieve perfect recovery, which is much smaller than sketch algorithms (§2.2). The detailed accuracy trend with different memory settings is in Table 3 in Appendix.

Scalability problem. However, the classical sensing method suffers from a scalability problem. The classical sensing matrices are dense matrices in which all elements are non-zero. Thus, each packet needs to update m (above 10^4) counters in $\vec{y}$. This is infeasible for commodity devices. In software switches, updating so many counters fails to keep pace with packet streams with the slow CPUs. In hardware switches, the updates far exceed the available computational units. Thus, classical sensing can only accommodate limited flows.

Note that the scalability problem does not occur in other compressive sensing applications in which $\vec{x}$ does not vary (e.g., image compression). In those scenarios, $\vec{y}$ is computed only once using the constant $\vec{x}$.

4.3 Method 2: Sketch-based Sensing

Matrix formulation. Sketch algorithms incur limited per-packet operations, which addresses the scalability problem in §4.2. we follow prior studies [18, 21] that regard sketch as *linear* mapping and formulate it in the form of $\vec{y} = \Phi\vec{x}$. Let m be the number of linear counters. For $\vec{y}$, we index the m counters and stack them as a vector $\vec{y}$ of length m. For Φ, we form Φ with m rows and n columns, where each column represents a flow while each row represents one counter in $\vec{y}$. Each element $\Phi_{i,j}$ implies that the counter y_i is incremented by $\Phi_{i,j}$ if the value x_j of flow j changes by one.

Examples. We present an example of CountMin in Figure 4. We consider four flows, i.e., $\vec{x}$ has its length n=4. We employ two rows and configure three counters in each row in the sketch. Hence, $\vec{y}$ has length $m = 6$. In each row, a packet (k, v)

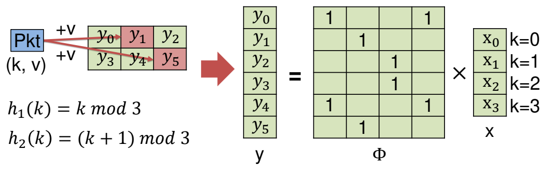


Figure 4: Matrix formulation of CountMin.

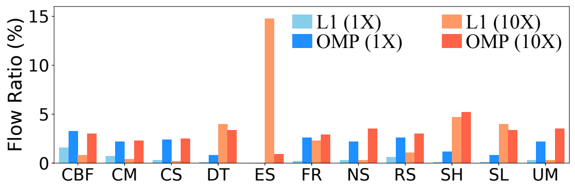


Figure 5: Ratio of flows that reach <50% per-flow errors in sketch-based sensing.

selects one counter with a hash function and increments it by v. Thus, we have $\Phi_{i,k} = 1$ if flow k is hashed to counter i, and $\Phi_{i,k} = 0$ otherwise. For each more flow, we may add a column to Φ and derive the column elements in the same method. We present more examples in Appendix.

Nonlinear structures. Not all components in a sketch are linear mapping. These components cannot be formulated by matrices. For example, FlowRadar maintains a set of counters that encode flow IDs via XOR operations [49]. We do not incorporate such nonlinear components in the optimization problem when reconstructing $\vec{x}$. Instead, we employ them to verify the correctness of the reconstructed $\vec{x}$. Specifically, we recompute these nonlinear structures with the reconstructed $\vec{x}$ and compare them with the original ones. For example, in FlowRadar, we encode all recovered flow IDs via the same XOR operations. We compare the new encoded results with the original XOR results to validate the correctness.

Results. We evaluate the sketch-based sensing method. We consider the sketch algorithms in §2.2. For each algorithm, we perform both L_1 minimization and OMP for the recovery. We employ two memory configurations: one with the same amount of memory as classical sensing, and the other with 10× memory. Figure 5 presents the ratio of flows whose error is below 50%. We see that even with 10× memory, all algorithms suffer from extremely low accuracy when using the optimization-based recovery. The results are even worse than those using their original query operations (see §2.2). The reason is that the matrices derived from existing sketch algorithms do not exhibit the required properties of compressive sensing although they exhibit the same form (see 5). This suggests us to explore new methods to boost sketch-based sensing to embrace compressive sensing, provided by the extensive study on the accuracy of compressive sensing [11–13, 24].

5 Root Causes

We examine the root cause of the poor accuracy in §4.3. Our methodology is to examine whether key properties of classical

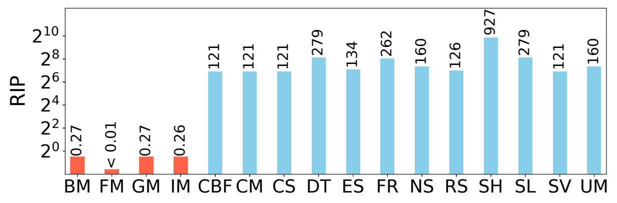


Figure 6: RIP of classical and sketch-based sensing.

sensing hold in sketch-based sensing.

Key properties. Compressive sensing guarantees its correctness by two properties: the *sparsity* of $\vec{x}$ and the *orthonormality* of Φ. Specifically, any orthonormal matrix Φ preserves the norms (and hence differences) for sparse vectors: given arbitrary two different sparse vectors $\vec{x}_1$ and $\vec{x}_2$, their mappings under matrix Φ (i.e., $\Phi\vec{x}_1$ and $\Phi\vec{x}_2$) remain distinct. Thus, when a sparse vector $\vec{x}^*$ that satisfies $\Phi\vec{x}^* = \vec{y}$ is found, it is must be equal to the desired $\vec{x}$ (otherwise two different vectors $\vec{x}^*$ and $\vec{x}$ have the same mapping, which compromises the property of the orthonormal matrix Φ) [12]. Since $\vec{x}$ is already sparse (§3), we only address whether Φ is orthonormal.

Orthonormal matrix and RIP. Unfortunately, orthonormality cannot hold in Φ, because an orthonormal matrix is requires to be a square matrix, but the number of rows is smaller than the number of columns in Φ. Compressive sensing deals with this issue with a notion of *restricted isometry property (RIP)* [10], which serves as an approximation of being fully orthonormal. RIP characterizes the extent to which Φ preserves the norm of sparse signals.

At a high level, for any sparse vector $\vec{x}$, $\Phi\vec{x}$ is its mapping under the matrix Φ. If Φ is highly orthonormal, the norm of $\Phi\vec{x}$ (denoted by $||\Phi\vec{x}||_2$) must be close to that of $\vec{x}$ (denoted by $||\vec{x}||_2$). Therefore, RIP evaluates the difference between $||\vec{x}||_2$ and $||\Phi\vec{x}||_2$. Since Φ should work for an arbitrary sparse vector $\vec{x}$, RIP is calculated as a sequence of *isometry constants* $\{\delta_S\}$. Each δ_S in the sequence is the maximum relative difference between the norms of $\Phi\vec{x}$ and $\vec{x}$ among all S-sparse signals:

$$\delta_S = \sup\{\frac{|\,||\Phi\vec{x}||_2 - ||\vec{x}||_2\,|}{||\vec{x}||_2} \text{ for any S-Sparse } \vec{x}\} \qquad (3)$$

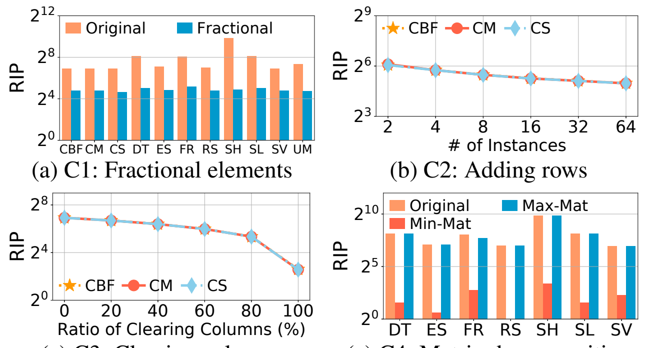
Benchmark results. We measure the RIP of both classical sensing matrices (§4.2) and the matrices induced by sketch algorithms (§4.3). We present RIP as δ_S, the isometry constant for S-sparse vectors, where S is the number of actual flows in each interval. Figure 6 shows that classical sensing matrices have RIP below 0.3. In contrast, RIP is above 120 in all sketch-induced matrices. The large RIPs degrade the efficiency of compressive sensing reconstruction.

6 Common Approach Analysis

We examine common approaches in general sketch design. Based on their impacts on matrix orthonormality, we categorize the approaches into four classes. For each class, we analyze its matrix property and use RIP as the metric to quantify the effectiveness. Although some approaches have been used in existing sketch algorithms (see Table 1), we study them from novel perspectives. First, we target the suitability of these approaches to compressive sensing, while existing algorithms study them for specific purposes. Second, we quantify the efficiency of these approaches via matrix analysis, while previous algorithms address probabilistic error bounds.

Algorithm	**C1**	**C2**	**C3**	**C4**
CU Sketch [25]	Conservative update			
Deltoid [19]		Multiple CM instances		Flow extraction
ElasticSketch [80]				Traffic splitting
FlowRadar [49]		Multiple Bloom Filters	Bloom Filter	Flow extraction
NitroSketch [53]	Sampling	Multiple CS instances	Heap	
RevSketch [68]				Flow extraction
SeqHash [8]		Multiple CM instances		Flow extraction
SketchLearn [37]		Multiple CM instances		Flow extraction
SketchVisor [35]				Traffic splitting
UnivMon [54]		Multiple CS instances	Heap	
SeqSketch	Fractional update		Bloom Filter + Controller	Splitting + Controller
EmbedSketch	Fractional update		Bloom Filter + controller	Extraction + Controller

Table 1: Common approaches in sketch algorithms.



(a) C1: Fractional elements (b) C2: Adding rows

(c) C3: Clearing columns (c) C4: Matrix decomposition

Figure 7: Impact of common approaches.

6.1 Class 1 (C1): Fractional Elements

Matrix analysis. We observe that elements in sketch-based sensing matrices are integers, which leads to the norm of matrix columns above one. However, an orthonormal matrix requires column vectors with norm one. Thus, the first class is to employ fractional matrix elements whose values are less than one, such as to reduce the norm of each column.

Benchmark results. Figure 7(a) evaluates the impact of fractional elements in existing sketch algorithms. For a sketch, we replace each element with a randomized value $1/\sqrt{t} + \sigma$, where σ is sampled from a Gaussian distribution with its mean equal to zero. Thus the mean of elements in the matrix is $1/\sqrt{t}$. Here, t is the number of counters accessed by a packet. Thus, the expected norm of each column vector is one. We see that RIP is decreased by 40% in all cases.

Approaches. In existing algorithms, there are two approaches producing fractional elements.

- **Sampling:** Sampling techniques [69, 70] discard some packets. For each flow, only partial packets contribute to $\vec{y}$. Thus, the elements in Φ are less than one. NitroSketch [53] has combined an adaptive sampling in its design.
- **Conservative updates:** In general, a packet incurs several updates in a sketch algorithm. Conservative update [25] preserves only the smallest update and drops the others. This also leads to smaller elements in Φ because not all updates are included.

Limitation. However, sampling and conservative updates are hard to be formulated as matrices. To obtain the exact fractional elements in Φ, it needs to track exact per-flow packet loss or update drops, which inevitably incurs excessive overheads and cancels out the benefit of sketch.

6.2 Class 2 (C2): Adding Rows

Matrix analysis. The second class is to add more rows to the matrix Φ. Ideally, two columns are orthonormal if and only if their nonzero elements occur in different positions. This implies that the two flows have no conflicts in all counters. Since each counter contributes one row in Φ (§4.3), adding rows means to configure more counters to reduce flow conflicts. Hence, column vectors become more orthonormal.

Approaches. In addition to simply allocating more counters to a single sketch, a common approach is to use multiple instances of sketch structures. For instance, FlowRadar [49] contains two Bloom Filters; UnivMon [54] employs multiple CS instances and filters flows for each instance; Deltoid [19] and SketchLearn [37] maintain multiple CM instances while each instance is updated based on the bits of flow IDs.

Benchmark results. Figure 7(b) shows RIP with respect to various number of sketch instances. We consider three commonly used basic sketch algorithms: CM, CS, and CBF. We see that the RIP decreases as the number of instances grows. With 64 instances, RIP is reduced by nearly 75%.

Limitation. However, the resulting RIP is still much higher than that of classical sensing matrices (§5). Although we can further reduce RIP with more instances, adding instances consumes more memory. It also incurs excessive usage of computational resources to update multiple instances.

6.3 Class 3 (C3): Clearing Columns

Matrix analysis. The third class is to clear elements of some columns. Recall that a column indicates the contribution of an unknown variable (§4.3). Clearing one column means to exclude a variable, which simplifies the optimization problem and hence improves accuracy.

Approaches. Identifying columns that can be cleared is equivalent to detecting flow IDs that never occur, such that discarding the flows does not compromise the results. Two approaches can track flow IDs in existing algorithms.

- **Heap:** CountMin [20], UnivMon [54] and NitroSketch [53] use a heap to store flow IDs whose flow values satisfy specific conditions (e.g., above a pre-defined threshold).
- **Bloom Filter:** Bloom Filter records Flow IDs compactly with bit arrays. An example is FlowRadar [49] that uses Bloom Filter to avoid duplicate flow IDs.

Benchmark results. Figure 7(c) measures how RIP varies as the ratio of cleared useless columns. Due to the interest of space, we present three sketches here and put the remaining results in Appendix. With no columns cleared, RIP is above 120 for all the three sketch algorithms. RIP significantly decreases as the number of cleared columns grows. It becomes 6 when all useless columns are cleared.

Limitation. However, tracking all Flow IDs with existing approaches is bounded by resource restrictions in switches. For the heap-based approach, per-flow tracking is infeasible due to the memory usage of heap. For BF, since it only examines the occurrence of flow IDs, extra resources are needed to store flow IDs (e.g., XOR arrays in FlowRadar [49]). For the controller-based approach, it needs careful design to avoid bandwidth exhaustion.

6.4 Class 4 (C4): Matrix Decomposition

Matrix analysis. The final class decomposes Φ as the sum of several component matrices. The decomposition distributes non-zero elements in Φ into different components. Thus, their conflicts are alleviated and hence each component becomes more likely to be orthonormal.

Approaches. There are two possible approaches to distribute flows and hence decompose matrix Φ.

- **Traffic splitting:** Traffic splitting employs multiple algorithmic parts in the data plane and splits traffic into different parts. Each part can produce a component matrix individually. For example, SketchVisor [35] maintains a fast path and a normal path, and directs traffic into either path based on real-time workloads. ElasticSketch [80] consists of a heavy part and a light part: traffic that is evicted from the heavy part enters the light part.
- **Flow extraction:** We can also form a component matrix by extracting flows from the sketch structure. These algorithms usually embed specific features in the sketch structures for the extraction. For example, FlowRadar [49] estimates the number of distinct flows in each counter (i.e., a row in Φ) and iteratively extracts from the counters with exactly one flow. Deltoid [19] and SketchLearn [37] extract from rows with large corresponding flow values.

Benchmark results. Figure 7(d) compares RIP before and after the decomposition. We present the component matrices with the minimum RIP (Min-Mat) and maximum RIP (Max-Mat). We observe that the orthonormality is significantly improved in each Min-Mat. For example, the original RIP of DT is nearly 1000, but it is reduced to 2.92 in Min-Mat.

Limitation. However, Max-Mat still exhibits high RIP in all algorithms. For some algorithms, the RIP of Max-Mat is close to that in the original matrix. We find that the decomposed traffic is limited because it needs extra structures to split traffic or extract flows. When resources are bounded, limited traffic can be decomposed.

7 New Algorithms

Motivation. §6 points out that existing algorithms fail to produce highly orthonormal matrices. The key issue is that they are not tailored for compressive sensing. On the one hand, the four classes C1-C4 are not realized efficiently. On the other hand, they do not collectively combine the approaches for compressive sensing. Thus, we need new algorithms that realize and combine the common approaches more efficiently.

Design choices. To better embrace compressive sensing, we examine each class in §6 to employ appropriate approaches to combine with compressive sensing.

- **C**1: As both sampling and conservative updates are hard to formulate, we realize a novel method of fractional updates. Specifically, for a packet (k,v), we use an additional hash function $g(\cdot)$ to change its value from v to $v \cdot g(k)$. Here, $g(\cdot)$ generates a value with its mean equal to $\frac{1}{\sqrt{r}}$ where r is the number of rows in the sketch. Thus, the expected norm of column vectors is reduced to one.
- **C**2: We discard C2 because of its excessive resource usage.
- **C**3: We employ a control-based approach to enhance Bloom Filter. Specifically, we store flow IDs in the controller. Since the controller has enough memory, the flow IDs can be recorded with zero errors. To reduce bandwidth usage, we employ a Bloom Filter to eliminate duplicate transfers.
- **C**4: We separate large flows as key-value pairs and small flows in the sketch with fractional values (referred to as fractional sketch). It has been proved that such separation can be realized with limited overheads [35,71,80]. To make *each* decomposed component matrix highly orthonormal, we also leverage the controller to steer the traffic in key-value pairs and the fractional sketch.

In summary, we maintain three types of components in the data plane: (1) key-value pairs to track large flows, (2) fractional-valued sketch to record small flows and (3) a Bloom Filter. We propose two algorithms that combine them in different manners. The first algorithm SeqSketch arranges the components sequentially, while the second algorithm EmbedSketch embeds the key-value table and Bloom Filter into the sketch arrays such that key-value pairs can be extracted

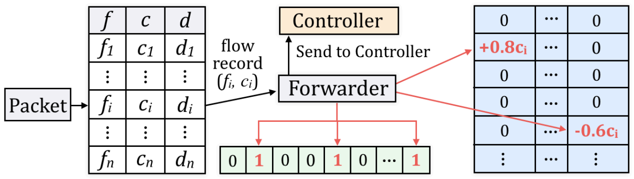


Figure 8: Structure of SeqSketch.

Algorithm 1 SeqSketch Data Plane

```
Input: Packet $(k,v)$
 1: procedure UPDATE$(k,v)$
 2:     $j = \text{hash}(k)$
 3:     if $H[j]$ is $\emptyset$ then
 4:         $H[j].f = k$, $H[j].c = v$, and $H[j].d = 0$
 5:     else if $H[j].f == k$ then
 6:         $H[j].c = H[j].c + v$
 7:     else
 8:         $H[j].d = H[j].d + v$
 9:         if $H[j].d > H[j].c$ then
10:             Send $(H[j].f, H[j].c)$ to controller
11:             $H[j].f = k$, $H[j].c = v$, and $H[j].d = 0$
12:         else
13:             for all row $i$ in $FS$ do
14:                 Compute $j = h_i(k)$
15:                 Increment counter $(i,j)$ by $g_i(k) \cdot v$
16:             if $k \notin BF$ then
17:                 Send $k$ to controller
18:                 Insert $k$ to $BF$
```

from sketch buckets. SeqSketch consumes less memory, while EmbedSketch needs fewer computational units. Network administrators can select the more suitable algorithm based on their resource budget.

7.1 SeqSketch

Data structure. Figure 8 presents an overview of SeqSketch. SeqSketch organizes its key-value pairs in a hash table H, and employs a *forwarder* to connect the hash table, Bloom Filter BF and fractional sketch FS. Every packet first enters the hash table H. Each tuple in H has three fields to identify large flows: apart from flow ID f, two counters c and d record flow values belonging and not belonging to f, respectively. The hash table evicts records of potential small flows based on the two counters when conflicts occur. The forwarder transfers an evicted record if it is a new flow, or sends it to the fractional sketch. It uses the Bloom Filter to record all occurred flows and examine new flows. The eviction is cheap because it incurs limited operations for each evicted record. Given the heavy-tailed distribution of network traffic, the hash table H absorbs a large portion of traffic, which alleviates the memory usage of BF and FS. Thus, SeqSketch is memory efficient.

Data plane. Algorithm 1 outlines how SeqSketch processes

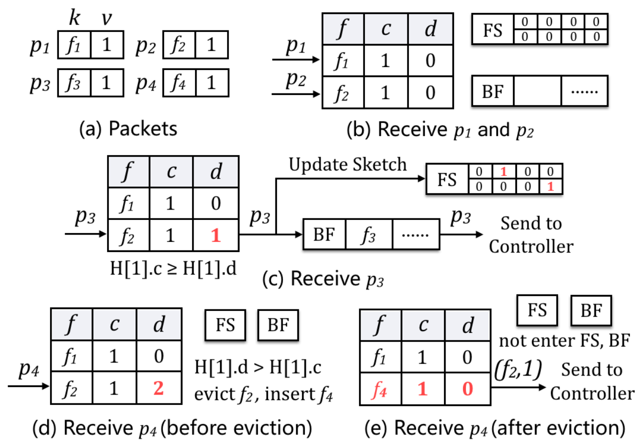


Figure 9: Example of SeqSketch.

a packet. When a packet (k, v) arrives, we first compute its position in the hash table H (Line 2). If its hashed entry is empty, a new record is created for this packet (Lines 3-4). If the entry already exists, there are two cases. First, if the existing entry has the same ID as the packet, the counter c is incremented (Lines 5-6). Second, if f and k are different, we add d by v (Line 8). At the same time, we need to evict the flow record of f or the packet (k, v) (Lines 9-18). When d is larger than c, we send the record to the controller (Line 10), and insert a flow record for k (Line 11). Otherwise, we evict (k, v) to FS (Line 13-15). In this case, the forwarder queries its Bloom Filter to examine whether the flow ID appears before. If it is a new flow, the ID is also forwarded to the controller (Line 16-18). Note that we increment each counter in FS with a fractional value instead of v (Lines 14-15).

Example. Figure 9 presents an example with two buckets in H. H directly inserts the first two packets p_1 and p_2 (Figure 9(b)). For the third packet p_3, H maps it to $H[1]$ that already records another flow f_2 . To deal with the conflict, H increments $H[1].d$. Since $H[1].d$ does not exceed $H[1].c$, p_3 is delivered to FS and BF. FS updates its counters with p_3, while BF transfers its flow ID to the controller because it is a new flow (Figure 9(c)). Finally, p_4 enters H and is also hashed to $H[1]$. Since p_4 does not belong to f_2, H increments $H[1].d$ by one. Since $H[1].d$ exceeds $H[1].c$ (Figure 9(d)), we evict $H[1]$ and insert p_4 (Figure 9(e)).

Control plane. We recover flow IDs and flow values by formulating an optimization problem. There are three portions of traffic: that in the hash table H, that transferred to the controller, and that in FS. Denote flow values in the three portions by $\vec{x}_H$, $\vec{x}_C$ and $\vec{x}_S$, respectively. Since $\vec{x}_H$, $\vec{x}_C$ can be obtained directly, we only need to solve $\vec{x}_S$ by formulating the per-flow update of FS as Φ and its counters as $\vec{y}$:

$$\begin{aligned} \text{minimize:} \quad & \|\vec{x}_S\|_1, \\ \text{subject to:} \quad & \vec{y} = \Phi \vec{x}_S \end{aligned} \tag{4}$$

Algorithm 2 EmbedSketch Data Plane

```
Input: Packet $(k, v)$
1: function UPDATEBUCKET($k$, $v$, $i$, $j$)
2:     $V_{i,j} = V_{i,j} + g_i(k)$
3:     if $f_{i,j}$ is empty then
4:         $f_{i,j} = k$, $c_{i,j} = v$, $d_{i,j} = 0$
5:     else if $f_{i,j}$ is $k$ then
6:         $c_{i,j} = c_{i,j} + v$
7:     else
8:         $d_{i,j} = d_{i,j} + v$
9:         if $d_{i,j} > c_{i,j}$ then
10:            Send $(f_{i,j}, c_{i,j})$ to controller
11:            $f_{i,j} = k$, $c_{i,j} = v$, $d_{i,j} = 0$
12:        else
13:            if $k \notin BF_{i,j}$ then
14:                Send $k$ to controller
15:                Insert $k$ to $BF_{i,j}$
16:
17: procedure UPDATE($k, v$)
18:     for row $i = 1, 2, ..., r$ do
19:         $j = h_i(k)$
20:         UPDATEBUCKET($k$, $v$, $i$, $j$)
```

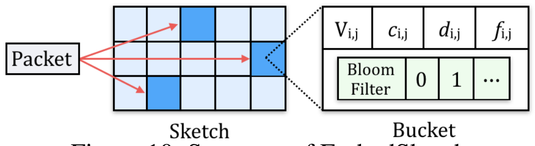


Figure 10: Structure of EmbedSketch.

7.2 EmbedSketch

Data structure. Figure 10 depicts EmbedSketch. It maintains a sketch with r rows. Each row i has two hash functions (h_i to select counters and g_i to generate fractional values) and w buckets. A bucket (i, j) in EmbedSketch consists of: (i) a counter $V_{i,j}$, which denotes the total values hashed to this bucket, (ii) $f_{i,j}$, which denotes the flow ID of the candidate for the largest flow in the bucket, (iii) $c_{i,j}$, which denotes the aggregated value of f, (iv) $d_{i,j}$, which denotes the total value of other flows in the bucket, and (v) a Bloom Filter $B_{i,j}$ that records flow IDs in this bucket. Essentially, EmbedSketch distributes monitoring operations in its buckets. This mitigates per-bucket hash conflicts. Thus, one hash function in each bucket is sufficient (see §7.4).

Data plane. Algorithm 2 details how EmbedSketch processes a packet (k, v). For each row i, EmbedSketch computes a bucket with $h_i(k)$ and updates the bucket (Lines 18-20). To update a bucket (i, j), EmbedSketch first increments $V_{i,j}$ by $g_i(k) \cdot v$ (Line 2). If the existing candidate $f_{i,j}$ equals to k, $c_{i,j}$ is also incremented (Lines 5-6). Otherwise, EmbedSketch increments $d_{i,j}$ (Line 8) and determines to evict either $f_{i,j}$ (Lines 9-11) or k (Lines 13-15). If $f_{i,j}$ is evicted, a record $(f_{i,j}, c_{i,j})$ is transferred to the controller (Line 10). At the same time, EmbedSketch uses k as the new candidate and sets $c_{i,j} = v$ and $d_{i,j} = 0$ (Line 11). Otherwise, if k is evicted, EmbedSketch queries its local Bloom Filter $B_{i,j}$ (Line 13). If k is a new ID, EmbedSketch forwards it to the controller and updates $B_{i,j}$ to include k (Lines 14-15).

Control plane. We form $\vec{y}$ with all $r \times w$ counters of $V_{i,j}$ in EmbedSketch. Traffic in $V_{i,j}$ comprises three portions: (1) each $f_{i,j}$ contains its value $c_{i,j}$, (2) per-flow values transferred to the controller, and the remaining traffic in $V_{i,j}$. Thus, we denote per-flow values in the three portions by $\vec{x}_f$, $\vec{x}_C$ and $\vec{x}_R$, respectively. Since only $\vec{x}_R$ is unknown, we build the following optimization problem:

$$\begin{aligned} \text{minimize:} \quad & \|\vec{x}_R\|_1, \\ \text{subject to:} \quad & \vec{y} = \Phi(\vec{x}_f + \vec{x}_C + \vec{x}_R) \end{aligned} \tag{5}$$

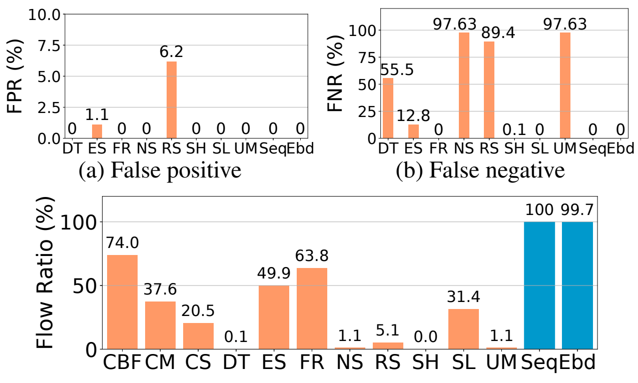
7.3 Parameters

We need to configure the three components: the fractional sketch (FS), the Bloom Filter (BF), and key-value pairs (KV).

Fractional sketch. For FS, two rows are sufficient as our optimization-based recovery does not need many rows to alleviate hash conflicts. However, compressive sensing theory requires a minimum amount of counters: $C \cdot S \log_2(n/S)$ (c.f. Equation(13) in [13]), where n is the number of possible flows, S is the expected number of actual flows, and C is a small positive number. In practice, we can select a proper C to make memory usage fits the device. For example, to monitor around S=100K 2-tuple flows ($n = 2^{64}$), setting C=0.1 leads to around 472K counters. If we employ 32-bit counters, the total memory of FS is 1888 KB.

Bloom Filter. The Bloom Filter BF determines the accuracy of the received flow IDs in the control plane. A false flow ID indicates 100% relative error for that flow, which seriously compromises the recovery accuracy. Thus, we need to carefully configure BF. The size of BF depends on the expected number of flows S. According to [18] (c.f. §5.2.5), the false positive rate of Bloom Filter is $(0.6185)^{m/S}$ where m is the length of Bloom Filter, if we set the number of hash functions to its optimal value $\frac{m}{S} \ln 2$. In our case, a false positive in Bloom Filter means wrongly clearing a column in Φ. To achieve our goal of $< 1\%$ error probability for flow ID extraction, we need to bound the false positive rate of the Bloom Filter below 1%. This requires $m = 9.6S$. For $S = 100$ K flows, this leads to a Bloom Filter with 120 KB. For SeqSketch, we employ the optimal number of hash functions: $9.6 \ln 2 \approx 7$. For EmbedSketch, since the Bloom Filter is distributed across buckets, one hash suffices to achieve low error probability.

Key-value pairs. In EmbedSketch, each bucket maintains one key-value pair by design. For SeqSketch, it can employ the same amount for simplicity.



(a) False positive (b) False negative

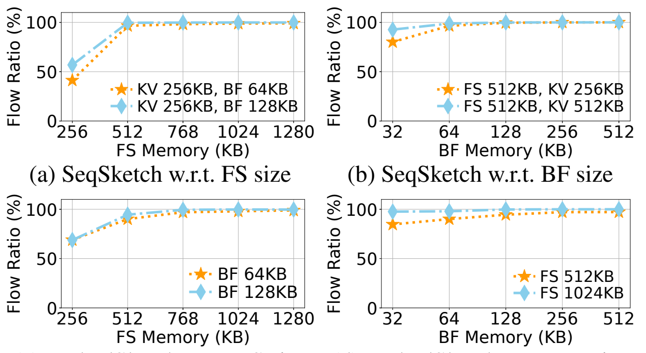
(c) Fractions of flows with relative errors less than 0.1%

Figure 11: (Experiment 1) Accuracy.

7.4 Evaluation

Setup. We implement both software version and hardware version of the two algorithms (see Appendix). We evaluate them via trace-driven experiments. We use the CAIDA-2018 backbone trace [9] and two data center traces [5]. We present 2-tuple flows and count their packets, while other flow definitions (e.g., 5-tuple) and statistics have similar results. We partition each trace into equal-length intervals. Due to the interest of space, we present the results with 2-second intervals, each of which has around 100 K flows. More results are in Appendix. We present the average results across all intervals. Here, we omit the standard deviations because the standard deviations are negligible. When measuring accuracy (Experiments 1 and 2), we run both update and query operations in a server with 36 CPU cores (2.6GHz each) and 128 GB memory to process the traces. When measuring resource overheads (Experiments 3 to 6), we build a testbed with 16 servers and a Barefoot Tofino switch [79]. Each server has a 40Gbps NIC for traffic transfers and a 10Gbps NIC to connect to the controller. We deploy our algorithms in the switch. Each server replays our traces and evenly sends the traces to others. We follow §7.3 to configure key-value pairs (KV), the Bloom Filter BF, and the fractional sketch FS.

Experiment 1: Accuracy (Figure 11). We compare the accuracy of SeqSketch (Seq) and EmbedSketch (Ebd) with 11 sketch algorithms. Every algorithm has its suggested theoretical configuration (see Appendix). However, they fail to achieve NZE even when we allocate 100 MB memory (§2.2). Thus, for a fair comparison, each algorithm is allocated with the same amount of memory as ours for a stress test. In Figure 11(a) and Figure 11(b), we exclude CBF, CM, and CS because they cannot extract flow IDs by design. We find that more than half existing algorithms have nearly zero false positive rate and false negative rate. However, none of existing algorithms achieve high accuracy for all flows. Recall §2.2, even when we allocate more memory (10 MB) and relax the desired per-flow error to 50%, the ratio of accurate flows in existing algorithms is less than 85%. In contrast, EmbedS-



(a) SeqSketch w.r.t. FS size (b) SeqSketch w.r.t. BF size

(c) EmbedSketch w.r.t. FS size (d) EmbedSketch w.r.t. BF size

Figure 12: (Experiment 2) Robustness to various memory configurations.

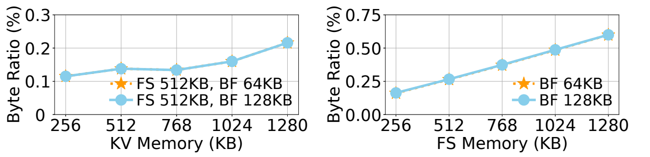
Name	PHV (Bytes)	VLIW	ALU	Stage
ElasticSketch	163 (21.22%)	13 (3.39%)	9 (18.75%)	10 (83.33%)
FlowRadar	134 (21.22%)	11 (2.86%)	15 (31.25%)	10 (83.33%)
SketchLearn	156 (20.31%)	11 (2.86%)	33 (68.75%)	8 (83.33%)
UnivMon	132 (17.19%)	13 (3.39%)	33 (68.75%)	12 (100%)
SeqSketch	151 (19.66%)	12 (3.12%)	13 (27.08%)	8 (66.67%)
EmbedSketch	137 (17.84%)	10 (2.60%)	6 (12.50%)	8 (66.67%)

Table 2: (Experiment 3) Switch resource usage.

ketch bounds the error below 0.1% for more than 99.72% and SeqSketch covers all flows. The reason is that existing algorithms depend on a large amount of memory to fully resolve hash conflicts. However, our algorithms recover flow values by solving an optimization problem based on compressive sensing, which is not so sensitive to hash conflicts.

Experiment 2: Robustness (Figure 12). We measure the ratio of flows with an error less than 0.1% in different configurations. To study the accuracy as memory changes, we fix two components and vary the size of the remaining one. For SeqSketch in Figure 12(a), when FS has 256KB, only 68% flows reach the accuracy level because the memory is far smaller than a reasonable size. However, the ratio increases to nearly 100% as the sketch size increases. Figure 12(b) shows that the accuracy remains stable for different BF configurations. Even with 32 KB (25% of the expected memory as §7.3), more than 96% flows remain per-flow error below 0.1%. In EmbedSketch, since each bucket of FS embeds one KV pair, we either fix BF and vary FS (Figure 12(c)) or vice versa (Figure 12(d)). We observe similar trends: the accuracy is low with 256 KB FS but grows as the size of FS, while remaining stable for various BF size.

We also find that SeqSketch is more memory efficient than EmbedSketch. For SeqSketch, 832 KB memory (512 KB FS, 64 KB BF, and 256 KB KV) is sufficient to achieve near-zero error. In contrast, EmbedSketch requires around 2.5 MB memory to reach the same level of accuracy, including at least 512 KB FS, 64 KB BF and 2048 KB KV. Here, the 2048 KB KV comes from the per-bucket key-value pairs. Recall that



(a) SeqSketch (b) EmbedSketch

Figure 13: (Experiment 4) Bandwidth usage.

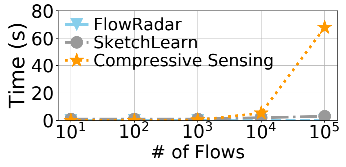


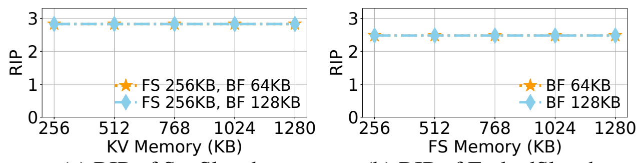
Figure 14: (Experiment 5) Recovery time.

each key-value pair occupies 16 bytes, which is 4× of a FS counter. Thus, KV consumes as 4× memory as FS. The root cause for different memory usage is that in SeqSketch, there are no duplicate flow IDs in the hash table, while a flow may be tracked multiple times in EmbedSketch.

Experiment 3: Resource usage in Tofino (Table 2). We compare SeqSketch and EmbedSketch with four state-of-the-art sketch algorithms. We consider four types of resources: stages, ALUs, and VLIW are used for updating sketch values, while PHV carries data across stages. We find that our algorithms consume fewer stages, ALUs, and VLIW than FR, SL, and UM. The reason is that the three algorithms need to update multiple instances (Table 1), while the components in our algorithms require only simple operations. SeqSketch incurs more resource usage than ES because it needs to update additional Bloom Filter and transfer flow records to the controller for evicted entries. EmbedSketch requires fewer resources than others because updating local structures is much simpler (e.g., fewer hash functions for BF).

Experiment 4: Bandwidth usage (Figure 13). We measure the ratio of incurred traffic to the traffic in a time interval. The incurred traffic compromises two parts: the evicted flow records and flow IDs during per-packet updates, and the transfer of the sketch at the end of each interval. We see that achieving NZE monitoring incurs less than 0.7% additional bandwidth consumption. Note that existing sketch algorithms only transfer the sketch structures. Although our algorithms additionally transfer flow IDs, the overall bandwidth usage remains limited for two reasons. First, the sketch structures are quite small. Second, we only send evicted flow records that aggregate a considerable number of packets to the controller. When an individual packet is evicted, it will be absorbed by FS. Further, BF avoids duplicate transfers of flow IDs.

Experiment 5: Recovery time (Figure14). We measure the recovery time for different number of flows. Currently, we use a single thread for recovery. The time is around 60 seconds in the worst case, which is much worse than sketch algorithms. We can optimize it by assigning recovery operations in different CPU cores. Recall that the time interval containing 100 K flows is around 2.5 seconds. Our 36-core server is sufficient



(a) RIP of SeqSketch (b) RIP of EmbedSketch

Figure 15: (Experiment 6) RIP.

to handle all recovery operations. Further, we can speed up with recent distributed machine learning architectures such as TensorFlow. Since solving optimization problem is not our focus, we leave it in the future work.

Experiment 6: RIP (Figure 15). We further examine the RIP of the two algorithms in Figure 15(a) and Figure 15(b), respectively. We observe that RIP remains below 3 in all cases, which is much smaller than that in existing algorithms (§6). The results reveal that SeqSketch and EmbedSketch produce highly orthonormal matrices, which lead to high accuracy when applying compressive sensing reconstruction.

More results. In Appendix, we also present the throughput in software (Experiment 7). We also present the complete accuracy results under different configurations.

7.5 Discussion

Correctness. The correctness of both SeqSketch and EmbedSketch can be derived from compressive sensing. Since we recover per-flow values with standard compressive sensing, the recovered results are close to the true values given the orthonormal matrices produced by the two algorithms (Experiment 3). We leave the formal proof in our future work.

Comparison to existing algorithms. Both the sequential design and embedding design have been used in prior algorithms. For example, ElasticSketch [80] evicts records from a hash table to a sketch; MV-Sketch [77] embeds heavy flows in buckets. Our algorithms are different in four aspects. First, our recovery is based on an optimization framework of compressive sensing. Second, we employ a fractional sketch that increments each counter by a fractional value. Third, we maintain a Bloom Filer to track all flow IDs. Finally, we leverage the controller to reduce the overheads in the data plane. With these design choices, our algorithms achieve near-zero errors.

8 Related Work

Measurement algorithms. Hash tables [1,2,52,59] achieve zero errors but incur excessive resource usage. Some approximate techniques reduce memory usage by addressing only heavy hitters [4,25,33,71]. Sampling techniques [14,41,69,70,74] selectively discard a portion of traffic to improve resource efficiency. Sketch algorithms [20,36,37,49,53,54,68,80,85] employ a compact structure in which multiple flows share a counter. These approximate algorithms usually provide theoretical guarantees to bound the incurred errors. However, the bounds are too loose to apply to all flows, leading to poor accuracy in practice (see §2).

Measurement systems. OpenSketch [82], SCREAM [58] and SketchVisor [35] enhance sketch algorithms in different aspects. Some systems boost performance with TCAM [40,57,60]. PacketHistory [32] and Planck [65] mirror traffic to the controller. EverFlow [87] and dShark [81] filter out uninterested traffic with pre-defined rules. mOS [38] and Confluo [42] address monitoring at edges. Studies on query languages [29,31,61,73,78] empower more fine-grained expressions to tune measurement tasks. TPP [39], MOZART [52], and SwitchPointer [76] combine software and hardware devices to provide both flexibility and programmability for network measurement. Different from these works, our work addresses the algorithmic design for flow monitoring. It is complementary to above system studies.

Compressive sensing for network measurement. CounterBraids [55] demonstrates that sketch and compressive sensing are thematically related, but does not actually apply compressive sensing. [48] applies Least Linear Square method to reconstruct flow values from CountMin Sketch, but does not consider other sketch techniques. [21] shows that sketch algorithms can be formulated as a special kind of compressive sensing. [16,44,83] leverage compressive sensing to restore missing values in traffic matrices. [7] uses compressive sensing for tomography. SketchVisor [35] merges its two paths with compressive sensing. In contrast, this paper leverages compressive sensing for NZE monitoring.

9 Conclusion

This paper revisits the theoretical bounds provided by sketch algorithms. We observe that the bounds in existing algorithms are too loose to achieve high accuracy for all flows. We address this problem with compressive sensing. We formulate sketch algorithms as matrices and study their suitability to compressive sensing. The results guide us to design two new algorithms accordingly. The efficiency of the two algorithms demonstrates the power of our methodology. We expect that more algorithms can be designed in the future.

Acknowledgements

We thank our shepherd, Jennifer Rexford, and the anonymous reviewers for their valuable comments. The work was supported in part by National Key R&D Program of China (2019YFB1802600), Joint Funds of the National Natural Science Foundation of China (U20A20179), National Natural Science Foundation of China (61802365), and "FANet: PCL Future Greater-Bay Area Network Facilities for Large-scale Experiments and Applications (No. LZC0019)".

References

[1] O. Alipourfard, M. Moshref, and M. Yu. Re-evaluating Measurement Algorithms in Software. In *Proc. of HotNets*, 2015.

[2] O. Alipourfard, M. Moshref, Y. Zhou, T. Yang, and M. Yu. A Comparison of Performance and Accuracy of Measurement Algorithms in Software. In *Proc. of ACM SOSR*, 2018.

[3] B. Arzani, S. Ciraci, L. Chamon, Y. Zhu, H. Liu, J. Padhye, B. T. Loo, and G. Outhred. 007: Democratically Finding The Cause of Packet Drops. In *Proc. of USENIX NSDI*, 2018.

[4] R. Ben Basat, G. Einziger, R. Friedman, M. C. Luizelli, and E. Waisbard. Constant Time Updates in Hierarchical Heavy Hitters. In *Proc. of ACM SIGCOMM*, 2017.

[5] T. Benson, A. Akella, and D. A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Proc. of ACM IMC*, 2010.

[6] P. Bosshart, G. Gibb, H.-s. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *Proc. of SIGCOMM*, 2013.

[7] Bowden, Rhys Alistair and Roughan, Matthew and Bean, Nigel. Network Link Tomography and Compressive Sensing. In *Proc. of SIGMETRICS*, 2011.

[8] T. Bu, J. Cao, A. Chen, and P. P. C. Lee. Sequential Hashing: A Flexible Approach for Unveiling Significant Patterns in High Speed Networks. *Computer Networks*, 54(18):3309–3326, 2010.

[9] Caida Anonymized Internet Traces 2018 Dataset. http://www.caida.org/data/passive/passive_dataset.xml.

[10] E. J. Candès et al. The Restricted Isometry Property and Its Implications for Compressed Sensing. *Comptes rendus mathematique*, 346(9-10):589–592, 2008.

[11] E. J. Candes, J. Romberg, and T. Tao. Robust Uncertainty Principles: Exact Signal Reconstruction from Highly Incomplete Frequency Information. *IEEE Transactions on Information Theory*, 52(2):489–509, 2006.

[12] E. J. Candes and T. Tao. Near-Optimal Signal Recovery From Random Projections: Universal Encoding Strategies? *IEEE Transactions on Information Theory*, 52(12):5406–5425, 2006.

[13] E. J. Candès and M. B. Wakin. An Introduction to Compressive Sampling. *IEEE Signal Processing Magazine*, 25(2):21–30, 2008.

[14] M. Canini, D. Fay, D. J. Miller, A. W. Moore, and R. Bolla. Per Flow Packet Sampling for High-Speed Network Monitoring. In *Proceedings of the First International Conference on Communication Systems and Networks (COMSNETS'09)*, 2009.

[15] M. Charikar, K. Chen, and M. Farach-Colton. Finding Frequent Items in Data Streams. *Theoretical Computer Science*, 312(1):3–15, 2004.

[16] Y.-C. Chen, L. Qiu, Y. Zhang, G. Xue, and Z. Hu. Robust Network Compressive Sensing. In *Proc. of MOBICOM*, 2014.

[17] Chen, Haoxian and Foster, Nate and Silverman, Jake and Whittaker, Michael and Zhang, Brandon and Zhang, Rene. Felix: Implementing Traffic Measurement on End Hosts Using Program Analysis. In *Proc. of SOSR*, 2016.

[18] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine. *Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches*. Now Publishers Inc., 2012.

[19] G. Cormode and S. Muthukrishnan. What's New: Finding Significant Differences in Network Data Streams. In *Proc. of IEEE INFOCOM*, 2004.

[20] G. Cormode and S. Muthukrishnan. An Improved Data Stream Summary: The Count-Min Sketch and its Applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[21] G. Cormode and S. Muthukrishnan. Towards an Algorithmic Theory of Compressed Sensing. Technical report, 2005.

[22] G. B. Dantzig, A. Orden, and P. Wolfe. The generalized simplex method for minimizing a linear form under linear inequality restraints. *Pacific Journal of Mathematics*, 5(2):183–195, 1955.

[23] Data Plane Development Kit. https://dpdk.org.

[24] D. L. Donoho. Compressed Sensing. *IEEE Transactions on Information Theory*, 52(4):1289–1306, 2006.

[25] C. Estan and G. Varghese. New Directions in Traffic Measurement and Accounting. In *Proc. of ACM SIGCOMM*, 2002.

[26] C. Estan, G. Varghese, and M. Fisk. Bitmap Algorithms for Counting Active Flows on High-Speed Links. In *Proc. of ACM SIGCOMM*, 2003.

[27] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[28] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. HyperLogLog: The Analysis of A Near-optimal Cardinality Estimation Algorithm. In *Proc. of AOFA*, pages 127–146, 2007.

[29] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A Network Programming Language. In *Proc. of ICFP*, 2011.

[30] Gong, Deli and Tran, Muoi and Shinde, Shweta and Jin, Hao and Sekar, Vyas and Saxena, Prateek and Kang, Min Suk. Practical Verifiable In-network Filtering for DDoS Defense. In *Proc. of IEEE ICDCS*, 2019.

[31] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger. Sonata: Query-Driven Streaming Network Telemetry. In *Proc. of ACM SIGCOMM*, 2018.

[32] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks. In *Proc. of USENIX NSDI*, 2014.

[33] R. Harrison, Q. Cai, A. Gupta, and J. Rexford. Network-Wide Heavy Hitter Detection with Commodity Switches. In *Proc. of ACM SOSR*, 2018.

[34] N. J. Harvey, J. Nelson, and K. Onak. Sketching and Streaming Entropy via Approximation Theory. In *Proc. of FOCS*, 2008.

[35] Q. Huang, X. Jin, P. P. C. Lee, R. Li, L. Tang, Y.-C. Chen, and G. Zhang. SketchVisor: Robust Network Measurement for Software Packet Processing. In *Proc. of ACM SIGCOMM*, 2017.

[36] Q. Huang and P. P. C. Lee. A Hybrid Local and Distributed Sketching Design for Accurate and Scalable Heavy Key Detection in Network Data Streams. *Computer Networks*, 91:298–315, 2015.

[37] Q. Huang, P. P. C. Lee, and Y. Bao. SketchLearn: Relieving User Burdens in Approximate Measurement with Automated Statistical Inference. In *Proc. of ACM SIGCOMM*, 2018.

[38] M. Jamshed, Y. Moon, D. Kim, D. Han, and K. Park. mOS: a reusable networking stack for flow monitoring middleboxes. In *Proc. of NSDI*, 2017.

[39] V. Jeyakumar, M. Alizadeh, Y. Geng, C. Kim, and D. Mazières. Millions of Little Minions: Using Packets for Low Latency Network Programming and Visibility. In *Proc. of ACM SIGCOMM*, 2014.

[40] L. Jose, M. Yu, and J. Rexford. Online Measurement of Large Traffic Aggregates on Commodity Switches. In *USENIX HotICE*, 2011.

[41] S. Kandula and R. Mahajan. Sampling Biases in Network Path Measurements andWhat To Do About It. In *Proc. of ACM IMC*, 2009.

[42] Khandelwal, Anurag and Agarwal, Rachit and Stoica, Ion. Confluo: Distributed Monitoring and Diagnosis Stack for High-Speed Networks. In *Proc. of USENIX NSDI*, 2019.

[43] D. Kim, Y. Zhu, C. Kim, J. Lee, and S. Seshan. Generic External Memory for Switch Data Planes. In *Proc. of ACM HotNets*, 2018.

[44] L. Kong, M. Xia, X. Liu, M. Wu, and X. Liu. Data Loss and Reconstruction in Sensor Networks. In *Proc. of IEEE INFOCOM*, 2013.

[45] B. Krishnamurthy, S. Sen, Y. Zhang, F. Park, and Y. Chen. Sketch-based Change Detection : Methods , Evaluation , and Applications. In *Proc. of ACM IMC*, 2003.

[46] A. Kumar, M. Sung, J. J. Xu, and J. Wang. Data Streaming Algorithms for Efficient and Accurate Estimation of Flow Size Distribution. In *Proc. of SIGMETRICS*, 2004.

[47] LASSO. https://en.wikipedia.org/wiki/Lasso_(statistics).

[48] G. M. Lee, H. Liu, Y. Yoon, and Y. Zhang. Improving Sketch Reconstruction Accuracy Using Linear Least Squares Method. In *Proc. of IMC*, 2005.

[49] Y. Li, R. Miao, C. Kim, and M. Yu. FlowRadar: A Better NetFlow for Data Centers. In *Proc. of USENIX NSDI*, 2016.

[50] Y. Li, R. Miao, C. Kim, and M. Yu. LossRadar: Fast Detection of Lost Packets in Data Center Networks. In *Proc. of ACM CoNEXT*, 2016.

[51] Li, Yuliang and Miao, Rui and Alizadeh, Mohammad and Yu, Minlan. DETER: Deterministic TCP Replay for Performance Diagnosis. In *Proc. of USENIX NSDI*, 2019.

[52] X. Liu, M. Shirazipour, M. Yu, and Y. Zhang. MOZART: Temporal Coordination of Measurement. In *Proc. of ACM SOSR*, 2016.

[53] Z. Liu, R. Ben-Basat, G. Einziger, Y. Kassner, V. Braverman, R. Friedman, and V. Sekar. Nitrosketch: Robust and General Sketch-Based Monitoring in Software Switches. In *ACM SIGCOMM*, 2019.

[54] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *Proc. of ACM SIGCOMM*, 2016.

[55] Y. Lu, A. Montanari, B. Prabhakar, S. Dharmapurikar, and A. Kabbani. Counter Braids: A Novel Counter Architecture for Per-Flow Measurement. In *Proc. of SIGMETRICS*, 2008.

[56] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proc. of ACM SIGCOMM*, 2017.

[57] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. DREAM: Dynamic Resource Allocation for Software-defined Measurement. In *Proc. of ACM SIGCOMM*, 2014.

[58] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. SCREAM: Sketch Resource Allocation for Software-defined Measurement. In *Proc. of ACM CoNEXT*, 2015.

[59] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. Trumpet: Timely and Precise Triggers in Data Centers. In *Proc. of ACM SIGCOMM*, 2016.

[60] S. Narayana, M. T. Arashloo, J. Rexford, and D. Walker. Compiling Path Queries. In *Proc. of USENIX NSDI*, 2016.

[61] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim. Language-Directed Hardware Design for Network Performance Monitoring. In *Proc. of ACM SIGCOMM*, 2017.

[62] OpenvSwitch. http://openvswitch.org.

[63] P4 Language. https://p4.org.

[64] Y. C. Pati, R. Rezaiifar, and P. S. Krishnaprasad. Orthogonal Matching Pursuit: Recursive Function Approximation with Applications to Wavelet Decomposition. pages 40–44, 1993.

[65] J. Rasley, B. Stephens, C. Dixon, E. Rozner, W. Felter, K. Agarwal, J. Carter, and R. Fonseca. Planck: Millisecond-scale Monitoring and Control for Commodity Networks. In *Proc. of ACM SIGCOMM*, 2014.

[66] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the Social Network' s (Datacenter) Network. *Proc. of ACM SIGCOMM*, 2015.

[67] B. Schlinker, I. Cunha, Y.-C. Chiu, S. Sundaresan, and E. Katz-Bassett. Internet Performance from Facebook's Edge. In *Proc. of IMC*, 2019.

[68] R. Schweller, Z. Li, Y. Chen, Y. Gao, A. Gupta, Y. Zhang, P. Dinda, M. Y. Kao, and G. Memik. Reversible Sketches: Enabling Monitoring and Analysis over High-Speed Data Streams. *IEEE/ACM Trans. on Networking*, 15(5):1059–1072, 2007.

[69] V. Sekar, M. K. Reiter, W. Willinger, H. Zhang, R. R. Kompella, and D. G. Andersen. cSAMP: A System for Network-Wide Flow Monitoring. In *Proc. of USENIX NSDI*, 2008.

[70] V. Sekar, M. K. Reiter, and H. Zhang. Revisiting the Case for a Minimalist Approach for Network Flow Monitoring. In *Proc. of ACM IMC*, 2010.

[71] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford. Heavy-Hitter Detection Entirely in the Data Plane. In *Proc. of ACM SOSR*, 2017.

[72] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood. Fast Hash Table Lookup using Extended Bloom Filter. In *Proc. of ACM SIGCOMM*, 2005.

[73] H. H. Song, L. Qiu, and Y. Zhang. NetQuest: A Flexible Framework for Large-scale Network Measurement. *IEEE/ACM Trans. on Networking*, 17(1):106–119, 2009.

[74] D. Stutzbach, R. Rejaie, N. Duffield, S. Sen, and W. Willinger. On Unbiased Sampling for Unstructured Peer-to-peer Networks. *IEEE/ACM Trans. on Networking*, 17(2):377–390, 2009.

[75] S. J. Szarek. Condition Numbers of Random Matrices. *Journal of Complexity*, 7(2):131–149, 1991.

[76] P. Tammana, R. Agarwal, and M. Lee. Distributed Network Monitoring and Debugging with SwitchPointer. In *Proc. of USENIX NSDI*, 2018.

[77] L. Tang, Q. Huang, and P. P. C. Lee. MV-Sketch: A Fast and Compact Invertible Sketch for Heavy Flow Detection in Network Data Streams. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, 2019.

[78] O. Tilmans, T. Bühler, I. Poese, S. Vissicchio, and L. Vanbever. Stroboscope: Declarative Traffic Mirroring on a Budget. In *Proc. of USENIX NSDI*, 2018.

[79] Tofino. `https://www.barefootnetworks.com/products/brief-tofino/`.

[80] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig. Elastic Sketch: Adaptive and Fast Network-wide Measurements. In *Proc. of ACM SIGCOMM*, 2018.

[81] D. Yu, Y. Zhu, B. Arzani, R. Fonseca, T. Zhang, K. Deng, and L. Yuan. dShark: A General, Easy to Program and Scalable Framework for Analyzing In-network Packet Traces. In *Proc. of USENIX NSDI*, 2019.

[82] M. Yu, L. Jose, and R. Miao. Software Defined Traffic Measurement with OpenSketch. In *Proc. of USENIX NSDI*, 2013.

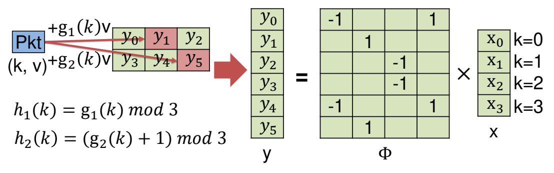
[83] Y. Zhang, M. Roughan, W. Willinger, and L. Qiu. Spatio-Temporal Compressive Sensing and Internet Traffic Matrices. In *Proc. of ACM SIGCOMM*, 2009.

[84] Q. Zhao, A. Kumar, and J. Xu. Joint Data Streaming and Sampling Techniques for Detection of Super Sources and Destinations. In *Proc. of IMC*, 2005.

[85] Y. Zhao, K. Yang, Z. Liu, T. Yang, L. Chen, S. Liu, N. Zheng, R. Wang, H. Wu, Y. Wang, and N. Zhang. LightGuardian: A Full-Visibility, Lightweight, In-band Telemetry System Using Sketchlet. In *Proc. of NSDI*, 2021.

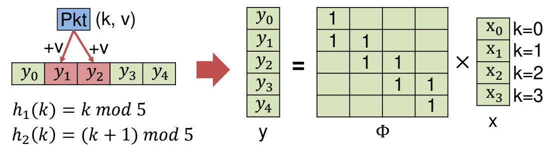
[86] Zheng, Shengbao and Yang, Xiaowei. Dynashield: Reducing the Cost of DDoS Defense Using Cloud Services. In *Proc. of HotCloud*, 2019.

[87] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and H. Zheng. Packet-Level Telemetry in Large Datacenter Networks. In *Proc. of ACM SIGCOMM*, 2015.



$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{pmatrix} = \begin{pmatrix} -1 & & & 1 \\ & 1 & & \\ & & -1 & \\ & & -1 & \\ -1 & & & 1 \\ & 1 & & \end{pmatrix} \times \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

(a) CountSketch



$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix} = \begin{pmatrix} 1 & & & \\ 1 & 1 & & \\ & 1 & 1 & \\ & & 1 & 1 \\ & & & 1 \end{pmatrix} \times \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

(b) Count Bloom Filter

Figure 16: Matrix formulation of CS and CBF.

Appendix A: Sketch-based Sensing

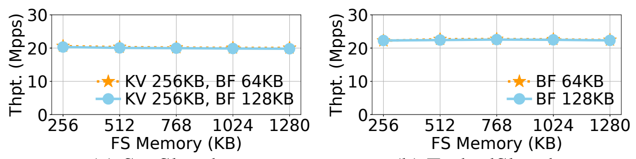
Examples. Figure 16 presents another two examples of the sensing matrices for CS and CBF. As §4.3, we also consider four flows. Figure 16(a) shows a CountSketch (CS). CS has the same structure as CM. However, a packet (k, v) increments its hashed counter in row t by $g_t(k) \cdot v$, where $g_t(k)$ is another function that maps a flow ID to $\{-1, 1\}$. Thus, $\Phi_{i,k}$ is either 1 or -1 if flow k hits counter i. Figure 16(b) presents a Bloom Filter. Note that the original Bloom Filter is not a linear mapping, because it maintains an array of bits and performs bitwise OR operations. We extend it to a Counting Bloom Filter (CBF) that replaces the bit array with a counter array. Each counter is updated by v for a packet (k, v) hashed to it. Thus, we set $\Phi_{i,k} = 1$ if flow k hashes to counter i.

Appendix B: Implementation

Software version. The software version integrates OpenVSwitch (OVS) [62]. We target two implementations of OVS: one resides in the kernel space, and the other bypasses the kernel via DPDK [23]. In each implementation, we intercept packets in the forwarding module. We put the packet headers in a region of shared memory. A dedicated thread reads the shared memory and updates the sketch (either SeqSketch or EmbedSketch) accordingly.

Hardware version. We implement the hardware version in P4 [63] and target PISA [6] switches. We place the data structures in switch registers, and invoke stateful ALUs to update register values for each packet. However, the limited memory access model of PISA raises two challenges. The first challenge is that each memory access can only manipulate at most 64-bit variables, but in our algorithms, we need to update more than 64 bits of data each time. Second, PISA partitions hardware resources into several stages, each of which is associated with its own ALUs and registers. An ALU can only access the registers belonging to its same stage.

To this end, we tailor SeqSketch and EmbedSketch to fit them into PISA switches. For the first challenge, we sepa-



(a) SeqSketch. (b) EmbedSketch.

Figure 17: Throughput in software.

Table 3: Compressive sensing results with different memory.

Matrix	Recovery	Memory (KB)	(<1e-1)	(<5e-2)	(<1e-2)	(<1e-3)
BM	L1	100	2.01%	1.67%	1.52%	1.52%
BM	L1	200	100%	100%	100%	100%
BM	OMP	100	1.94%	1.62%	1.47%	1.46%
BM	OMP	200	100%	100%	100%	100%
FM	L1	100	26.25%	26.25%	26.25%	26.25%
FM	L1	200	52.50%	52.50%	52.50%	52.50%
FM	L1	300	78.76%	78.76%	78.76%	78.76%
FM	L1	400	100%	100%	100%	100%
FM	OMP	100	14.81%	14.23%	14.06%	14.04%
FM	OMP	200	100%	100%	100%	100%
GM	L1	100	1.94%	1.59%	1.44%	1.44%
GM	L1	200	100%	100%	100%	100%
GM	OMP	100	1.90%	1.60%	1.46%	1.46%
GM	OMP	200	100%	100%	100%	100%
IM	L1	100	1.98%	1.62%	1.48%	1.48%
IM	L1	200	100%	100%	100%	100%
IM	OMP	100	1.98%	1.62%	1.48%	1.48%
IM	OMP	200	100%	100%	100%	100%

rate different types of variables across stages, such that the size of accessed variables in each stage does not exceed the 64-bit limit. For the second challenge, we introduce a few intermediate variables to break the inter-dependencies among variables. More precisely, we store only f and c in the same stage, but replace the variable d with a new variable d'. The new variable d' resides in a stage before f and c. It counts all incoming flows (i.e. $d' = c + d$) and records its value in a metadata field such that it can be shared across stages. The later stage (i.e., that actually maintains f and c) reads d' from the metadata, and determines to perform an eviction operation based on whether $d' - c > c$.

Appendix C: more experiments

Experiment 7: Throughput in software switches (Figure17). We measure the throughput of the two algorithms. We observe that both algorithms keep stable throughput. The throughput of EmbedSketch is higher than that of SeqSketch because its local structures are simpler (see Experiment 3).

Complete results. Table 3 provides the results of classical compressive sensing under different memory settings. Table 4 shows the theoretical configurations of state-of-the-art algorithms with 1% threshold, 1% relative error, and 5% error probability. Table 5, Table 6 Table 7, and Table 8 show the complete results of SeqSketch and EmbedSketch.

Table 4: Theoretical configurations of exiting algorithms.

Algorithm	CU Sketch	Deltoid	ElasticSketch	FlowRadar	NitroSketch
Memory (KB)	312	32500	4438	2115	32672
Algorithm	RevSketch	SeqHash	SketchLearn	SketchVisor	UnivMon
Memory (KB)	58594	32500	32500	2123	32672

Table 5: SeqSketch under different epoch lengths.

Epoch Length (s)	1s	2s	5s	10s	25s
Total Memory (KB)	672	1344	2016	3360	6720
KV Memory (KB)	128	256	384	640	1280
BF Memory (KB)	32	64	96	160	320
FS Memory (KB)	512	1024	1536	2560	5120
(<1e-1)	98.80%	99.51%	98.55%	98.13%	99.95%
(<5e-2)	98.79%	99.50%	98.51%	98.07%	99.95%
(<1e-2)	98.78%	99.50%	98.48%	98.04%	99.95%
(<1e-3)	98.77%	99.49%	98.48%	98.03%	99.95%
Precision (%)	100	100	100	100	100
Recall (%)	99	99	99	99	99
Bandwidth Overhead	0.33%	0.29%	0.20%	0.17%	0.13%

Table 6: EmbedSketch under different epoch lengths.

Epoch Length (s)	1s	2s	5s	10s	25s
Total Memory (KB)	2592	5184	7776	12960	25920
BF Memory (KB)	32	64	96	160	320
FS Memory (KB)	2560	5120	7680	12800	25600
(<1e-1)	98.62%	99.34%	98.46%	98.39%	98.31%
(<5e-2)	98.55%	99.30%	98.38%	98.30%	98.19%
(<1e-2)	98.51%	99.26%	98.32%	98.23%	98.13%
(<1e-3)	98.50%	99.25%	98.31%	98.22%	98.13%
Precision (%)	100	100	100	100	100
Recall (%)	99	99	99	99	99
Bandwidth Overhead	0.81%	0.79%	0.50%	0.41%	0.33%

Table 7: SeqSketch configurations.

Total Memory (KB)	KV Memory (KB)	BF Memory (KB)	FS Memory (KB)	(<1e-1)	(<5e-2)	(<1e-2)	(<1e-3)	Precision (%)	Recall (%)	Bandwidth Overhead
544	256	32	256	37.87%	37.28%	36.81%	36.70%	100	71	0.0854%
800	256	32	512	81.46%	80.80%	80.20%	80.11%	100	92	0.115%
576	256	64	256	43.08%	42.15%	41.39%	41.24%	100	77	0.0854%
832	256	64	512	96.78%	96.66%	96.55%	96.53%	100	99	0.115%
1088	256	64	768	98.28%	98.21%	98.17%	98.16%	100	99	0.144%
1344	256	64	1024	98.89%	98.86%	98.84%	98.84%	100	99	0.173%
1600	256	64	1280	99.04%	99.01%	98.99%	98.99%	100	99	0.202%
640	256	128	256	59.61%	58.19%	57.06%	56.87%	100	87	0.0854%
896	256	128	512	99.63%	99.61%	99.60%	99.60%	100	100	0.115%
1152	256	128	768	99.83%	99.82%	99.82%	99.82%	100	100	0.144%
1408	256	128	1024	99.90%	99.90%	99.90%	99.90%	100	100	0.173%
1664	256	128	1280	99.90%	99.90%	99.90%	99.90%	100	100	0.202%
768	256	256	256	60.12%	58.72%	57.59%	57.41%	100	87	0.0854%
1024	256	256	512	99.97%	99.97%	99.97%	99.97%	100	100	0.115%
1280	256	256	768	99.98%	99.98%	99.98%	99.98%	100	100	0.144%
1536	256	256	1024	100%	100%	100%	100%	100	100	0.173%
1024	256	512	256	60.15%	58.75%	57.62%	57.44%	100	87	0.0854%
1280	256	512	512	100%	100%	100%	100%	100	100	0.115%
1280	256	768	256	60.15%	58.75%	57.62%	57.44%	100	87	0.0854%
1536	256	768	512	100%	100%	100%	100%	100	100	0.115%
1536	256	1024	256	60.15%	58.75%	57.62%	57.44%	100	87	0.0854%
1792	256	1024	512	100%	100%	100%	100%	100	100	0.115%
800	512	32	256	54.65%	53.99%	53.44%	53.34%	100	81	0.109%
1056	512	32	512	93.04%	92.84%	92.68%	92.66%	100	97	0.138%
1312	512	32	768	95.01%	94.87%	94.76%	94.74%	100	98	0.167%
832	512	64	256	67.98%	67.25%	66.69%	66.60%	100	88	0.109%
1088	512	64	512	98.92%	98.89%	98.87%	98.86%	100	100	0.138%
1344	512	64	768	99.38%	99.37%	99.35%	99.35%	100	100	0.167%
896	512	128	256	80.63%	79.98%	79.48%	79.39%	100	94	0.109%
1152	512	128	512	99.88%	99.88%	99.88%	99.88%	100	100	0.138%
1408	512	128	768	99.94%	99.94%	99.94%	99.94%	100	100	0.167%
1024	512	256	256	88.02%	87.58%	87.24%	87.19%	100	96	0.109%
1280	512	256	512	99.98%	99.98%	99.98%	99.98%	100	100	0.138%
1536	512	256	768	99.99%	99.99%	99.99%	99.99%	100	100	0.167%
1792	512	256	1024	100%	100%	100%	100%	100	100	0.196%
1280	512	512	256	100%	100%	100%	100%	100	100	0.109%
1536	512	768	256	100%	100%	100%	100%	100	100	0.109%
1792	512	1024	256	100%	100%	100%	100%	100	100	0.109%
1280	768	256	256	99.91%	99.91%	99.91%	99.91%	100	100	0.105%
1536	768	256	512	100%	100%	100%	100%	100	100	0.134%
1536	768	512	256	100%	100%	100%	100%	100	100	0.105%
1792	768	768	256	100%	100%	100%	100%	100	100	0.105%
2048	768	1024	256	100%	100%	100%	100%	100	100	0.105%
1536	1024	256	256	99.97%	99.97%	99.97%	99.97%	100	100	0.16%
1792	1024	256	512	99.99%	99.99%	99.99%	99.99%	100	100	0.189%
2048	1024	256	768	99.99%	99.99%	99.99%	99.99%	100	100	0.218%
2304	1024	256	1024	100%	100%	100%	100%	100	100	0.247
1792	1024	512	256	100%	100%	100%	100%	100	100	0.16%
2048	1024	768	256	100%	100%	100%	100%	100	100	0.16%
2304	1024	1024	256	100%	100%	100%	100%	100	100	0.16%
2048	1280	256	512	100%	100%	100%	100%	100	100	0.216%
2304	1280	512	512	100%	100%	100%	100%	100	100	0.216%

Table 8: EmbedSketch configurations.

Total Memory (KB)	BF Memory (KB)	FS Memory (KB)	(<1e-1)	(<5e-2)	(<1e-2)	(<1e-3)	Precision (%)	Recall (%)	Bandwidth Overhead
1568	32	1536	72.26%	71.70%	71.21%	71.13%	100	87	0.205%
2080	32	2048	85.62%	85.12%	84.73%	84.67%	100	94	0.259%
2592	32	2560	91.87%	91.63%	91.40%	91.35%	100	97	0.315%
4128	32	4096	97.81%	97.72%	97.64%	97.63%	100	99	0.484%
1600	64	1536	82.09%	81.34%	80.73%	80.62%	100	94	0.21%
2112	64	2048	90.61%	90.37%	90.13%	90.09%	100	96	0.264%
2624	64	2560	95.76%	95.62%	95.51%	95.49%	100	98	0.318%
4160	64	4096	98.14%	98.07%	98.00%	97.99%	100	99	0.485%
1664	128	1536	88.48%	87.80%	87.17%	87.06%	100	98	0.214%
2176	128	2048	94.95%	94.75%	94.53%	94.50%	100	98	0.266%
2688	128	2560	98.51%	98.47%	98.43%	98.41%	100	99	0.32%
4224	128	4096	99.82%	99.81%	99.81%	99.80%	100	00	0.487%
1792	256	1536	89.02%	88.35%	87.70%	87.59%	100	98	0.214%
2048	256	1792	93.83%	93.45%	93.06%	93.00%	100	99	0.24%
2304	256	2048	97.47%	97.31%	97.12%	97.10%	100	100	0.267%
2560	256	2304	99.16%	99.09%	99.03%	99.02%	100	100	0.294%
2816	256	2560	99.75%	99.74%	99.72%	99.72%	100	100	0.321%
3072	256	2816	99.97%	99.97%	99.97%	99.97%	100	100	0.348%
4352	256	4096	100%	100%	100%	100%	100	100	0.487%
2048	512	1536	88.92%	88.24%	87.59%	87.48%	100	98	0.214%
2304	512	1792	93.78%	93.40%	93.02%	92.95%	100	100	0.240%
2560	512	2048	97.54%	97.38%	97.19%	97.17%	100	100	0.267%
2816	512	2304	99.10%	99.04%	98.97%	98.96%	100	100	0.294%
3072	512	2560	99.84%	99.83%	99.82%	99.82%	100	100	0.321%
3328	512	2816	100%	99.99%	99.99%	99.99%	100	100	0.348%
4068	512	4096	100%	100%	100%	100%	100	100	0.488%
2304	768	1536	88.92%	88.24%	87.59%	87.48%	100	98	0.213%
2560	768	1792	93.78%	93.39%	93.01%	92.95%	100	99	0.24%
2816	768	2048	97.54%	97.38%	97.19%	97.17%	100	100	0.267%
3072	768	2304	99.10%	99.03%	98.96%	98.96%	100	100	0.294%
3328	768	2560	99.84%	99.83%	99.82%	99.82%	100	100	0.321%
3584	768	2816	100%	100%	100%	100%	100	100	0.348%
2560	1024	1536	88.91%	88.24%	87.58%	87.47%	100	98	0.214%
2816	1024	1792	93.78%	93.40%	93.02%	92.95%	100	99	0.24%
3072	1024	2048	97.53%	97.37%	97.18%	97.16%	100	100	0.267%
3328	1024	2304	99.10%	99.03%	98.96%	98.96%	100	100	0.294%
3584	1024	2560	99.84%	99.83%	99.82%	99.82%	100	100	0.321%
3840	1024	2816	100%	100%	100%	100%	100	100	0.348%
2816	1280	1536	88.92%	88.24%	87.58%	87.47%	100	98	0.214%
3072	1280	1792	93.78%	93.40%	93.02%	92.95%	100	100	0.24%
3328	1280	2048	97.53%	97.37%	97.18%	97.16%	100	100	0.267%
3584	1280	2304	99.10%	99.03%	98.96%	98.96%	100	100	0.294%
3840	1280	2560	99.84%	99.83%	99.82%	99.82%	100	100	0.321%
4096	1280	2816	100%	100%	100%	100%	100	100	0.348%
3072	1536	1536	88.92%	88.24%	87.58%	87.47%	100	98	0.214%
3328	1536	1792	93.78%	93.40%	93.02%	92.95%	100	100	0.24%
3584	1536	2048	97.53%	97.37%	97.18%	97.16%	100	100	0.267%
3840	1536	2304	99.10%	99.03%	98.96%	98.96%	100	100	0.294%
4096	1536	2560	99.84%	99.83%	99.82%	99.82%	100	100	0.321%
4352	1536	2816	100%	100%	100%	100%	100	100	0.348%