TOWARD ZERO-WASTE TERABIT NETWORKED SYSTEMS

Liangcheng Yu

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2024

Dedicated to my significant other and my parents.

ACKNOWLEDGEMENT

First and foremost, I would like to deeply thank my advisor, Vincent Liu, for the enduring support over the past five years. I vividly recall our first conversion where Vincent said: 'The essence of a doctoral journey transcends mere publication counts or lines of code; it lies in the cultivation of independent thinking'. Since then, every interaction has been profoundly enlightening and influential to my growth, nurturing not only my first-principles reasoning and research taste, but also fueling my internal interest and passion into the field of systems and networking.

It is my privilege to have Boon Thau Loo, Benjamin C. Lee, Jonathan M. Smith, and Arvind Krishnamurthy on my dissertation committee. I am immensely grateful to Boon for his continuous support and for curating a incredible group of individuals in the Distributed Systems Lab, of which I am honored to be a part. I express my gratitude to Ben for his insightful feedback on my research as well as his encouragement and support in exploring the realm of sustainable computing. I will never forget the time spent with JMS, particularly in the DSL machine room over the years, where he, as a role model, generously shared practical tips, toolkits, and inspirational mottos about system administration that continue to resonate with me. I am also fortunate to have crossed paths with my grand advisor, Arvind, since my initial internship at Google in 2021, and I am grateful for his guidance and mentorship throughout my academic journey.

I want to thank all my coauthors including John Sonchack, Yinda Zhang, Gianni Antichi, Ran Ben Basat, Prateesh Goyal, Ilias Marinos, Xinyi Chen, Qizhen Zhang, Nofel Yassen, Caleb Stanford, Ryan Beckett, Xiao Zhang, Haoran Zhang, Dan Ports, Yiran Lei, Mingwei Xu, Abhiram Ravi, Weida Huang, Tyler Griggs, KK Yap, Milad Hashemi, Neal Cardwell, Arvind Krishnamurthy, Nandita Dukkipati, Weitao Wang, and T. S. Eugene Ng. I am blessed to have worked with them. Their valuable support is indispensible to the completion of this thesis.

In addition, I extend my heartfelt thanks to Daehyeok Kim, Behnaz Arzani, Wei Bai, Konstantinos Kallas, Dongjoon Park, Andrew Head, Andreas Haeberlen, JK Lee, Yanfang Le, Vladimir Gurevich, Fabian Ruffy, Hugo Sadok, Alex Forencich, Alex Snooren, Amy Ousterhout, Sylvia Ratnasamy, and my internship mentor at Microsoft Derek Chiou for the discussion and inspiring feedback on my research. I am also grateful to people at the Office of the CTO team in Azure for Operators Research and Microsoft Research Redmond for their invaluable advice and support as I transition into my career post-graduation there. I greatly cherish the opportunity to expand the vision of the thesis into broader areas beyond core data center networking, where numerous intriguing problems with practical impact await exploration.

I am also grateful for the generous support provided by the Penn Presidential Ph.D. Fellowship along with its equipment fund. This support has empowered my research with access to emerging hardware accelerators and state-of-the-art servers. I wish to give back and contribute to the community in the future.

I am deeply indebted to my significant other, Mio, for her unwavering love during challenging times and for our journey ahead. Despite our different personalities, she completes me in ways that profoundly change my life—I cannot imagine it without her. I also thank our two kids (cats), Vinty and Wuggi, for their companionship during our moves between Philadelphia, San Diego, and Redmond. In addition, I am grateful to my parents for their dedication and support throughout every stage of my life. Words will never suffice to express my gratitude to them.

Finally, to the reader of this thesis, thank you for dedicating your time to read any part of it. If you have any questions, I would love to discuss them with you.

ABSTRACT

TOWARD ZERO-WASTE TERABIT NETWORKED SYSTEMS

Liangcheng Yu

Vincent Liu

To support modern applications, computer networks perform a plethora of auxiliary functions beyond basic application data forwarding. Obvious examples include serialization and encryption, triggered on most data transfers, but also control and monitoring that analyze and update network states.

Unfortunately, the unprecedented increase in application demand and a concurrent slowdown in the scaling of compute capability make it increasingly challenging to maintain these functions performantly and cost-effectively. On the one hand, continued exponential increases in network link speeds have led to the majority of congestion events occurring at microsecond time scales, diminishing the effectiveness of current control and monitoring protocols. Conversely, adding supporting resources (e.g., bandwidth, processing cores, and power budget) incurs expensive costs at scale, entailing not only capital and operating expenditures but also carbon footprint.

In this dissertation, we characterize and explore a zero-waste design approach by unlocking the potential of widespread in-network waste and present three case studies for auxiliary functions spanning across data, control, and management planes: (a) OrbWeaver, a weaved stream abstraction that reuses IDLE cycles in Ethernet links at 100s of ns granularity for state-of-the-art in-band control protocols; (b) Mantis, a switch-local reaction framework that recycles switch-local resources and co-designs them with the programmable data planes for user-defined and fine-grained (at 10s of µs granularity) closed-loop control functions; and (c) Beaver, an optimistic gateway marking primitive that reduces the waste of additional servers and instrumentation cost to enable partial snapshots 'in-situ' for diagnosing distributed cloud services with near-zero impact to existing service traffic. We also show that it is possible to integrate these functions performantly at near-zero cost.

The dissertation concludes with a vision for zero-waste networked systems, where we instantiate zero-waste designs to maximize the utility of residual network capacity despite existing efforts toward high-efficiency designs. More broadly, we posit that a grand challenge of our computing infrastructure is pushing waste to its limits amidst technology scaling slowdowns and increasing environmental concerns. This dissertation invites us to rethink the design patterns for networked system and outlines a spectrum of opportunities to advance this goal.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF ILLUSTRATIONS

CHAPTER 1

INTRODUCTION

Entities should not be multiplied beyond necessity.

William of Ockham

1.1. Motivation

1.1.1. Networks, Beyond Application Data Forwarding

Modern networked systems have directly enabled applications that transform billions of people's lives, from the way we travel, trade, and entertain to how we communicate and find information. The massive growth of these applications has, in turn, led to an increase in network speed, massive scaling of deployment sizes, growth in the diversity of workloads and network policies, and rising heterogeneity of hardware and software.

To sustain these trends and continue serving tomorrow's users, modern networks—from the perspective of network operators—must perform not only massive computations and communications for applications, but also a spectrum of *auxiliary functions*. Examples include additional processing associated with each service request (such as for serialization and encryption), but also a wide spectrum of control tasks to monitor, analyze, and update network states [35, 118, 147, 113, 101, 123, 124], as shown in Table 1.1. These functions are instrumental in meeting the ever-increasing performance demands of applications.

Increasing networking bandwidth, a double-edged sword. The past decades have witnessed the continuous evolution of the Ethernet link bandwidth, advancing from 10 Mbps in the early 1980s to the 400 Gbps commonplace today (Figure 1.1). 800 GbE is on the horizon—with the recent approval of the IEEE P802.3df Task Force standard, alongside new technologies like next-generation switching chips, higher modulation schemes, and energy-efficient optical modules—and the timeline for 1.6 TbE standardization is anticipated by 2026 [28, 25]. While this unprecedented growth in bandwidth

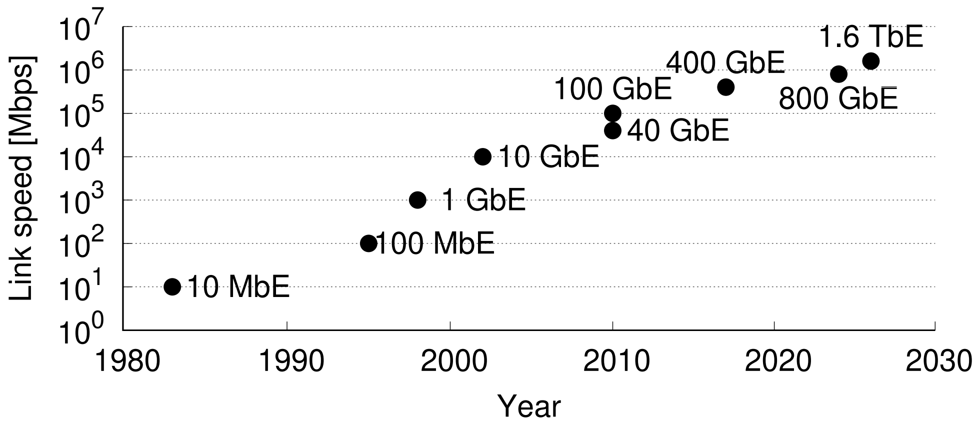


Figure 1.1: Ethernet link bandwidth (per the corresponding Ethernet standard) has increased by five orders of magnitude during the past decades and is approaching Terabit speed to sustain the application demand.

has brought significant benefits to users, such as faster transfer of application data, it has also made it increasingly challenging for network operators to support auxiliary functions such as monitoring and control.

One direct implication of faster bandwidth is that the timescale of network events is getting smaller, which poses significant challenges to current control and monitoring protocols. Consider the timescale of a single packet transmission. While networking bandwidth has increased by several orders of magnitude, packet sizes have remained relatively static since the inception of the Ethernet standard. For example, with 400 GbE, the transmission delay for a 1500B packet decreases to just 30 ns. Recent studies in major production networks have also shown that the majority of congestion events occur on microsecond timescales [204, 193, 72]. As a result, terabit networks push the necessary granularity to more microscopic scales, causing a growing mismatch with today's control and monitoring protocols, which are increasingly coarse-grained and ineffective in reacting to network behaviors. The severity of this mismatch is increasing as the bandwidth continues to grow. Moreover, the network is encompassing a more heterogeneous set of devices, and emerging user-facing and high-performance computing applications are demanding ever-increasing latency and throughput requirements, raising the bar for performance-critical control functions even further.

Adding to the challenge is that while the bandwidth has grown dramatically, many other elements

Tax functions	**Overhead considerations**
Layer-4 load balancing	*"...in our cloud, less than 1% of the total server cost would be considered low cost, so any solution that would cost more than 400 general-purpose servers is too expensive."* [147]
Layer-3 load balancing	*"The ToRs in the network need to send HULA probes frequently enough so that the network receives fine-grained information about global congestion state. However, the frequency should be low enough so that the network is not overwhelmed by probe traffic alone."* [105]
In-band telemetry	*"In the presence of 48 bytes overhead, which corresponds to 3.2% of a 1500B packet, the average FCT increases by 10%, while the goodput for long flows degrades by 10% if network utilization is approximately 70%."* [47]
Failure diagnosis using packet trace analysis	*"A similar same scalability challenge applies to trace analysis. Because commodity switches have limited storage and CPU power, traced packets must be sent to servers for analysis...still need 3200 servers for analysis which is prohibitively expensive."* [208]
Clock synchronization	*"However, it does not handle synchronization between network devices (in the data-plane) and incurs a non-negligible amount of bandwidth and processing overhead."* [103]
RPCs, protobufs, and so on	*"...'tax cycles' consistently comprise 22-27% of all execution... We have observed services that spend virtually all their time paying tax, and would benefit disproportionately from reducing it."* [101]
Virtual switch firewall	*"Each feature we add to the Fast Path has a cost and consumes per-packet CPU budget. Only performance-critical low-latency work belongs on the Fast Path."* [58]
Service mesh sidecars	*"...microservices spend much more time sending and processing network requests over RPCs or other REST APIs...this time increases to 36.3% of total execution time, causing the system's resource bottlenecks to change drastically."* [68]

Table 1.1: Example auxiliary functions and associated cost considerations.

have not kept up. Consider the latency between various components of the network, such as the latency between the control and data planes and the end-to-end latency between end hosts. These latencies are becoming less effective compared to the increase in bandwidth, approaching their lower bounds due to fundamental limits in hardware components (e.g., Forward Error Correction (FEC)) and physical distances [160, 25]. This makes timely feedback more difficult for closed-loop control functions, such as failure diagnosis, where it becomes more challenging to correlate information about misbehaviors in connectivity, storage, configurations, and power supply and to locate the exact culprits and victims in time. Such limitations are fundamental and extend far beyond the latency factor, with the concurrent slowdown in the scaling of the compute capability exacerbating the issue.

Performance-cost trade-off in supporting auxiliary functions. Given the challenges described, what options are available to network operators? Maintaining the current status is not viable, as the onus is on the network operator to keep up with the increasingly stringent performance requirements and application demand. Instead, operators might consider adding the necessary supporting resources (e.g., bandwidth, processing cores, and power budget) to keep pace with the successive generations of link speeds and increasing demand. Unfortunately, this approach incurs significant costs at scale, encompassing both capital and operating expenditures, as well as an increased carbon footprint, which includes both embodied carbon (the cost incurred during semiconductor manufacturing and amortized throughout the hardware's lifetime) and operational carbon (the day-to-day footprint of the system during energy consumption) [81, 149, 79, 116]. Such cost concerns (Table 1.1) are becoming more pronounced as technological advancements in DRAM $/byte, disk $/byte, and power efficiency plateau, alongside growing environmental concerns [38, 149, 39].

At the heart of this dilemma is the tension between performance and costs. As a result, network operators are often forced to carefully consider these costs and make ad-hoc decisions. In some cases, this involves making delicate continuous trade-offs, such as limiting the aggressiveness of auxiliary functions despite evidence of the benefits of finer granularity [195, 105, 81, 124]. In others, the trade-off is binary, and operators may, unfortunately, have to lose the efficacy of functions such as correctness guarantees due to prohibitive costs [196, 133].

In this light, we ask the question: Are these trade-offs fundamental, or are they merely artifacts of existing designs? Are there ways to maintain their performance while minimizing costs and the impact on regular operations?

1.1.2. Underutilization in Terabit Networks

Somewhat surprisingly, modern networks often harbor a substantial amount of *wasted* capacity—an unwanted by-product[1] despite efforts to improve resource efficiency while optimizing performance for application workloads.

A case of idle cycles in Ethernet links. Recent studies, including our own measurements, reveal

[1]Here we primarily refer to waste as unused capacity *after* today's resource use by applications, including any optimizations and efficiency improvements.

pervasive underutilization in Ethernet links, especially at high-resolution time scales. As mentioned, this underutilization is in spite of employing custom TCP protocols, complex traffic engineering, and prioritization schemes and also applies to networks that primarily handle large bulk-data transfers [193, 195, 204, 192]. The root causes of this waste are fundamental and span from resource provisioning to the online allocation process.

Gap between resource provisioning and online demand: Network architects and operators often provision bandwidth capacity for peak loads, leading to wasted bandwidth during off-peak times. This occurs not only in the form of tidal patterns at coarser time scales but also due to unpredictable micro-bursts at microsecond intervals, caused by faster networking [87, 121, 62, 204]. The root cause is the inherent uncertainty and high temporal variance in workloads. Recent studies have shown that the median usage of NIC-connected links can be orders of magnitude smaller than the peak load on most servers in production data centers [44, 50]. Additionally, even with predictable workloads, hardware procurement involves fundamentally discrete, coarse-grained options, forcing imperfect match against the application demand. These margins scale and accumulate across hundreds of thousands of machines and links at the data center level, contributing to the gap.

Sub-optimal resource allocation: Achieving perfect resource allocation is inherently challenging. For instance, despite decades of research in congestion control, recent studies have identified deficiencies in the backoff behaviors of state-of-the-art transport protocols for high-bandwidth links, resulting in underutilized bandwidth that cannot be reclaimed by contending flows [40]. More fundamentally, real-world application traffic patterns are often far from ideal, potentially excluding solutions to achieve full utilization of all links. For instance, spatial localities across sender-destination pairs, such as partition-aggregation traffic where a frontend server receives incasted traffic from multiple response senders, prevent tighter bin-packing of flows. Unfortunately, we cannot alter these given traffic patterns to fit idealized permutations. Moreover, any changes in conditions, such as failures or workload variations, can lead to potential waste during the adaptation of the allocation scheme.

Structural asymmetry: Underutilization can also stem from asymmetric capabilities between up-

Category	Examples
Compute	Switch data plane [197, 158] and on-board CPUs [193, 204], SNIC computes [125], SLBs [147]
Communication	Ethernet links [44], switching ASIC PCIe links [193, 204]
Storage	On-chip SRAM [205], switch CPU DRAM [109], SNIC HBMs [198]

Table 1.2: Instances of in-network wastes.

stream and downstream resources. For instance, host CPU processing bottlenecks in servers with high-bandwidth NICs can result in underutilized network bandwidth, even when CPUs are fully utilized [30, 195, 125]. Additionally, host compute resources such as CPUs and GPUs can be underutilized in practice [178, 95]. The interplay between these components becomes more pronounced as the ratio of network bandwidth to other resources, such as DRAM or CPUs, increases [12].

Broader scope of in-network waste. Combining all the aforementioned reasons, it is fundamentally challenging to fully utilize Ethernet links 100% of the time. These underutilization patterns are not limited to Ethernet links but extend to various network resources, as shown in Table 1.2. In the end, they can occur across different time dimensions, spatial locations, and vertical system stacks within contemporary scale-out data center networks. Moreover, they typically result in wasted power consumption for various components, such as middleboxes, switch transceivers, and memory [62, 195, 91, 135].

In addition, many of these in-network wastes, unlike host server resources such as CPUs, are not well-suited for direct application use. For example, switch CPUs are disconnected from the application data and are not set up for general-purpose computing [193, 192, 204]. Similarly, while SNICs are physically closer to host applications, offloading application executions using their compute resources can have negative effects, such as increased host interconnect congestion and caching disturbance, affecting application performance [164, 19, 30].

1.2. Principles of Zero-waste Designs

Inspired by observation, this dissertation explores the question: *Why not harness the in-network waste left after the primary use by applications? Is it possible to integrate auxiliary functions with near-zero*

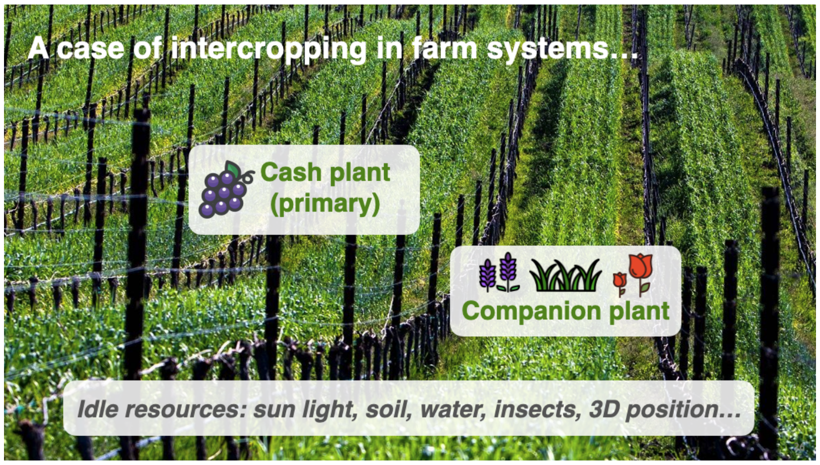

A case of intercropping in farm systems...
Cash plant (primary)
Companion plant
Idle resources: sun light, soil, water, insects, 3D position...

Figure 1.2: An illustrative example of intercropping for vineyards.

costs by exploiting in-network waste? We answer the question in the affirmative. To illustrate the intuition, we make an analogy to intercropping in agriculture (Figure 1.2). In vineyards, primary crops, such as grapes, are often paired with companion plants such as cover crops, lavender, and rosemary. These companion plants utilize idle resources such as sunlight and spatial differences to enhance the overall farm system, improving soil fertility, microclimate, biodiversity, disease management, and pest control—all without negatively impacting primary crops [120, 177].

Characterizing zero-waste designs. We note that the above goals are distinct from that of existing efforts toward high-efficiency designs, where operators, given a user workload, output a network that optimizes end-to-end performance metrics (e.g., throughput) while minimizing resource usage. Instead, our goal is to, *given the workload and network from the previous step*, maximize the utility of that network, i.e., minimize the waste—we characterize our approach as *zero-waste designs*[2].

We find that this distinction in objective suggests a broader scope of practices to minimize waste beyond existing efforts on resource reduction or scheduling. Moreover, by unleashing the potential of in-network waste and exploiting the characteristics of the underlying environment, we can navigate previous trade-offs effectively and support auxiliary functions with minimal costs and impact on existing applications while maintaining their efficacy.

[2]The term ‘zero-waste’ is familiar in the context of daily sustainability practices such as garbage recycling or upcycling for greater environmental or artistic value [29]; here, we extend the notion to networked system designs.

Thesis: *By harnessing the existence of prevalent in-network waste, zero-waste designs can support auxiliary functions both effectively and cost-efficiently.*

Rationale for harnessing in-network waste. While the intuition to exploit the in-network waste left by applications (§1.1.2) is conceptually straightforward, we delve deeper into why it is fundamentally beneficial to harvest these resources. One general finding is that many underutilized resources incur costs even when idle. For instance, keeping a server fully idle still incurs the cost of embodied carbon—which is now recognized as a major contributor to Information and Communication Technology (ICT) emissions [79]. Therefore, utilizing them is 'free' to some extent.

One might question if using these resources could lead to additional costs, such as increased power consumption. However, pushing towards full utilization remains beneficial. For example, in terms of power, the relationship between power consumption and utilization is concave: power consumption increases marginally with utilization [63, 41, 91, 144, 195]—a behavior common in CPUs, DRAMs, or switching ASICs—partly due to the dominant idle power component. Consequently, consolidating workloads to maximize utilization is advantageous[3].

Elements of zero-waste designs. Creating zero-waste designs and developing primitives that unlock their full potential is challenging and context-specific—there is no one-size-fits-all mechanism. Here, we highlight key questions to navigate for each custom scenario.

How to effectively identify in-network waste? Identifying in-network waste can be straightforward in some cases but challenging when it occurs at finer time scales or varies with fluctuating user traffic loads. Understanding its patterns and fundamental causes, particularly in the spatial dimension (such as the dynamic interplay among different resources), adds complexity. How can we identify waste online, especially when pushing the granularity to extremes?

[3]Subject to the thermal operating range of the underlying system. Note that the Power Usage Effectiveness (PUE) in modern data centers, which accounts for the 'wasted' energy (i.e., the consumption that does not get to computers), is around 1.10 and improves over time with practices such as underwater datacenters [148, 1, 11], meaning that the cooling overhead is relatively small compared to effective power consumption.

How to integrate auxiliary functions using in-network waste? How to ensure a near-zero impact on user applications? Control functions differ from user functions in their execution model and are not part of the traditional resource scheduling pipeline. They can encompass a larger set of resources, some of which applications will never touch (e.g., switch PCIe). These resources may not even be running on the same hardware, often along the end-to-end path. Furthermore, integrating auxiliary functions should have a negligible impact, ensuring minimal disruption to metrics such as throughput, latency, consistency requirements, and power consumption. How can we achieve near-zero impact when multiplexing resources? This prompts the consideration of worst-case scenarios when designing new primitives.

How about the performance of integrated functions themselves? What if there are no idle resources? Even if auxiliary functions utilize idle resources, they may have their own performance requirements. This necessitates consideration of scenarios where idle resources are absent and the mechanisms to handle them.

1.3. Contributions

This thesis explores the instantiation of zero-waste designs in modern networks by presenting three novel systems for monitoring and control functions across data, control, and management planes.

OrbWeaver (NSDI 2022) [195] is a weaved stream abstraction that reuses idle cycles in Ethernet links at 100s of nanosecond granularity for state-of-the-art in-band control protocols.

- By carefully scavenging these idle cycles using the fine-grained control capabilities of programmable switches, OrbWeaver's communication channel incurs near-zero overhead on user packet throughput, latency, buffer usage, and switch power consumption.

- It also provides comparable or better performance for control applications such as failure detection, time synchronization, and congestion feedback, while eliminating their messaging overheads.

Mantis (SIGCOMM 2020) [193] is a reactive transaction interface that recycles underutilized switch-local resources such as on-board CPUs (which are increasingly capable) and the PCIe channel (connecting with programmable ASICs), tightly coupling them for fine-grained (at 10s of μs granularity) closed-loop control functions.

- Unlike traditional control planes, which are much slower than typical data center congestion events, and pure data-plane approaches, which struggle with advanced computations due to hardware limitations, Mantis co-designs data/control plane by repurposing these resources for high-frequency, synchronous interactions. By pushing the critical logic of the auxiliary functions closer to the events they react to, this localized closed-loop primitive also mitigates the divergence between bandwidth and latency.

- Mantis also automates the flexible definition of in-network functions that sense and react to current network conditions at the sub-RTT time scale—such as hash polarization mitigation, gray failure handling, and DoS attack defense—while maintaining consistency requirements and ensuring no interruption or impact on line-rate user traffic.

Beaver (OSDI 2024) [196] is an optimistic gateway marking primitive that reduces the additional hardware procurement and instrumentation costs from the outset for enabling practical partial snapshots of distributed cloud services.

- Rather than adopting classic snapshot primitives that require the instrumentation of all involved machines, Beaver eliminates the coordination costs of external servers entirely. Furthermore, it avoids the excessive overhead typical of generic distributed system designs by tightly coupling its protocol with the regularities of the underlying data center environment. By leveraging the placement of software load balancers in public clouds and their associated communication pattern, Beaver also removes the need for additional processing servers or any blocking overhead on existing user communication.

- While incurs minimal costs for current data center operations, Beaver guarantees causal consistency for a wide range of use cases such as distributed deadlock detection, garbage collection, and integration testing.

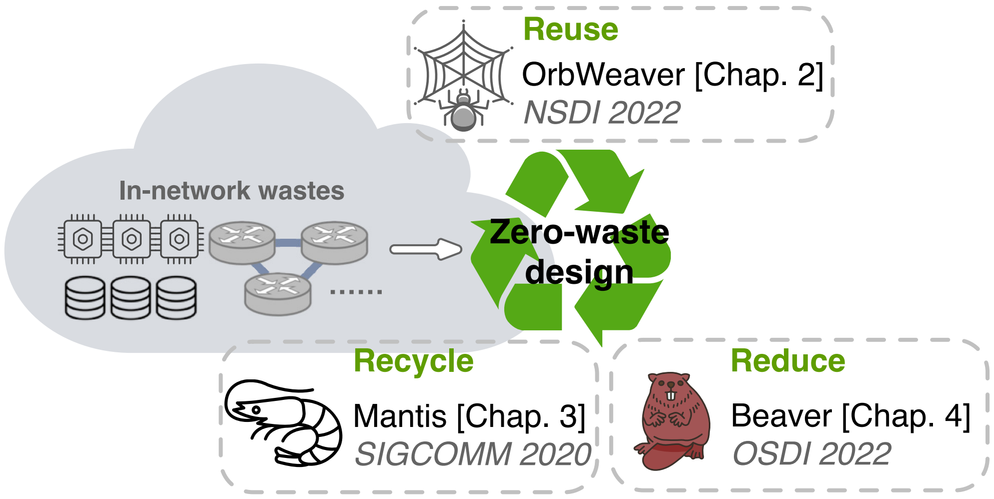


Figure 1.3: Zero-waste designs presented in this dissertation.

Summary. This dissertation introduces a zero-waste design perspective for rethinking and minimizing waste in today's networks. As an initial exploration, each system presented instantiates practical primitives and illuminates the practices of zero-waste designs (reduce-reuse-recycle[4]) to upcycle in-network waste and enable performant auxiliary functions at near-zero cost, as shown in Figure 1.3.

With the relentless increase in application demands, technology scaling slowdowns in the post-Moore era, and pressing environmental concerns, we posit that zero-waste designs will increasingly become a norm. In fact, Mantis, for example, has inspired Google to upcycle their idle switch resources for valuable auxiliary tasks such as traffic management, congestion control, and network debugging. A variant of Mantis has been deployed fleet-wide at Google across a heterogeneous set of switches with varying programmability capabilities, providing fine-grained visibility into network behaviors and serving emerging latency-sensitive workloads. It has also demonstrated practical utility in resolving real-world production issues, including protocol debugging & evaluation, root cause analysis of network outages, and the design of network functions. Additionally, it has influenced the design of an evolvable INT solution deployable in large-scale production networks (reflected in an IETF standard draft [152]), and has also led to a collaborative paper under submission [192].

[4]Inspired by the three-Rs for waste minimization [29, 99].

1.4. Roadmap And Previous Publications

The remainder of this dissertation is organized as follows. Chapter 2, 3, and 4 dive into the three instantiations of zero-waste designs, including OrbWeaver, Mantis, and Beaver. Finally, Chapter 5 discusses other collaborative work on zero-waste designs, summarizes lessons learned, and envisions future work.

Previously publications. Three of the chapters are adapted from previous publications, where the dissertation author is the primary author. The source code for all systems presented in this dissertation is available online.

- Chapter 2 revises: (NSDI 2022) **Liangcheng Yu**, John Sonchack, and Vincent Liu. *OrbWeaver: Using IDLE cycles in programmable networks for opportunistic coordination*. In 19th USENIX Symposium on Networked Systems Design and Implementation [195].

- Chapter 3 revises: (SIGCOMM 2020) **Liangcheng Yu**, John Sonchack, and Vincent Liu. *Mantis: Reactive programmable switches*. In Proceedings of the ACM SIGCOMM 2020 Conference [193].

- Chapter 4 revises: (OSDI 2024) **Liangcheng Yu**, Xiao Zhang, Haoran Zhang, John Sonchack, Dan Ports, and Vincent Liu. *Beaver: Practical partial snapshots for distributed cloud services*. In 18th USENIX Symposium on Operating Systems Design and Implementation [196].

Other co-authored publications:

- (SIGCOMM 2022) **Liangcheng Yu**, John Sonchack, and Vincent Liu. *Cebinae: scalable in-network fairness augmentation*. In Proceedings of the ACM SIGCOMM 2022 Conference [194].

- (SIGCOMM 2022) Yiran Lei, **Liangcheng Yu**, Vincent Liu, and Mingwei Xu. *Printqueue: performance diagnosis via queue measurement in the data plane*. In Proceedings of the ACM SIGCOMM 2022 Conference [118].

- (SIGCOMM 2023) Xinyi Chen, **Liangcheng Yu**, Vincent Liu, and Qizhen Zhang. *Cowbird:*

Freeing cpus to compute by offloading the disaggregation of memory. In Proceedings of the ACM SIGCOMM 2023 Conference [53].

- (NSDI 2025) Yinda Zhang, **Liangcheng Yu**, Gianni Antichi, Ran Ben Basat, and Vincent Liu. *Enabling Silent Telemetry Data Transmission with InvisiFlow*. In 22nd USENIX Symposium on Networked Systems Design and Implementation [205].

CHAPTER 2

REUSING IDLE LINK CYCLES FOR IN-BAND CONTROL COMMUNICATION

Waste not, want not.

Benjamin Franklin

This chapter was previously published in press as *Liangcheng Yu, John Sonchack, and Vincent Liu. OrbWeaver: Using IDLE cycles in programmable networks for opportunistic coordination. In 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2022)*. The dissertation author led all phases of the project, from idea development to system prototyping and writing.

Abstract. Network architects are frequently presented with a tradeoff: either (a) introduce a new or improved control-/management-plane application that boosts overall performance, or (b) use the bandwidth it would have occupied to deliver user traffic.

This chapter presents OrbWeaver, a framework that can exploit unused network bandwidth for in-network coordination. Using real hardware, we demonstrate that OrbWeaver can harvest this bandwidth (1) with little-to-no impact on the bandwidth/latency of user packets and (2) while providing guarantees on the interarrival time of the injected traffic. Through an exploration of three example use cases, we show that this opportunistic coordination abstraction is sufficient to approximate recently proposed systems without any of their associated bandwidth overheads.

2.1. Introduction

The purpose of a computer network is to transmit messages to and from connected devices. The bulk of these messages are sent between two or more end hosts and are intended for use in applications therein (video streaming, web browsing, ssh terminals, stock trackers, etc). It is important to note, however, that networks are also frequently used for other purposes that are not directly related to end-to-end application traffic. These uses include but are not limited to control messages, keepalives, and probes.

In some cases, this second category of messages is sent over dedicated networks (e.g., an out-of-band control plane). Nevertheless, a significant portion is not, and for good reason. Multiplexing the traffic over a unified network results in more efficient resource utilization and helpful fate-sharing properties. For many uses, it is also required for correctness. For instance, active probing generally relies on the probe facing the same network conditions as normal traffic.

For in-band coordination, there is often a choice between fidelity and overhead. More so as many protocols use high-priority messages that directly cut into network capacity. For example, when deciding on an appropriate interval for sending routing-protocol keepalive messages, sending keepalives more frequently results in faster failure detection but at the cost of many extra packets in the network. Similarly, while techniques like congestion tagging [36, 89] and in-band network telemetry [108] can provide timely information about the recent state of network paths, they require either extra probe packets or space in the headers of existing packets, both of which occupy valuable bandwidth.

Given this tradeoff between fidelity and overhead, today's networks end up settling for a little bit of both. In some cases, the sacrifices are modest; in others, network operators are forced to limit the aggressiveness of their systems despite evidence of the benefits of finer granularity [204, 42]. In OrbWeaver, we argue that for a diverse set of protocols, the sacrifice is entirely unnecessary—systems can coordinate at high-fidelity with a near-zero cost to usable bandwidth and latency. In short: we can have our cake and eat it too.

Our system, OrbWeaver, is a framework for the opportunistic transmission of data across today's programmable networks. OrbWeaver takes advantage of gaps between user traffic and 'weaves' (i.e., injects) into *every* such gap customizable IDLE packets that convey information across devices. For modern, high-speed networks, these opportunities are plentiful. Crucially, OrbWeaver provides guarantees about the 'weaved' stream—guarantees on the maximum time between any two packets and guarantees on the impact of the injected packets on user traffic, switch resources, and power draw. A consequence of this predictability is that, even when there is no opportunity to send, the absence of IDLE packets reveals concrete information about the state of the network.

We note that a similar abstraction already exists at the data-link layer. In particular, in today's full-duplex Ethernet standards, the Physical Coding Sublayer (PCS) will fill any gaps in transmission with IDLE symbols [173, 129]. The continuous stream of incoming signals allows receivers to—with no impact to user traffic—test for corruption and link integrity at a fine granularity, even when there is no traffic on the network. Further, by continuing to transmit IDLE symbols after a link integrity issue has been raised, switches can also determine when the link becomes usable again.

OrbWeaver extends this technique to higher layers of the network stack by exploiting the data plane programmability, architecture, and packet generation capabilities of emerging programmable switching platforms. The resulting stream of packets can be used to generalize Ethernet's robust failure detection properties to a broader class of faults; however, its benefits go far beyond L3 failure detection. Rather, we demonstrate that with proper application, the nearly free communication that IDLE packets provide can be used to eliminate the fidelity-utilization tradeoff of solutions to several classic problems in networking including clock synchronization and load balancing.

Implementing OrbWeaver's packet weaving presented several technical challenges. First, while IDLE symbols are part of the Ethernet standard and enjoy direct hardware/protocol support, to utilize today's devices and maintain their current performance, OrbWeaver must provide similar behavior without changes to switch architectures. Second, while many systems can benefit directly from opportunistic data transmission, many must continue to operate during periods where user traffic is already occupying all available bandwidth. To address the first challenge, OrbWeaver introduces a co-design of the selective data-plane filtering mechanisms and the rich priority configurations found in modern switches to guarantee minimal impact on user traffic. We verify the approach through a detailed examination of the specifications of the queuing subsystems on a Tofino switch along with experiments that stress-test worst-case behavior. To address the second, we introduce novel mechanisms that exploit IDLE packets and the guarantees of weaved streams to eliminate the bandwidth overheads of existing network protocols. We demonstrate these mechanisms through three case studies.

Our implementation[5] and evaluation demonstrate the efficiency and efficacy of OrbWeaver using real hardware, optical attenuators, and power meters. We find that, despite the introduction of the IDLE stream, OrbWeaver incurs negligible impact on user traffic, the computational/state resources of participating switches, or their power draw. We further demonstrate that this messaging substrate can be used to (re-)design recently proposed systems to eliminate their bandwidth overheads while closely approximating their performance.

2.2. Motivating Weaved Streams

This section presents the definition of a 'weaved' stream, its motive, and where data plane programmability can help.

Definition. A *weaved stream* is a union of user and IDLE packets traversing a link between two arbitrary network devices that provides two guarantees:

R1 That no link stays unutilized for too long. More precisely, there is some period τ where the interval between any two consecutive packets, d, satisfies $d \leq \tau$.

R2 That the injected packets do not decrease the effective throughput of user traffic or increase their loss rate.

Why weaved streams? Network protocols are, fundamentally, distributed computations that require coordination between different devices—sometimes adjacent devices, sometimes remote devices—for monitoring, control, and management. A perennial problem is how much bandwidth to allocate to these protocols, as each byte devoted to coordination is a byte that could have been used for user traffic instead. This tradeoff has tangible effects for many networking tasks:

- *Failure handling:* A common strategy for detecting the failure of remote network devices is the use of continuous keepalive messages. Here, each node periodically sends a keepalive to each of its neighbors; if a neighbor ever stops sending keepalives, nodes assume that they have failed. Fundamentally, the period between keepalives bounds the speed at which we can detect failures. Unfortunately, because keepalives are most accurate when sent over the same or higher-priority

[5] Code is available at https://github.com/eniac/OrbWeaver.

channels as user traffic, their sending rate is typically kept low (e.g., at a period of O(100 ms)) at the cost of slower detection.

- *Clock synchronization:* Prior work has also noted the utility of synchronizing network devices [117], e.g., for coordinated network updates [188, 150] or real-time streams [60]. Clock synchronization protocols typically pass messages that periodically compute the drift between the clocks of participating machines. Constant changes to not only the relative clocks but also the relative clock *rates* mean that more frequent updates can provide more accurate synchronization (at the cost of additional packets in the network, typically configured at a high priority).

- *Congestion notification:* Finally, this tradeoff can be seen in the detection/communication of congestion and current load. ACKs (and their corresponding loss/RTTs) are a particularly common method for inferring the presence of congestion, e.g., in TCP NewReno. As others have noted [36, 105], however, there are also advantages to more explicit signaling of the current congestion and queue statistics. Unfortunately, while effective, these statistics typically occupy packet header space or introduce additional packets into the network.

Over the years, network architects have developed many workarounds. These include hardware changes [129, 117], co-opting unused fields in headers [36, 207], carefully balancing the tradeoff for a particular service-level expectation [47], or otherwise coming to terms with the cost of coordination. Outside factors can guide the above decisions, such as whether ACKs are already necessary (e.g., for reliability) or if extraneous fields can be eliminated. However, in this chapter, we ask a more fundamental question: are these tradeoffs necessary?

To that end, OrbWeaver is a framework for implementing network coordination that does not interfere with user traffic. OrbWeaver's weaved streams are both opportunistic and highly predictable—consuming every inter-packet gap of sufficient size but no more. Not every protocol can be implemented solely using weaved streams (though many can benefit from it). Even so, we demonstrate that at least for the three use cases above, weaved streams are sufficient to approximate state-of-the-art systems while reducing their impact on user traffic to virtually zero.

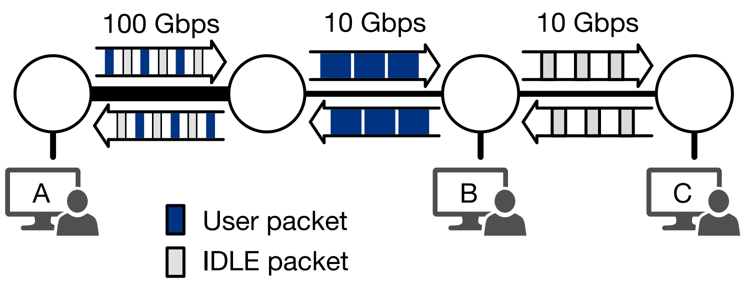


Figure 2.1: An example OrbWeaver-enabled network with four switches and three end hosts (connected with 10 Gbps links). A single two-sided connection $A \leftrightarrow B$ occupies the network, but a significant portion remains unused. Gaps between packets can occur for many reasons, but OrbWeaver can weave IDLE packets into all of those gaps.

Why are there gaps? Usable gaps between packets can occur for many reasons, the most basic being application-level patterns and TCP effects. Indeed, prior work [204, 156] and our conversations with several large clouds/ISPs verify that micro-/milli-second inter-packet gaps are ubiquitous, even in networks that primarily handle large bulk-data transfers.

Gaps can also happen for structural reasons. For example, consider Figure 2.1 (sans IDLE packets). In it, a single connection $A \leftrightarrow B$ occupies all usable end-to-end bandwidth. Even if A and B pace packets perfectly, no host can send additional packets without displacing the existing user traffic, despite significant opportunities to do so (because of, e.g., congestion, link speed changes, and asymmetric connections). These gaps present a chance for opportunistic coordination.

Why now? OrbWeaver's ability to weave IDLE packets into gaps between user traffic is enabled by several features in modern switches: programmable data plane behavior, the capacity for local packet generation, and the ability to fully configure the queuing/prioritization of different traffic classes. We note that none of these are sufficient on their own.

For example, consider strict packet prioritization, which has been used for opportunistic bandwidth allocation [94, 86]. In SWAN [86], for instance, end hosts send low-priority background traffic to capture any bandwidth remaining after handling interactive and elastic services. A naïve application of these techniques, however, is a poor fit for in-network coordination, which occurs between devices

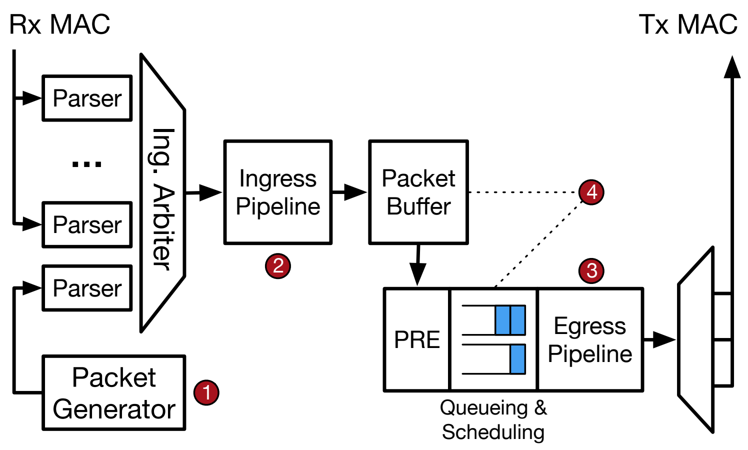


Figure 2.2: Conceptual diagram of the relevant components of an RMT switch, derived from the switch specifications in [57]. Only a single ingress/egress pipeline are shown. Circled numbers indicate steps and potential points of contention with user traffic that are handled in §2.3.1.

in the network (as opposed to end hosts) and typically involves small data sizes that benefit from even short sending opportunities. Figure 2.1, for example, would not benefit from end-host actions.

2.3. Generating a Weaved Stream

Before we dive into the potential uses of weaved streams in §2.4, we first detail how to implement the abstraction in today's programmable switches.

Switch model. For simplicity, we primarily focus on the popular Tofino family of programmable networking devices (and discuss generalization to other types of devices in §A.2). Figure 2.2 shows a conceptual diagram of the relevant components of the switches we consider. At a high level, when a packet enters from one of the Ethernet ports, its header is extracted by the programmable parser responsible for that port. An ingress pipeline arbiter is then responsible for selecting one of the parsed packets and passing it through the ingress match-action pipeline.

After ingress processing, the packet will be placed in a shared packet buffer until it is ready to be sent out. Instead, the switch uses a shorter 'packet descriptor' for the next steps: optional replication by a Packet Replication Engine (PRE) (e.g., for multicast) and placement onto a per-port egress queue for eventual processing/deparsing. The data plane program and the traffic manager configuration decide whether an incoming packet should be buffered and whether a buffered packet should be enqueued for transmission.

Goal. R1 of the weaved stream abstraction requires a constant stream of packets on every link such that the union of user and IDLE packets satisfies $d \leq \tau$. We note that the optimal guarantee for τ is dependent on both the bandwidth, B, of each link and the MTU of the network. To see why, consider the extreme case where a user is occupying all of the bandwidth of a port i with MTU-sized packets. The receiver on the other side of the link will receive packets at a period of $\tau_i = \frac{\text{MTU}}{B_i}$, with OrbWeaver unable to inject any additional packets without impacting user traffic. Therefore, unless otherwise noted, OrbWeaver uses $\tau_i = \frac{\text{MTU}}{B_i}$ even if smaller IDLE packets would allow for faster injection.

In the worst case when there were zero user packets and N egress ports, the resulting target IDLE-injection rate is:

$$T = \sum_{i=1}^{N} \frac{B_i}{\text{MTU}} \tag{2.1}$$

For reference, for a 32 port switch with $B = 100\,\text{Gbps}$ and $\text{MTU} = 1500\,\text{B}$, the per-port inter-packet gap, τ_i, is 120 ns, which results in $T = 266.7\text{M}$ packets/sec.

Constraints. Complicating the injection of IDLE packets into the network are R2 and hardware constraints on the throughput of each switch pipeline, defined in terms of both byte-level bandwidth ($N \times B$) and packet-level bandwidth (proportional to the clock rate of the pipelines). For the latter, switches typically provide guarantees up to a certain minimum packet size, and best-effort behavior for very small packets.

2.3.1. Mechanism Overview

OrbWeaver's IDLE-packet weaving leverages a combination of features found on our target platform: data-plane packet generation, data plane programmability, and fine-grained arbiter/scheduler configuration options. The switches' onboard per-pipeline packet generator modules, in particular, form a convenient substrate for our techniques. Using these modules, a network operator can create packets with predetermined content at a predetermined rate.

In principle, one could configure the generators to create packets at a rate T (thus providing OrbWeaver with its consistent stream of packets to convert into IDLE packets). Unfortunately, in practice, these generators do not have nearly enough capacity to satisfy the requirements of OrbWeaver. Moreover, blind injection of packets may interfere with the throughput, latency, or loss of user traffic. Instead, OrbWeaver uses the selective amplification method described below.

❶ **Packet generation.** The IDLE stream generation of OrbWeaver begins with a low-rate but predictable stream of generated IDLE packets. The focus of this process is to provide a 'seed' stream with an emphasis on regularity; amplification up to T occurs later in the pipeline. More specifically, the generator module is configured to send a packet every $\frac{\tau_{\text{min}}}{2}$ secs, where τ_{min} is the minimum τ_i of any port on the pipeline.

There are two important aspects of this seed stream. The first is that the rate is double that of τ_{min} in order to provide a degree of oversampling for the subsequent optimizations without sacrificing guarantees on the eventual spacing of packets. The second is that the IDLE packets are configured with a strict high priority at the ingress arbiter so that the packet will always be serviced as soon as it is generated. While this implies that IDLE packets are preferred over user traffic in the ingress pipeline, the low rate of this seed stream means that OrbWeaver incurs $<$1.5% overhead even for the worst case of minimum-sized packets sent at $\tau_{100\,\text{Gbps}}$ (denoting the optimal τ_i for a 100 Gbps link). More typical packet sizes and utilization eliminate the overhead.

❷ **Amplifying the stream on-demand.** OrbWeaver takes the low-rate seed stream above and amplifies it, potentially up to the full rate T, by leveraging another hardware feature found in modern switches: flexible multicast. In Figure 2.2, this behavior is implemented in the PRE, which can replicate a packet descriptor to the egress queues at line rate.

Unfortunately, the naïve approach of replicating a packet to every egress queue every τ_{min} seconds can crowd out normal multicast packets and waste significant egress capacity. More specifically, there are two instances where it is not necessary to multicast a packet to a particular port i:

1. If the port is slower than the maximum speed, then sending at τ_{min} will be too fast by a factor

of $\frac{B_{\max}}{B_i}$.

2. If a user packet was already sent to the egress port recently, sending an IDLE packet is unnecessary.

OrbWeaver addresses both cases by oversampling the sending history of each port (at rate $\frac{\tau_{\min}}{2}$) and then selectively filtering/multicasting toward only the ports that need an IDLE packet. When a port is has bandwidth $B_i < B_{\max}$, the switch downsamples the IDLE packets by configuring two multicast groups (one with port i and one without) and picking the one with i every $\lceil \frac{B_{\max}}{B_i} \rceil$ packets. Similarly, if a port has sent a packet (user or IDLE) in the past $\frac{\tau_{\min}}{2}$ seconds, we can select a multicast group that does not contain the given port.

Concretely, this filtering step uses a single stateful register entry with a bit width equal to the number of ports attached to the pipeline. In essence, the register is a bitvector where each bit represents whether we have sent a packet to the associated port within the last $\frac{\tau_i}{2}$ seconds. For every incoming seed packet, if the associated bit is 1, we omit the port and flip the bit to 0; if the bit is originally 0, include the port in the multicast and flip the bit to 1. Specifically:

```
user packet: filter_reg |= 1 << egress_port
seed packet: filter_reg ^= speed_mask
```

When all ports are the same speed, `speed_mask` is always $2^N - 1$; for hybrid configurations, the ith bit is 1 for every $\lceil \frac{B_{\max}}{B_i} \rceil$ packets and 0 otherwise. After updating the register, OrbWeaver multicasts the current seed packet to the multicast group specified by `filter_reg` (in particular, its value before the xor)—if and only if bit i in the multicast group ID is 0, port i is included in the multicast.

In principle, a direct application of the above filtering step guarantees that the PRE will have enough bandwidth for all user multicasts, assuming that each user multicast results in at most one packet on each egress port. Two aspects of modern switch design potentially complicate this design.

The first is that today's switches typically cannot support a unique multicast group for each of the 2^N possible combinations of target ports. OrbWeaver addresses this by reducing the number of groups by coalescing ports into groups of M such that, if any port in the set has its bit in `filter_reg` set, the entire set receives the multicasted packet. This approach trades a factor of 2^M reduction in the number of multicast groups for a worst-case $\frac{M-1}{N}$-factor decrease in PRE bandwidth. The second is that modern switches are often composed of different pipelines, each supporting distinct packet generators, sets of registers, and groups of ports. Lack of visibility across pipelines means that `filter_reg` may only track local sends, which can also lead to higher PRE usage.

We note, however, that in both of the above cases, OrbWeaver will only incur false negatives (and no false positives) of user packet presence, thus satisfying R1. We also note that very few modern networks are continuously multicasting to all ports at near line-rate.

❸ **Weaving the IDLE stream between user packets.** After the stream is amplified, it reaches the egress queues and pipeline of the switch. To bound the impact of the stream on user traffic, OrbWeaver configures its packets to have a strictly lower priority than any other user traffic on the same port. If there is user traffic to send, the IDLE packets will not impact them; if there is no traffic to send, the IDLE packets will be sent at a minimum rate of τ_i per port i. The only potential impact to the latency/throughput of user traffic is when an IDLE packet is scheduled just before a user packet arrives, in which case the user packet will be delayed by at most `pkt_size`$/B_i$. The delay is only incurred once per packet burst, which implies a bound on OrbWeaver's end-to-end impact on latency and throughput.

Upon arriving at the ingress pipeline of the downstream switch, the packets will be dropped. This also has near-zero impact on user traffic as IDLE packets are only received when the upstream switch has nothing to send.

❹ **Managing the packet buffer and egress queues.** Finally, through the above process, there are two primary places where IDLE packets can compete with user packets for memory in addition to bandwidth. The first is the per-egress output queues that hold packet descriptors before they are serviced by the egress pipeline. The second is the shared packet buffer that stores packet contents until they are sent out on the wire.

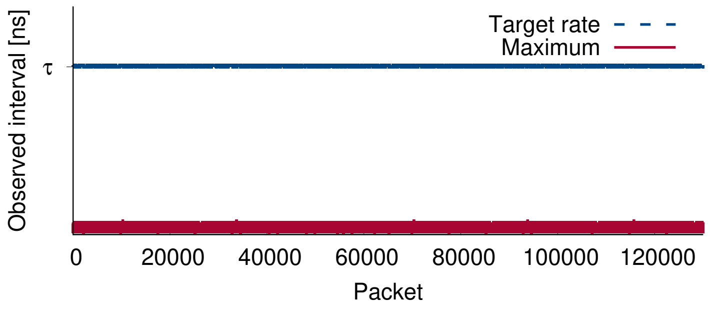


Figure 2.3: An empirical evaluation of the switch's capacity to generate IDLE packets. Packets were injected to all ports, but the graph depicts the observed inter-packet gap at only one of those ports. Results are shown for both the target rate ($B_i = 100\,\text{Gbps}$, MTU $= 1500\,\text{B}$) and the maximum achievable rate. y-axis omitted to protect confidential information.

To bound the impact of OrbWeaver on both resources, we statically carve the buffer using egress and ingress buffer accounting mechanisms, respectively. For the former, we note that the queue for IDLE packets (the lowest priority queue for the port) is distinct from those of user packets. This queue only needs to be one cell deep as another IDLE packet is guaranteed to arrive in a timely fashion, and thus, the impact on aggregate memory capacity is negligible. For the latter, we can likewise keep the required buffer shallow because of the guarantees of the packet generation process. Specifically, we can confine the IDLE packets to a fixed-size, non-shared region of the packet buffer. The buffer only needs to have a depth equal to the sum of the egress, per-port IDLE-packet queues plus a small amount of headroom for any potential cycle-level processing delays. This is $< 0.01\%$ of the total buffer size of a typical modern switch.

2.3.2. Evaluating the Weaved Stream

In this section, we delve deeper into OrbWeaver's potential impacts on user traffic. We do this with the assistance of a prototype implementation on a $2\times$ Wedge 100BF-32X testbed. Additional experiments can be found in §A.6.

2.3.2.1 Can OrbWeaver Inject at Rate T?

To demonstrate that our approach can achieve T on a fully provisioned switch, we validate it empirically. Specifically, we configure a switch with all 32 ports active and running at a full 100 Gbps. We then configured the switch's packet generator module to generate seed packets at a rate of $2/\tau_{100\,\text{Gbps}}$

and then multicast every other IDLE packet to all ports.

Figure 2.3 shows a time series of the interval between IDLE packets, as observed by the egress pipeline of a single port. To record the series, we maintained a ring buffer (implemented via a data plane register) of the difference between the current `egress_global_tstamp` and the previous. The observations were maintained in the egress pipeline and for a single port (other ports' results are identical).

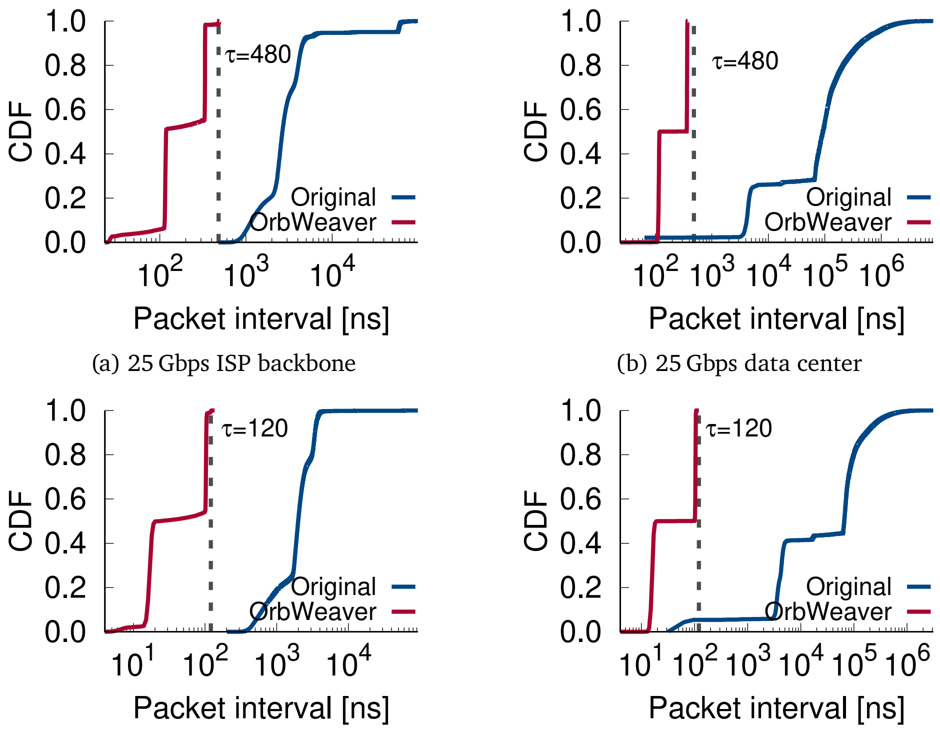
We find that, not only is the injected stream able to achieve $\tau_{100\,\text{Gbps}}$ for every port simultaneously, the observed rate is stable across packets. Further, increasing the amplification factor of the multicast configuration enables IDLE packet generation more than an order of magnitude faster than the target interval, $\tau_{100\,\text{Gbps}}$. Among other implications, this means IDLE packet injection is robust to higher bandwidth and lower MTUs, even without improvements to packet replication capacity.

2.3.2.2 Can OrbWeaver Bound Packet Gaps?

In addition to being able to generate IDLE packets at rate T, R1 also requires regularity in the form of a bound on the gap between packets. We note that Figure 2.3 already demonstrates the regularity of this gap on a switch without traffic. We also note that in the other extreme (when ports are always congested), R1 is trivially satisfied.

In this section, we extend these results to a network with burstiness and varying levels of traffic. Specifically, we use a hardware testbed consisting of two OrbWeaver-enabled switches (A and B) and a set of servers connected to A. User traffic is passed hosts$\rightarrow A \rightarrow B$ with amplification to fully utilize the ports at B. For this experiment, we used `tcpreplay` and pcap traces from an ISP backbone [10] and a cloud data center [48]. We set up a register in the ingress pipeline of the downstream switch B to record the distribution of the interval between consecutive packets.

Figure 2.4 shows the results for a single 25/100 Gbps port. Without OrbWeaver, very few intervals are under τ for the target link speed, and the tail is very long. OrbWeaver, on the other hand, is able to weave in IDLE packets to guarantee an upper bound on the packet interval regardless of the original traffic pattern. In particular, for a configured generation interval of t ns, out of 2.14×10^9



(a) 25 Gbps ISP backbone (b) 25 Gbps data center (c) 100 Gbps ISP backbone (d) 100 Gbps data center

Figure 2.4: Observed intervals between packets with/without OrbWeaver's weaved stream. The dotted line shows the ideal period τ for each link speed. Without OrbWeaver, the maximum interval was $>$100s of ms but we truncate for readability.

interarrival periods, the maximum observed interval was $(t + 3)$ ns (observed for only 32 intervals). The discrepancy is likely due to either clock drift or the aforementioned cycle-level processing delays. Notably, the presence or absence of cross traffic had negligible effect on the frequency of these 3 ns outliers so in practice, we can set $t = \tau - 3$ and achieve reliable results.

Explanation. The regularity of OrbWeaver's weaved stream derives from the architecture of the switch and the mechanisms of OrbWeaver. From the components of Figure 2.2, the parser used by the packet generator is separate from those of the external traffic, the ingress pipeline grants strictly higher priority to the generated packets over external traffic (user or IDLE), and the packet buffer protects IDLE packets from interference through static reservations for worst-case capacity. When combined, a generated IDLE packet can only be delayed through HoL blocking when an external packet arrives just before the generated packet. For unicast packets, this is a 1-cycle delay; for full

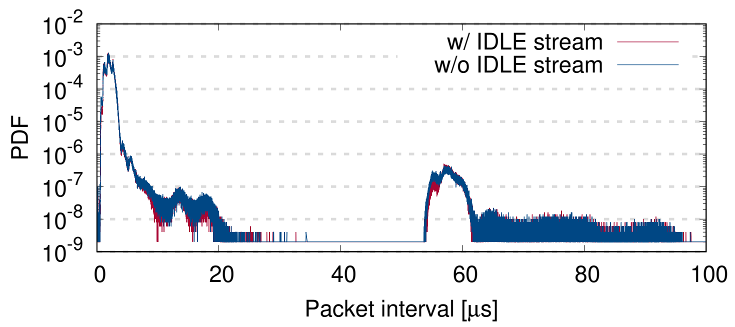


Figure 2.5: The impact of IDLE packets on user traffic at the ingress pipeline with/without a generation rate of $2/\tau_{100\,\text{Gbps}}$.

broadcasts, this is up to an N-cycle delay (which is short for today's high-speed networks).

At the egress pipeline, the priorities are reversed: IDLE packets are set to a strictly *lower* priority than user traffic. This change stems from a change in objective: in the egress pipeline, it is no longer necessary for the IDLE packets to be sent at a precise rate; instead, the goal is to send *any* packet at above the minimum rate, τ_i. Choosing a user packet instead of an IDLE one can only decrease the inter-packet gap.

Note that, in a Tofino, these priorities (unlike those at the ingress) are only effective within their respective ports. Thus, the switch will send a low-priority packet on port i even if there is a higher-priority packet queued for a different port. As long as the average packet size is above the minimum for line-rate processing, ports can be considered in isolation.

2.3.2.3 Do IDLE Packets Affect External Traffic?

As important as the impact of cross traffic on generated IDLE packets are the impacts of the generated packets on (1) user traffic and (2) incoming IDLE packets. A significant impact on (1) implies violations of R2; on (2), it implies inaccuracy in inter-arrival times and potential violations of R1. We discuss potential impacts in the two pipelines separately.

Ingress pipeline. While OrbWeaver's packet prioritization means that IDLE packets will be preferred over external traffic in the ingress pipeline, its use of multicast amplification reduces their impact to 1.5% of maximum packet-level capacity, with zero impact to byte-level capacity.

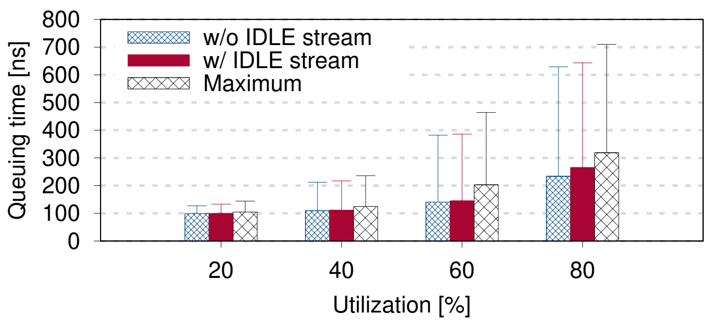


Figure 2.6: The impact of IDLE packets on the latency of user traffic at the egress pipeline. Results are shown for various levels of average utilization. 0% and 100% are not shown as OrbWeaver becomes trivially optimal. To provide an upper bound on the impact, we disable adaptive ingress filtering and populate the pipeline with only small (64 B) user packets. A real OrbWeaver deployment would have much lower impact.

To evaluate the practical effects of this overhead, we replayed a real-world packet trace over an ingress pipeline of an OrbWeaver switch. The packet trace was generated using `tcpreplay` and link-level packet traces captured from 10 Gbps Internet routers [10]. To saturate the pipeline, we sped the traces up to match our setup's 100 Gbps per-link bandwidth and replicated them to fill the switch.

We compare two cases. In the first, only the above external traffic is present. In the second, we used the exact same traces but, in parallel, we injected IDLE packets into the same pipeline just as we did in the previous subsection. In both cases, we measured the packet count and interarrival times of user packets in the ingress pipeline with the help of stateful registers that aggregate both statistics.

We find that, for the speeds and packet sizes in the evaluated trace, the throughput and congestion loss of user traffic is the same whether the generated IDLE stream is present or not. The only metric that is impacted is latency, where a slight delay can be introduced each time a generated packet is processed one 'clock cycle' ahead of a user packet; however, this is minor and mitigated by the low frequency of IDLE packet injection. Figure 2.5 depicts the cumulative impact of this delay using a histogram of the packet interarrival time of the traces, with and without the IDLE stream—the majority of the differences are due to randomness in `tcpreplay` between executions, rather than OrbWeaver.

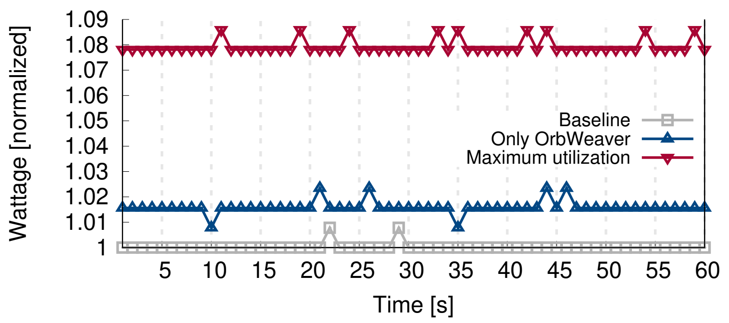


Figure 2.7: The power draw of a OrbWeaver switch compared to that of an idle (baseline) and a maximally utilized switch. Y-axis is normalized to the average power draw of the baseline.

Egress pipeline. The benefits of the amplification strategy to contention mitigation stop at the PRE, but two other factors take its place in ensuring that user traffic is not impacted in the egress pipeline. The first factor is the filtering step that was introduced in Section 2.3.1, which prevents superfluous usage of both the PRE and egress pipeline when the egress ports are already occupied. For IDLE packets that are not filtered in the ingress pipeline, the second factor is the strict prioritization of user traffic over IDLE packets of the same port, also introduced in Section 2.3.1. The second factor, in particular, provides an upper bound on the impact of the IDLE packets as long as the user traffic respects the minimum average frame size requirements of the switch specification (see Appendix A.4 for a formal analysis).

To truly stress these mechanisms, we evaluate an extreme scenario in which multiple hosts send minimum-size (64 B) packets toward a single egress port and OrbWeaver's filtering mechanism is disabled. This situation is not possible in OrbWeaver, but is helpful in demonstrating the efficacy of egress prioritization for protecting user traffic. The results verify the analysis above, even for high user-traffic utilization. For comparison, we also show the impact of an IDLE stream operating at the order-of-magnitude-higher maximum rate of Figure 2.3 but still set to low-priority. Again, across all experiments, throughput was unaffected.

2.3.2.4 Does Injection Affect Power Usage?

Finally, we investigate the impact of weaving on the power consumption of today's switches. A natural concern is that the continuous stream of packets will increase consumption; however, we find the actual impact is minimal as the underlying Ethernet MAC already continuously sends IDLE symbols.

To evaluate this, we used a P3 Kill-A-Watt Electricity Usage Monitor (Model P4400) to measure the total power draw of a Wedge100BF-32X programmable switch. The monitor sits between the switch's power plug and its power outlet and can measure wattage to within 0.2–2.0%. To emulate the switch's deployment into a network of programmable switches, we connect every port on the switch to a second switch that logically functions as 32 neighboring switches. We test three distinct configurations:

- *Baseline:* All ports on the switch are connected at 100 Gbps; however, the switch is otherwise inactive, i.e., there is no incoming traffic nor any IDLE packets.
- *Only OrbWeaver:* Same as above, but with OrbWeaver's IDLE stream generation enabled on all switches. The switch is, thus, both sending and receiving packets at T.
- *Maximum utilization:* The worst case scenario, where the switch is both sending and receiving user packets at the maximum rate and generating IDLE packets (that are eventually dropped in the ingress pipeline).

Figure 2.7 shows the power draw of each configuration over a 1 min period. OrbWeaver's transmission of packets at rate T increases the average power draw of the switch by $<$2%.

2.4. OrbWeaver Use Cases

Figure 2.8 outlines the general structure of a P4 program that uses OrbWeaver. Whereas a standard P4 program processes a stream of user packets, an OrbWeaver P4 program processes a weaved stream of user and IDLE packets. OrbWeaver programs can append/read information from the payloads of the IDLE packets (which appear on the wire as a special EtherType) or infer statistics from the timing of the weaved stream. In either case, the content of IDLE packets can be manipulated just like any other packet (metadata like the drop decision, priority, or egress port should not be changed).

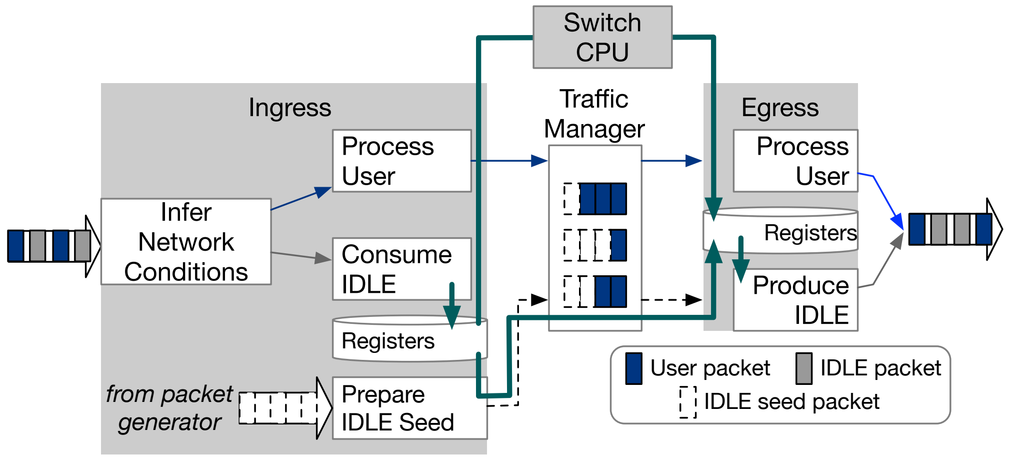


Figure 2.8: Structure of a P4 program that processes a weaved stream. The ingress pipeline extracts information from the weaved stream, then processes user and IDLE packets separately. The egress pipeline processes user packets and transforms seed packets into IDLE packets. Pipelines can communicate using registers that are synchronized with either seed packets or the switch CPU, as shown by the thick lines.

In typical usage, the receiving switch will process, record, and drop incoming IDLE packets before the end of the ingress pipeline. In most cases, the IDLE packets bypass the normal pipeline logic and, thus, will not affect user byte/drop/error counters. Separately, they use either (a) an agent on the switch CPU [193] or (b) a locally generated IDLE seed packet to transfer data from the ingress to the egress pipeline before sending to the downstream switch. Together, they facilitate multi-hop communication over IDLE packets.

In this section, we detail three example use cases of OrbWeaver (see §A.1 for others). For each example, we consider a recently proposed network system, and we explore how well OrbWeaver can approximate it without introducing any additional impact on user traffic. We note that, in some cases, this restriction can result in suboptimal designs (i.e., imposing on user traffic may result in better overall performance, even if it incurs overhead). Rather, we ask: how far can operators go before needing to ever consider the choice between network throughput and features?

2.4.1. Fast Failure Detection

Failures of network components are common in large networks where the number of devices involved ensures a constant flow of incidents. Reasons for the failures include overheating components, power instability, bit flips in the signal, loose transceivers, bent fibers, or any number of other causes [210, 185, 209, 73]. In the end, however, the symptom of many of these failures is the same: lost packets in the network.

Thus, as the first steps toward mitigation, quickly detecting and quantifying packet loss is critical to maintaining high availability and stringent SLOs, particularly as networks improve in both bandwidth and reaction time such that control-plane processing is no longer the sole bottleneck [193, 55, 124, 105, 127, 129]. Unfortunately, as mentioned in §2.2, common detection approaches—periodic keepalives or pings—force network architects to sacrifice detection latency to constrain overheads. Moreso as pings are traditionally prioritized over user packets to minimize false positives.

Even recent systems like NetSeer [207] that track user-packet loss inband (without injecting additional packets) suffer from this tradeoff. For example, NetSeer's choice to not inject additional packets means that the network is necessarily slow to detect a black hole (differentiating from a lack of demand requires CPU coordination to compare the flow counters of adjacent switches). Likewise, their choice to tag every packet with a sequence number incurs a bandwidth overhead of 0.3%∼6.3% in return for higher detection granularity (unless there are previously unused bits in the header and we cannot change the data plane to remove them).

2.4.1.1 An OrbWeaver Redesign

Taking NetSeer as a base, we can replace its inter-switch communication with an OrbWeaver-influenced design to eliminate bandwidth overheads and significantly improve detection time. We refer readers to the original paper [207] for full details of the existing system but summarize the relevant components as follows. NetSeer records the 5-tuple of each packet in the egress pipeline using per-port ring buffers and tags it with a 4-byte sequence number. The downstream switch stores the last observed sequence number. Upon detecting a gap (e.g., packet 14 after packet 12), it sends 3 duplicate and high-priority drop notifications to the upstream switch for each missing sequence number. If the upstream switch receives at least one such notification, it will use the records in the ring buffer to generate a flow event for the missing packet, which will be compressed/summarized for the management plane.

In NetSeer-OW, switches maintain per-port hash tables that, like NetSeer, record the 5-tuples and packet counts of passing flows (using the 5-tuple hash as the index). The caches are maintained in the egress pipeline of each upstream switch as well as the ingress pipeline of each downstream switch. As channels are FIFO and the tables use the same size and deterministic hash function, their content should always be identical. The only exceptions occur after a packet loss, at which point either a counter or a 5-tuple will differ.

In this re-design, user packets are *not* tagged with any additional data nor does it require triple-notifications. Instead, the upstream switch will opportunistically embed in IDLE packets psuedo-randomly selected cache records[6]. If the downstream switch finds that a record differs from its local copy, it will generate an event for the contained 5-tuple. It will also generate an event if packets stop arriving, which is detected with locally generated IDLE seed packets that scan per-port weaved-stream counters. After NetSeer-OW compresses/filters these events, the control plane sends the results over a low-priority TCP connection to the central controller.

Note that, in addition to exploiting the IDLE stream to carry flow information, (R1)'s guarantee of packet arrival rates enables provably optimal detection speed of link failures. In principle, OrbWeaver can trigger an alert if the `ingress_mac_tstamp` of any two consecutive packets is $\leq \tau$. While that level of granularity may be too aggressive for many networks, we note that recent proposals for data plane rerouting have made detection speed a bottleneck [129, 124, 55, 105], particularly if a goal is zero-loss failure recovery. In the end, the point is that OrbWeaver can provide arbitrarily precise failure detection/statistics for current and future networks.

Dealing with a lack of sending opportunities. While extended periods of maximum utilization are rare in most networks [204, 166, 10], NetSeer-OW can still provide useful properties during these extreme conditions. For example, for failure detection, a downstream switch in a fully utilized network can immediately detect a packet drop by examining the gaps between adjacent packets (a drop occurred when the gap $> \tau$).

[6]To improve the update rate, we can pack up to three 5-tuple-counter records (IPv4 and counters of 3 B) in each packet. To handle register access limitations, we can pack the records or split the table across multiple arrays.

Flow attribution is slightly more challenging, with the chief concern that the switch evicts the flow before including it in an IDLE packet. We can quantify the probability of this happening using the formalization in §A.5. For reference, using the assumptions of §A.5, average utilization of [166, 10], and flow cache performance of [170], ISP routers with 128 cache entries per port would have a $P(\text{notified}) \approx \frac{0.72}{0.72+0.28*0.45/3} = 94.4\%$. A data center switch with 128 cache entries would have $P(\text{notified}) \approx \frac{0.75}{0.75+0.25*0.16/3} = 98.2\%$, or with 512 entries $P(\text{notified}) \approx \frac{0.75}{0.75+0.25*0.05/3} = 99.4\%$.

Benefits. Compared to the original NetSeer design, the primary benefit of the OrbWeaver augmentation is to completely eliminate *all* sources of bandwidth overhead—in essence, we can apply NetSeer for 'free.' In particular, it eliminates the overhead of sequence number tagging (0.3%∼6.3%) of capacity; the replicated, high-priority failure notifications (up to 100% of reverse link capacity); and the impact on user traffic of the event reports. Beyond overhead, it also improves the speed to detect inter-switch failures, particularly during periods of low utilization.

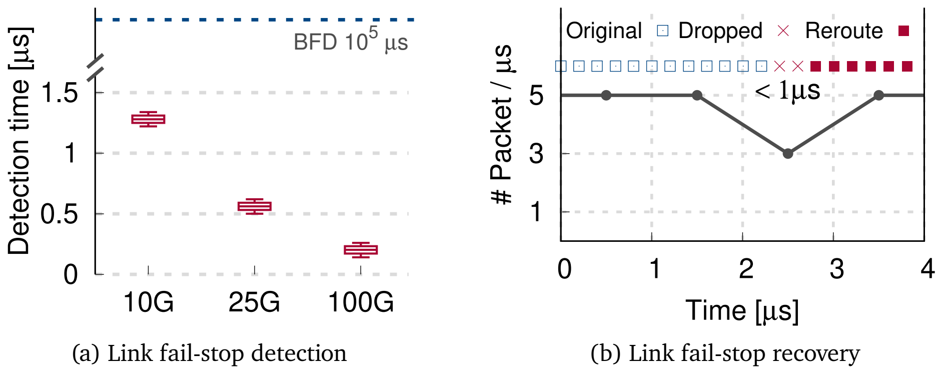
Table 2.1 shows the data plane memory consumption of both systems. Additional memory increases $P(\text{notified})$, however the relationship is different for each system. As a concrete data point, consider the coverage goal highlighted in the original NetSeer evaluation [207]—to correlate 90% of packet loss events with flows. For a 64×100 Gbps switch and a similar estimation strategy as above, NetSeer-OW meets this goal with 320 KB of SRAM (128 cache slots per port) in both ISP and data center workloads. On the other hand, assuming the network's minimum packet size is 64 B, NetSeer requires approximately 384 KB of SRAM to meet the 90% coverage objective because it must allocate enough ring buffer slots per port (256) to ensure that sequence numbers are not overwritten before switches have a chance to correlate their results.

2.4.1.2 Evaluation

Detecting failures more quickly. To quantify how quickly NetSeer-OW can detect a failure, we deployed NetSeer-OW to a hardware testbed and randomly disconnected a link between the two switches A and B 100 times to emulate 100 fail-stop link failure events. To test the limits of our approach, we configured the probes to mark a τ-timeout failure as soon as even a single packet loss is detected.

	NetSeer		NetSeer-OW	
Data structure size (per-port)	**256**	**64**	**512**	**128**
SRAM (KB)	384	192	896	320
Number of sALU/register arrays	6	6	7	7

Table 2.1: Data plane resource usage for typical NetSeer and NetSeer-OW configurations on a 64×100 Gbps switch.



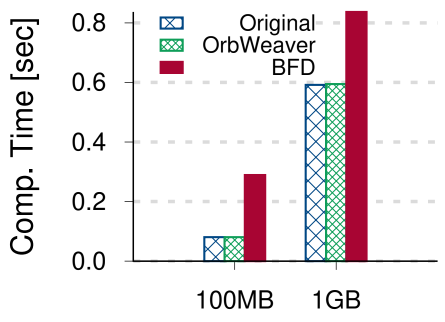
(a) Link fail-stop detection (b) Link fail-stop recovery

Figure 2.9: (a) the min, Q_1 (p25), median, Q_3 (p75), and max of OrbWeaver's time to detection across 100 failure events. (b) OrbWeaver's time to recovery (<1 µs) from a bidirectional failure of a 25 Gbps link. A total of two packets are lost.

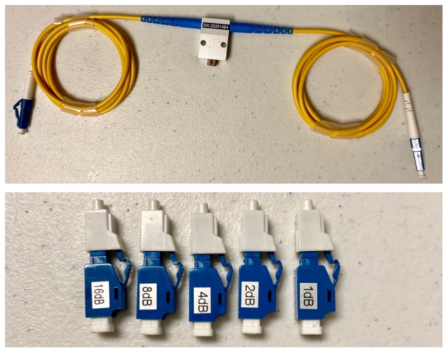
Figure 2.9a shows the detection time of trials for 10, 25, and 100 Gbps links. NetSeer-OW achieved 100% precision and recall. It also consistently detected the failure within 10s of nanoseconds of the optimal time. In contrast, typical configurations for protocols like Bidirectional Forwarding Detection (BFD) are closer to 10s or 100s of milliseconds; even recent data plane detection systems [124, 85] are several orders of magnitude slower than NetSeer-OW can achieve.

Figure 2.9b shows the resulting seamless recovery when NetSeer-OW is combined with a simple data plane rerouting mechanism. In the experiment, we induce a bidirectional failure in one link between A and B, and we configure B to failover to a backup path as soon as it detects an error. On top of this setup, we send a steady stream of packets on the target link at a relatively high rate of 5M packets per second. A total of two packets were lost—likely in-flight.

End-to-end impact. To evaluate the end-to-end impact, we emulate a leaf-spine topology with 2 leaf switches L1, L2 and 2 spine switches S1, S2. All switches run OrbWeaver with pre-computed



(a) Completion time during failures



(b) Optical attenuators

Figure 2.10: (a) shows the transfer completion time comparison for original, NetSeer-OW, and BFD (100 ms) in a simple leaf spine topology. With NetSeer-OW's fast detection and data plane reroutes, the impact is minimal.

data plane backup paths. Between L1 and L2, we insert a variable fiber optic in-line attenuator capable of 0∼60 dB attenuation. On hosts connected to the leaf switches, we run TCP transfers of varying sizes using `iperf`, during which we increase attenuation from zero until failure and examine the impact over the transfers experiencing the events. As Figure 2.10a shows, with OrbWeaver, the impact of failure is negligible with respect to completion time. In contrast, with BFD, failures cause the 100MB transfers to take over 4× longer and the 1GB transfers to take over 30% longer.

2.4.2. Time Synchronization

Time synchronization is another common task in modern networks. Like failure detection, time synchronization requires coordination between adjacent switches, and many other applications rely on its accuracy [188, 60, 190, 150].

Unfortunately, the most common methods for synchronizing time between adjacent machines involve the computation of One-Way Delay (OWD) using periodic, high-priority echo requests/replies [71, 124, 3]. Here too, architects are presented with a tradeoff: clock frequency drifts imply that the faster we send echoes, the more closely we can bound the clock offset and the more accurate the synchronization. Protocols like DTP [117] that integrate the protocol into the physical layer can circumvent this overhead but require hardware changes.

2.4.2.1 An OrbWeaver Redesign

The state-of-the-art in time synchronization for programmable switches is DPTP [103]. In it, two adjacent switches (a client, A, and a server, B) compute the offset of their local clocks by leveraging switches' ability to embed timestamps into each packet during different stages of packet processing. Host and multi-hop synchronization are also possible using multiple strata. The protocol calls for three messages in each round of the protocol: (1) a DPTP request [$A \to B$], (2) a DPTP response [$B \to A$], and (3) a DPTP follow-up [$B \to A$]. All three messages are high-priority to eliminate queuing delay.

(1) is timestamped using the Tofino `egress_deparser_tstamp` and `ingress_mac_tstamp` of A (t_1) and B (t_2), respectively. (2) is timestamped using the same counters in B (t_3) and A (t_4), respectively. In a traditional clock synchronization protocol, the offset would be computed as $\frac{(t_2+t_3)-(t_1+t_4)}{2}$. Unfortunately, we note a fundamental limitation of today's programmable switches—that the `egress_deparser_tstamp` does not capture the actual point of packet serialization. Thus, the computed offset is subject to variable delays as a result of egress MAC contention. As a result, DPTP introduces the third packet, the follow-up, which embeds a more accurate egress serialization timestamp (obtained out-of-band). Again, we refer interested readers to [103] for full details.

An OrbWeaver-inspired redesign can obviate the need for the third, follow-up message by inferring the egress MAC contention from the weaved stream (and only using results with no contention). This allows us to use the traditional two-way protocol of Figure 2.11. It can also eliminate the impact of the remaining messages using opportunistic sends.

Opportunistic synchronization: Rather than relying on high-priority echoes, a system can rely solely on OrbWeaver's IDLE packets to piggyback timestamps. In particular, whenever A has an opportunity, it sends a request to B on an IDLE packet with a field for t_1. Upon receiving the packet, B maintains a cache for the most recent values of t_1 and t_2. Separately, whenever B has an opportunity, it sends the most recent values of t_1 and t_2 along with the local `egress_deparser_tstamp` in t_3. In an empty network, A can calculate the clock skew as $\frac{(t_2+t_3)-(t_1+t_4)}{2}$ just as DPTP but with much

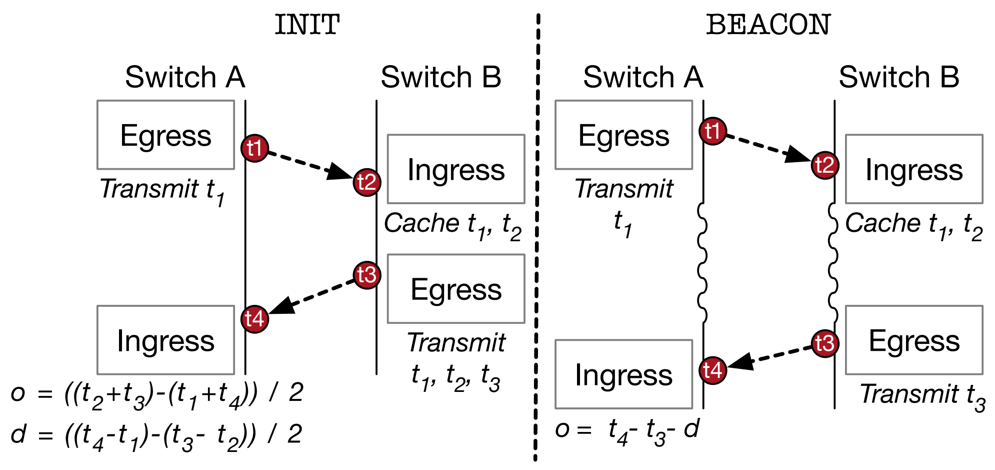


Figure 2.11: Time sync in DPTP-OW, using IDLE packets. When the difference between t_2 and t_3 is small, A treats the message as part of an INIT phase and calculates o, the clock offset, and d, the one way delay. When it is high, the BEACON phase uses the most recent d to track clock frequency drift.

more frequent synchronization (leading to lower *jitter*, i.e., nominal error [5]).

A challenge with the above approach occurs in networks with high utilization. The traditional OWD estimation method used above implicitly assumes that the clock drift is constant for the duration of the protocol round; otherwise, the delays at the time of the request and response may not be comparable due to clock frequency drift. In OrbWeaver, this can happen if there is congestion from B to A; the gap between t_2 and t_3 can be unbounded, leading to inaccurate results.

We address this challenge by borrowing an idea from a different protocol, DTP [117]: the decoupling of synchronization into INIT and BEACON rounds. If the time between t_2 and t_3 is sufficiently small, the round is treated as an INIT round and A computes the offset as above. Otherwise, A treats the message as part of a BEACON round where it takes d, the OWD computed from the last INIT round ($d = \frac{(t_4-t_1)-(t_3-t_2)}{2}$), and it computes a new offset: $o' = t'_4 - t'_3 - d$.

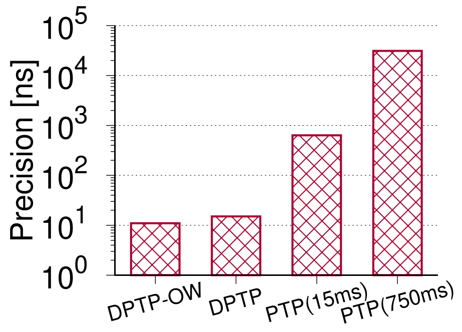
Selective synchronization: Finally, to remove the need for DPTP's third 'follow-up' message, we can exploit the implicit information contained in the woven stream's timing. The underlying intuition is simple: if the gap between an IDLE packet and its preceding packet is less than τ, the IDLE packet may have encountered contention at the egress MAC. In this case, the packet's timestamp may be unreliable. DPTP corrects for this contention with the follow-up message; OrbWeaver simply ignores these protocol rounds. While this filtering effectively requires that usable gaps be $> \tau \sim 2\tau$, it greatly improves the accuracy of the protocol while still permitting frequent re-synchronization in modern networks.

Dealing with a lack of opportunity to send. The above protocol fully synchronizes switches when both links have concurrent IDLE gaps. The protocol also includes support for correcting small drifts when only one direction has a gap (by adjusting to the fastest clock in the network). We note that in a network with multiple paths, we can configure synchronization to propagate among any one of those paths. Thus, if we view the network as a directed graph, the only time a switch may lose synchronization is if sufficient links are maintaining 100% utilization that the links form a cut of the graph. In the end, if operators need assurances, they may need to send higher-priority messages if too much time elapses; however, we can extend our techniques so that the messages only need to be prioritized above the lowest-priority user traffic—high-priority, interactive applications would be unaffected.

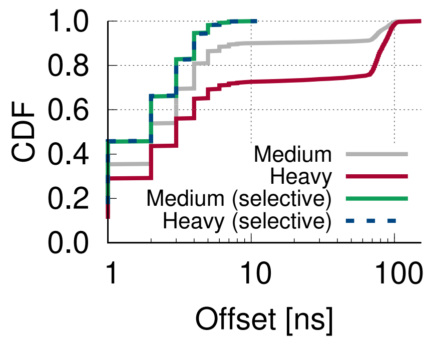
Benefits. As long as there is occasional usable bandwidth in the network, OrbWeaver again eliminates all bandwidth overheads without sacrificing accuracy or nominal error. When the network is underutilized, it actually provides similar re-sync intervals as DTP but using commodity PISA switches.

2.4.2.2 Evaluation

Following prior work, we evaluate DPTP-OW's precision [117, 180] (defined as the maximum clock skew in the network), as well as its jitter [5] (defined as the distribution of measured offsets or nominal error). Again to match prior work, we evaluate these in a two-switch testbed during a 20 min collection for 10 Gbps link with a medium workload (a CAIDA trace with 25% average utilization) and a heavy workload (the same trace sped up to ~80% average utilization). We compare to both DPTP (with 2000 requests/sec) and PTP. For PTP, prior work has suggested message frequencies ranging from 15 ms to 2 s [3, 124, 27, 117]; we pick two points in this range: 15 ms as a lower



(a) Clock precision



(b) Clock jitter

Figure 2.12: (a) shows the precision for different synchronization protocols and a heavy workload (∼80% CAIDA user traffic). (b) shows the CDF of observed offsets (absolute value) for DPTP-OW upon medium and heavy loads for 10 Gbps link ($\tau = \mathbf{1200}$ ns), w/ or w/o selective sync. OrbWeaver achieves a precision of 11 ns even under heavy user traffic.

bound and 750 ms per the evaluation baseline [117].

We observe that, even at high loads, DPTP-OW can achieve 10 ns bounds in both precision (Figure 2.12a) and jitter (Figure 2.12b) without imposing on user traffic. These bounds are similar to or better than DPTP, which incurs high-priority bandwidth overhead. Preliminary tests on higher-link speeds indicate that precision will only improve as τ decreases. In Figure 2.12b, we further observe that selective synchronization is an effective technique to reduce the message complexity of the protocol while maintaining low jitter and good precision.

2.4.3. Congestion Feedback

Finally, many modern networks rely on robust load balancing algorithms to efficiently utilize their multiple paths. There are numerous approaches to load balancing, but among them, adaptive approaches [36, 105] are attractive as they can react to current network conditions when making balancing decisions.

A state-of-the-art approach is taken by HULA [105], which proposes a protocol for adaptive data center load balancing using programmable switches. In HULA, every switch maintains two tables: a `bestHop` table that stores the best next-hop to each destination ToR, and a `pathUtil` table that stores the utilizations of those next-hops. Destination ToRs periodically flood the network with high-priority probes that traverse all paths (in the reverse direction, dst-to-src) and track the bottleneck link utilization of the best such path—intermediate switches update their `bestHop/pathUtil` tables accordingly.

As in the previous use cases, congestion feedback mechanisms like the one in HULA force a tradeoff between overhead and the availability/freshness of congestion data. HULA eventually sets the probing interval to 1-RTT and makes a case for why that is a good tradeoff, but OrbWeaver can potentially provide similar performance using only opportunistic sends.

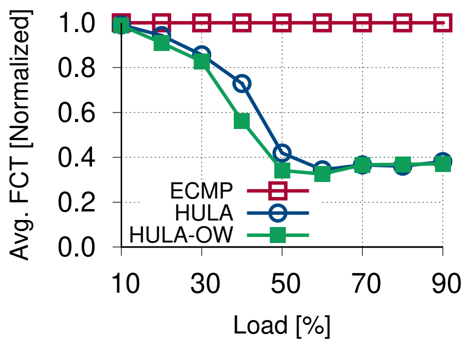
2.4.3.1 An OrbWeaver Redesign

An OrbWeaver-inspired redesign replaces the high-priority HULA probes with OrbWeaver's opportunistic IDLE packets. There are two new challenges. The first is building a 'flood' communication model on top of OrbWeaver's opportunistic sends. The second is dealing with congestion on the reverse path and the resulting lack of new information.

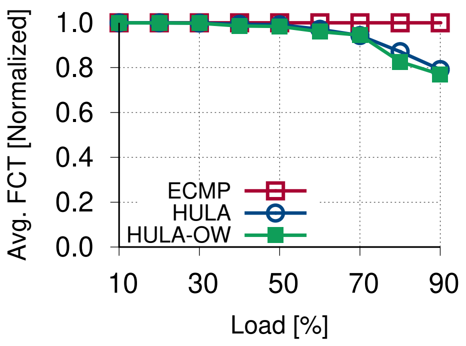
Per-path propagation: For any path through the network, there are two types of hops: ingress-to-egress hops (that bridge the pipelines of a local switch) and egress-to-ingress hops (that bridge adjacent switches).

For the former, HULA-OW leverages the switching ASIC's PCIe interface to asynchronously mirror the `pathUtil` table between the ingress and egress pipelines of a single switch. We use Mantis [193] to mirror the registers, which completes a mirror operation every ∼20 µs without impacting data plane throughput. For the latter type of hop, the system simply sends the contents of `pathUtil` using IDLE packets. To make this process more efficient, we can stripe the `pathUtil` table across m registers and pack m `(dstToR, pathUtil)` records into each IDLE packet round-robin style. In an unloaded network, the full table is transmitted in $\frac{R\tau}{m}$ time, where R is the number of ToRs in the data center. We note that even for $R = 1000$ and $m = 1$ (i.e., an unoptimized update rate), this is still more frequent than HULA.

Stale information: If there is persistent congestion on the reverse path, utilization information may not be able to propagate across the network; the switch adjacent to the congestion will know the



(a) Web Search



(b) Data Mining

Figure 2.13: Avg. FCT (normalized to ECMP) for HULA and HULA-OW upon different loads of DCTCP and VL2 traces.

utilization of the adjacent link, but not downstream links. To handle this case, HULA-OW uses a simple aging mechanism. Specifically, it will track the EWMA of all observed `pathUtil` values for every destination ToR (in addition to the minimum). After each RTT with no information from the best path, it will shift the best path's `pathUtil` value toward the average (with a lower bound of the adjacent link's utilization). If no information comes from *any* neighbor for several RTT and the adjacent links are all equal, the switch will fall back to random flowlet placement.

Dealing with a lack of opportunity to send. We note that the effect of the above metric-aging strategy is that `bestHop` will be quickly overwritten by the 'next-best hop' whose reverse path has opportunities to send. Assuming that at least some congestion information gets through, HULA-OW will still provide substantial benefits due to properties like the power of two choices [139]. In the worst case, it achieves equivalent performance to flowlet ECMP.

Benefits. Across all regimes, HULA-OW eliminates the probe overhead on network bandwidth. In networks with low utilization or high burstiness, it provides more frequent utilization updates than HULA in addition to increasing the peak usable bandwidth (see below).

2.4.3.2 Evaluation

Performance. We evaluate HULA-OW in NS-2 using the same FatTree topology as the original paper (Figure 4 of [105]). Also like HULA, we leverage synthetic workloads based on web-search [34] and data-mining [74]) and configure HULA to probe at a 200 µs interval. Figure 2.13 shows the avg. FCT (normalized to ECMP) for HULA and HULA-OW.

Despite the frequent periods of full utilization in these workloads (especially at high average load), we observe that HULA-OW is able to find sufficient gaps between packets to efficiently transfer utilization information. Overall, HULA-OW is able to provide comparable or better performance than HULA in all of the tested cases, even in the presence of very high average utilization. The performance is also always either equivalent or better than the ECMP baseline.

Overhead reduction. The bandwidth overhead of HULA probes is given by [105]:

$$\frac{probeSize \times numToRs \times 100}{probeFreq \times linkBandwidth} \tag{2.2}$$

With 500 ToRs, probeFreq=200 µs, probeSize=64 B, and 100 Gbps links, HULA occupies 1.6% of the network's bandwidth. In contrast, HULA-OW occupies close to *zero* of the network's usable bandwidth and only 1.5% of the packet-level capacity of the ingress pipeline (which HULA's probes also consume).

2.5. Related Work

Leveraging unused resources. OrbWeaver is not the first system to propose the opportunistic use of leftover resources. Indeed, many applications of priorities are in a similar spirit. Even in contexts outside of computer networking, others have used low-priority background tasks and spot VMs to harvest unused CPU cycles and memory [37].

In networking, close related work includes software WANs like SWAN [86] and B4 [94], which divide traffic into classes that range from interactive to background—interactive traffic is given priority while background traffic soaks up any remaining bandwidth. These systems successfully provide opportunistic bandwidth utilization but focus on end-host data. As explained in §2.2, these approaches can leave parts of the network unutilized due to both application traffic patterns and structural bottlenecks. OrbWeaver is, thus, complementary to these approaches and can be used to reclaim the remaining bandwidth for intra-network coordination.

Prior work has also applied similar techniques to lower layers, for instance, in the case of Ethernet's IDLE symbols or F10's rapid heartbeats [129]. F10, in particular, proposed a failure detection mechanism that is close to OrbWeaver's in which devices continue to send traffic even when idle. In comparison, OrbWeaver's contribution is make the idea practical on high-speed programmable switches, to closely examine the resulting impacts on switch configurations and user traffic, and to show how to seamlessly integrate the weaved stream into a spectrum of applications beyond the use case of F10.

Applications of OrbWeaver. OrbWeaver also builds explicitly on prior work that improves networks with coordination, signaling, and probes. We refer readers to the relevant parts of §2.4 for a discussion of the systems on which OrbWeaver builds, and to the original papers for a more complete examination of related work for our applications.

In general, however, OrbWeaver improves on much of the prior work by providing comparable or better performance with near-zero overhead. Exceptions include systems like F10 [129] and DTP [117], which use hardware support to eliminate protocol overheads. As mentioned above, OrbWeaver's contribution is to generalize the concept and demonstrate a practical framework for it on commodity network devices.

2.6. Conclusion

Must data plane applications always choose between coordination fidelity and bandwidth overhead? OrbWeaver demonstrates that, somewhat surprisingly, they do not. To that end, we introduce OrbWeaver, a framework for opportunistic coordination in a manner that does not affect user traffic or switch power consumption. Using three recently proposed systems, we show how to leverage OrbWeaver to eliminate their bandwidth overheads while maintaining their efficacy.

CHAPTER 3

RECYCLING SWITCH RESOURCES FOR FLEXIBLE, SUB-RTT REACTIONS

Simplicity is the ultimate sophistication.

Leonardo da Vinci

This chapter was previously published in press as *Liangcheng Yu, John Sonchack, and Vincent Liu. Mantis: Reactive programmable switches. In Proceedings of the ACM SIGCOMM 2020 Conference (SIGCOMM 2020)*. The dissertation author led all phases of the project, from idea development to system prototyping and writing.

Abstract. For modern data center switches, the ability to—with minimum latency and maximum flexibility—*react* to current network conditions is important for managing increasingly dynamic networks. The traditional approach to implementing this type of behavior is through a control plane that is orders of magnitude slower than the speed at which typical data center congestion events occur. More recent alternatives like programmable switches can remember statistics about passing traffic and adjust behavior accordingly, but unfortunately, their capabilities severely limit what can be done.

In this chapter, we present Mantis, a framework for implementing fine-grained reactive behavior on today's programmable switches with the help of a specialized reactive control plane architecture. Mantis is, thus, a combination of language for specifying dynamic components of packet processing and an optimized, general, and safe control loop for implementing them. Mantis provides a simple-to-reason-about set of abstractions for users, and the Mantis control plane can react to changes in the network in 10s of μs.

3.1. Introduction

Modern data center networks are becoming increasingly dynamic. Their switches, in addition to providing simple forwarding functionality, are often expected to change their packet processing behavior in reaction to fluctuating network conditions, e.g., to distribute traffic [35, 72, 182, 93], handle failures [128, 126], implement flow control [6, 140, 82], or apply expressive security policies [171, 102, 119]. For each of these tasks, reacting to the current state of the network is critical to maintaining strict Service-Level Objectives (SLOs).

Parallel to this trend has been a realization that the majority of congestion events in today's data centers are microscopic in duration. For instance, [203] found that, in one production data center, 90% of continuous periods of high utilization lasted for less than 200 µs. Studies of other data centers have shown similar levels of volatility in network traffic over small timescales [48, 155].

As a result, recent work has proposed pushing an increasing amount of adaptive behavior into the network devices themselves. Load balancing is one such example. While switches have long been able to statically spread load over the network using mechanisms like ECMP, microbursts and other transient events provide a compelling case for making more complex routing decisions locally at each device, where it is possible to react at very small timescales [35, 72, 6]. This is in contrast to more traditional OpenFlow-style approaches, which rely on a relatively slow control loop passing through a centralized controller. Similarly reactive systems have been proposed for other use cases [128, 102, 119, 186].

The cost of faster reaction time in these systems? For many, it is custom hardware modifications that add the features directly into the data plane [72, 128, 35]. Unfortunately, developing these custom ASICs is both extremely expensive and time-consuming. Programmable switches provide a promising alternative, allowing users to integrate some amount of statistics gathering and computation into the packet processing pipeline, but as we describe in §3.2, the limitations of today's programmable switches are well known and difficult to overcome, despite sustained efforts from the networking community.

In this chapter, we present Mantis, a framework for implementing fine-grained reactive behavior in today's programmable switches. Like traditional network architectures, Mantis relies on the data plane to perform packet processing and the control plane to implement arbitrary control logic. Un-

```
malleable value value_var { width : 16; init : 1; }
malleable field field_var {
    width : 32; init : hdr.foo;
    alts {hdr.foo, hdr.bar}
}
malleable table table_var {
    reads { ${field_var} : exact; }
    actions { my_action; drop; }
}
action my_action() {
    add(${field_var}, hdr.baz, ${value_var});
}
reaction my_reaction(reg qdepths[1:10]) {
    uint16_t current_max = 0, max_port = 0;
    for (int i = 1; i <= 10; ++i)
        if (qdepths[i] > current_max) {
            current_max = qdepths[i]; max_port = i;
        }
    ${value_var} = max_port;
}
```

Figure 3.1: An example P4R code snippet with fields, values, and a table that can be modified at runtime using fine-grained reactions. `malleable` variables are annotated as such. Malleable field and value variables are referenced as `${var}`.

like traditional architectures, the Mantis control plane is designed to—at the granularity of 10s of µs—continually measure and adjust the behavior of the data plane. Mantis is, thus, a combination of (1) an extension to the P4 language, P4R, that helps to specify which parts of the data plane should be malleable, and (2) the Mantis agent, an optimized control plane that provides both a Turing-complete substrate and serializability guarantees for user-defined reactions. While Mantis's reliance on the CPU means that it cannot react to *every* packet, it enables sub-RTT reaction time, which we show is sufficient for many applications.

Figure 3.1 shows an example P4R program. In it, we can observe a set of novel primitives that can replace any of their traditional P4 counterparts: malleable *values*, which can be reconfigured at runtime to take on any numeric value; malleable *fields*, which can be reconfigured to reference P4 packet/metadata fields; and malleable *tables*, which function as normal match-action tables, but are augmented with support for fast and serializable updates. Mantis will continuously poll headers/state from the data plane and modify the above primitives based on user-specified *reactions*—each iteration of the reaction loop allows arbitrarily complex reaction logic, is guaranteed to operate on

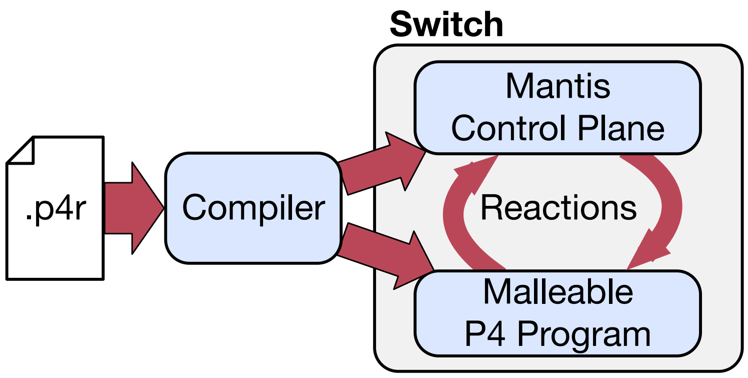


Figure 3.2: Mantis. A P4R program is compiled into a pair of artifacts that support high-frequency, switch-local reactions.

fresh data, permits concurrent traditional control plane operations, and provides serializable isolation with respect to reads, writes, and packet processing. Our prototype and evaluation of Mantis and the P4R language demonstrate their utility in a wide range of use cases that are difficult/expensive to implement otherwise.[7] This chapter makes the following contributions:

- We introduce a novel extension to the P4 language, P4R, that treats *reactions* to current network conditions as a first-class citizen. Along with this language, we present a Flex/Bison-based compiler that translates P4R into a pair of artifacts: (1) a valid but malleable P4-14 v1.0.5 program and (2) C reaction code that polls data plane state and updates its malleable portions.

- We also develop the Mantis agent, a control plane architecture that can execute reactions quickly and safely on a Wedge100BF-series switch. Depending on the reaction complexity, our current implementation can react at a granularity of 10s of µs (less than an RTT in most networks) and guarantees serializable isolation of both the measurements and updates.

- Finally, we present a series of use cases that demonstrate the utility of Mantis and dynamic reaction. A surprising result of our work is that, not only does Mantis outperform centralized approaches, it can often outperform pure data plane approaches along important metrics.

3.2. Background and Motivation

We begin this chapter by describing the architecture of today's Reconfigurable Match Table (RMT) switches with a focus on their capacity to react to current network conditions. To that end, RMT switches are based on the abstraction of a pipeline of match-action tables. For a given packet and table, the switch will index into the table using a subset of packets' fields and metadata, extracting an action that it will then apply to the current packet. Some switches may also include a small amount of SRAM for stateful processing. The 'reconfigurable' of RMT refers to the ability to change both the fields considered in the match as well as the action that is executed.

In this context, a *reaction* involves aggregating statistics (e.g., packet count or queue depth) from across packets and then using those statistics to influence the processing of subsequent packets (e.g., by redirecting a subset of them or tagging them with a computed value). In principle, an RMT switch with stateful SRAM can be configured to do both of the above actions—measurement and control—entirely within the data plane, and prior work has done exactly this for a subset of use cases [77, 162]. In practice, however, today's state-of-the-art RMT implementations suffer from a number of well-known limitations, some of which may be fundamental to efficient ASICs [46] and none of which are addressed by existing hardware. These include, but are not limited to constraints on the operations allowed in actions (e.g., no multiplication/division, limited branching, etc.), a fixed number of stages in the pipeline, restrictions of SRAM accesses to a single element/stage, and a disconnect between ingress/egress pipelines.

When encountering one of these limitations, prior work has tended to take one of a few different approaches. Some expend heroic efforts approximate the original algorithms in a way that fits the constraints [162, 187, 161]. Others assume novel hardware primitives that add the appropriate flexibility to the data plane [105, 88, 142, 163]. Some in the latter category might still be developed into the former; however, for a general approach that works on today's networks, workarounds typically involve one of the following.

[7]The open-source P4R compiler can be found at https://github.com/eniac/Mantis.

Resubmission and recirculation. The most direct way to circumvent the data plane limitations is to send traffic back through the packet processing pipeline multiple times and, if necessary, across pipelines [46, 163, 105, 88]. Theoretically, with enough recirculations, one can overcome limitations in both computational power (achieving Turing-completeness) and memory flexibility (acquiring access to any number of SRAM entries any number of times).

The primary drawback is that, each time a packet is recirculated, it potentially impacts other traffic. Recirculating every packet twice, for instance, drops usable throughput of the switch to 38%; three times reduces throughput to just 16% [184]. As modern switches are generally limited by their packet-level bandwidth, the size of the recirculated packets is immaterial. Recirculated packet processing also introduces the potential for violations of consistency/isolation.

Chaining/reloading the data plane. For cases where stage count is the main limitation, an alternative approach is to chain or swap in/out network functionality as needed [107, 186]. Unfortunately, chaining imposes requirements on the network topology and workload, and installing a new data plane program renders the switch temporarily unavailable (several seconds in current implementations). It, therefore, only applies for coarse-grained reactions and cases where sufficient capacity exists elsewhere in the network.

Control plane assistance. Finally, we note that data planes have long been unable to implement all of the functionality needed inside a network. Instead, they are typically augmented with a control plane an onboard CPU to which the data plane can offload more complex tasks such as routing and management when those types of packets arrive. The two planes communicate by passing messages or, from the control plane, by polling counters and updating table entries—all of these can be done without disrupting normal switch operations. Unfortunately, traditional control planes assume that accesses are not time-critical and, thus, are generally orders of magnitude slower than the duration of network events [128, 72].

3.3. Mantis Design Overview

Mantis is a framework for fast, expressive in-network reactions on today's RMT switches. Mantis has two primary goals:

1. To enable general and flexible dynamic reactions that surpass the limitations of today's switches—measuring an arbitrary set of data plane metrics, computing an arbitrary set of statistics over that data, and manipulating the data plane without impacting normal traffic. To sustain typical network volatility, this process should be able to operate on a sub-RTT granularity.

2. To package the above capability into a reaction abstraction that hides the many complexities of implementing reactive behavior, e.g., ensuring that the data plane is malleable, coordinating data-control plane communication at runtime, and reasoning about asynchronous behavior.

Explicitly *not* a goal of our system is support for arbitrary changes to the data plane at runtime. For that, we refer interested readers to prior work that has successfully emulated P4 using match-action tables [80], albeit at a high cost (for an L2 switch, a $6.5\times$ increase in match stages and 83% bandwidth penalty). Instead, we assume that the general structure of the data plane is known a priori and that reactions only need to touch a subset of data plane objects.

We demonstrate, using Mantis, that the above approach is sufficient to implement a range of network architectures that are difficult and/or costly to implement in today's RMT switches. We discuss and evaluate these applications, listed in Table 3.1, in §3.8.

Core abstractions. Two abstractions underlie our work:

Malleable Entities — In Mantis, specific primitives in the data plane program can be tagged as 'malleable,' indicating that they should be amenable to fine-grained modification at runtime. Malleable values, used in the expressions of data-plane actions, can take on any constant value; malleable fields act as dynamic references to a predefined set of existing header and metadata fields; and malleable tables function exactly like normal match-action tables, but with the ability to be modified at a fine-granularity.

```
<p4_declaration> ::= <mbl_declaration> | <reaction_declaration> | ...
% (malleable entities)
<mbl_declaration> ::= 'malleable' <table_declaration>
    | 'malleable' <mbl_val_declaration>
    | 'malleable' <mbl_fld_declaration>
<mbl_val_declaration> ::= 'value' <mbl_name> '{'
    <width_declaration> ';'
    'init :' | <const_value> ';' '}'
<mbl_fld_declaration> ::= 'field' <mbl_name> '{'
    <width_declaration> ';'
    'init :' <field_ref> ';'
    'alts {' <field_ref> [',' <field_ref> ]* '}' ';' '}'
% (reactions)
<reaction_declaration> ::= 'reaction' <reaction_name>
    '(' [ <reaction_args> [, <reaction_args>]* ] ')'
    '{ // C-like code }'
<reaction_args> ::=  'ing' <reaction_arg>
    | 'egr' <reaction_arg>
    | 'reg' <register_ref> '[' <const_value> ':' <const_value> ']'
<reaction_arg> ::= '${' <mbl_read_ref> '}'
    | <field_ref>
    | <header_ref>
    | <field_value>
% (references)
<field_or_masked_ref> ::= '${' <mbl_read_ref> '}'
    | '${' <mbl_read_ref> '}' 'mask' | <const_value>
    | ...
<arg> ::= '${' <mbl_read_ref> '}' | ...
<exp> ::= '${' <mbl_read_ref> '}' | ...
```

Figure 3.3: The P4R extensions to the P4-14 v1.0.5 grammar. Gray non-terminal nodes refer to legacy rules in [9], and nodes ending in *_name* indicate strings whose first character is a letter. *mbl_read_ref*s can access both malleable values and fields. Note that all writes in P4-14 are done via primitive actions, which we omit for similar reasons to [9].

Reactions — Measuring the network and modifying malleable portions of the data plane are 'reactions'—small C functions that are compiled and dynamically loaded into a custom, reaction-centric control plane running on each switch's local CPU. In Mantis, the control plane will, as quickly as possible, poll the parameters of each reaction function and react to the measurements by updating malleable entities according to these user-defined functions.

System architecture. On top of the above abstractions, Mantis combines a language, a compiler, and a control plane architecture, all designed to enable fast, simple, and safe control loops for programmable switches. Figure 3.2 depicts the architecture of Mantis.

- *The P4R language:* Actualizing our two core abstractions is a simple extension to P4—one where certain fields, values, and tables can be tagged as 'malleable.'

- *The Mantis compiler:* The compiler transforms P4R programs into a pair of artifacts: (1) a valid P4 program that reformulates the P4R to ensure that metrics are exported and that `malleables` are runtime-configurable and (2) a reaction function implementation that interfaces with the generated P4 program.

- *The Mantis control plane:* An optimized control plane agent running on the switch CPU is responsible for the rapid, serializable coordination of measurement and updates.

Paper roadmap. We start in §3.4 by describing the P4R language and how Mantis translates from P4R to P4 *without* any isolation guarantees. §3.5 then introduces Mantis's approach for guaranteeing per-pipeline serializable isolation of reads, writes, and packet processing. Finally, §3.6 presents the Mantis control plane before delving into the implementation/evaluation.

3.4. Language and Transformations

Adhering to best practices in language design [104], P4R reuses the basic syntax and semantics of the P4 programming language. It then allows users to tag various P4 objects as 'malleable' and define the reaction functions that modify those objects. We already saw an example of both language features in Figure 3.1.

Grammar. Briefly, malleable fields and values are declared with a width, an initial value, and in the case of a malleable field, a set of potential aliases to which the entity can reference. Malleable tables are declared with an annotation to indicate that the compiler should prepare for its use in a reaction loop. Otherwise, all three can be used in the same way as their traditional P4 counterparts.

The precise extensions we make to the P4-14 grammar are shown in Figure 3.3. The grammar follows the naming conventions of [9]. Note that, like [9], we omit our changes to primitive actions such as `modify_field` and `add_to_field`, whose existence is platform dependent. In general, however, any field or value parameter to these primitive actions may be replaced with a reference to a `malleable` (*mbl_read_ref* or *mbl_write_ref*, depending on the semantics).

Reaction functions. Of note are reaction functions like the one in Figure 3.1 that allow users to embed C code that specifies the control plane behavior that accompanies the data plane implementation.

Syntactically, reactions mirror C, but with a couple of changes. The first is the parameters to a reaction, which are a set of fields, registers, or malleable fields/values from the data plane. Before executing the body of the reaction, Mantis polls the current value of all of these parameters. Note that if there are multiple line cards with distinct register state, a separate instance of the Mantis agent will run for each. The second is the use of `malleables` within the function body. For malleable fields and values, these can be referenced with the same `${var}` notation used in the rest of the P4R program—the compiler will replace them with generated functions that write to the data plane or read the last written value, depending on the context. For malleable tables, users can interact directly via a set of automatically generated library functions, e.g., `table_var.addEntry(...)`.

Semantically, all registered reaction functions are executed sequentially, in a loop. Mantis does not guarantee a specific ordering but does guarantee serializability between parameter polling, updates to malleable entities, and packets' processing (see §3.5).

3.4.1. Producing Malleable P4

Supporting fast and safe reactions is Mantis's compiler, which transforms P4R into a valid P4 program, but one in which malleable entities can be rapidly updated at runtime. In this section, we describe through several examples the necessary transformations for malleable fields and values without considering isolation guarantees. Note that we do not describe one-off writes of malleable tables as (ignoring isolation) they are already modifiable in today's switches.

Values. Figure 3.4 shows a simple example of a definition and use of a malleable value, `value_var`. The original P4R code can be found in the four *non*-bolded lines, which contain the entity definition and its use within the P4 `add` primitive.

The Mantis compiler transforms the code as follows. It instantiates the value in a metadata header (`p4r_meta_`) and generates an associated table (`p4r_init_`) with a single possible action. This table is applied at the beginning of each packet processing pipeline, and it is what allows Mantis to assign

```
- ~~malleable value value_var { width : 16; init : 1; }~~
+ header_type p4r_meta_t_ {
+     fields { value_var : 16; }
+ }
+ metadata p4r_meta_t_ p4r_meta_ { value_var : 1; };

  // Applied once at the beginning of the pipeline
+ table p4r_init_ {
+     actions { p4r_init_action_; }
+     size : 1;
+ }
+ action p4r_init_action_(value_var) {
+     modify_field(p4r_meta_.value_var, value_var);
+ }
  action my_action() {
±     add(hdr.foo, hdr.bar, ~~${value_var}~~ p4r_meta_.value_var);
  }
```

Figure 3.4: Mantis's transformation of a malleable value. ~~Strikethroughs~~ and '-' annotations indicate P4R code that is removed by the transformation; **bold** text and '+' annotations indicate P4 code that is generated by Mantis.

different values to the malleable at runtime by updating just a single table entry. As we will see later, this initialization table serves many purposes, configuring malleables and version control bits for the entire pipeline.

Fields (write). P4R also includes malleable fields, which act as references to a predefined set of existing P4 fields; users can dynamically 'shift' the target of the reference to any member of the set, e.g., to change the matched field of a table. As references, malleable fields are L-values, meaning that they can appear on either the left- or right-hand side of an assignment operator in the data plane program. We focus first on 'left-hand' usages. Figure 3.5 shows an example. Specifically, it shows a scenario where the programmer seeks to store the value of baz into either hdr.foo or hdr.bar.

A naïve implementation of this functionality would be to replace the malleable with a generated metadata field and, after every use of it, add a match-action table whose sole purpose is to copy the current value of the generated field back into the referenced field. The inserted table would have a distinct action for every possible 'alt,' and users would modify the default action when changing the target of the reference. Unfortunately, there are several issues with this strawman. First, it adds additional tables and potentially stages to the data plane program. Second, it violates the

```
- ~~malleable field write_var {~~
-    ~~width : 32; init : hdr.foo;~~
-    ~~alts { hdr.foo, hdr.bar }~~
- }
+ header_type p4r_meta_t_ {
+     fields { write_var_alt : 1; }
+ }
+ metadata p4r_meta_t_ p4r_meta_;
  // Action applied once (with value loads)
+ action p4r_init_action_(write_var_alt) {
+     modify_field(p4r_meta_.write_var_alt, write_var_alt);
+ }
  // For every use of the malleable field
  table my_table {
      reads { hdr.qux : exact;
±             p4r_meta_.write_var_alt : exact; }
±     actions { my_action_hdr_foo_;
±               my_action_hdr_bar_; }
  }
± action my_action_hdr_foo_(baz) {
±    modify_field(~~${write_var}~~ hdr.foo, baz);
  }
+ action my_action_hdr_bar_(baz) {
+     modify_field(hdr.bar, baz);
+ }
```

Figure 3.5: Transformation for malleable fields that we wish to use on the 'left-hand side' of assignments.

atomicity of reference shifts as a concurrent shift might cause the reference to act as `hdr.foo` or `hdr.bar` in different actions applied to the same packet. Even without concurrent shifts, uses of both the `malleable` and the field to which it references in the same action can be problematic.

To address the above challenges, Mantis performs two tasks. The first is to declare and load, at the beginning of the pipeline and for every relevant malleable field, a metadata field (e.g., `write_var_alt`) with width $\lceil \log_2 |alts| \rceil$ that determines, at runtime, the alternative that it references. The second is to transform every table that assigns the malleable field to also match on `write_var_alt`.

This extra match field allows the data plane to call specialized action functions that are instantiated for each possible configuration of the malleable fields. While this strategy increases the number of entries in affected tables to:

$$\sum_{(m,a) \in Entries} \left(\prod_{v \in \text{mbls}(a)} |v_{alts}| \right) \tag{3.1}$$

```
- ~~malleable field read_var {~~
-    ~~width : 32; init : hdr.foo;~~
-    ~~alts { hdr.foo, hdr.bar }~~
- }
+ header_type p4r_meta_t_ {
+    fields { read_var_alt : 1; }
+ }
+ metadata p4r_meta_t_ p4r_meta_;
  // Action applied once (with value loads)
+ action p4r_init_action_(read_var_alt) {
+     modify_field(p4r_meta_.read_var_alt, read_var_alt);
+ }
  // For every use of the malleable field
  table my_table {
±    reads { ~~${read_var} : exact;~~
+            hdr.foo : ternary; hdr.bar : ternary;
±             p4r_meta_.read_var_alt : exact; }
±    actions { my_action_hdr_foo_;
±               my_action_hdr_bar_; }
  }
± action my_action_hdr_foo_() {
±    add(hdr.qux, hdr.baz, ~~${read_var}~~ hdr.foo);
  }
+ action my_action_hdr_bar_() {
+     add(hdr.qux, hdr.baz, hdr.bar);
+ }
```

Figure 3.6: Transformation for malleable fields that we wish to use on the 'right-hand side' of assignments. This example combines uses inside an action and a table match field.

it avoids the table/stage costs of the strawman (often the bottleneck in programmable switches) and the atomicity issues. We anticipate that the number of affected actions, the number of malleable fields per action, and the number of alternatives per malleable field will all be relatively small in most cases.

Fields (read). Malleable fields can also be used on the right-hand side of assignments almost anywhere a field can be referenced in P4, e.g., inside an action, as a table match field, or in a field_list. Figure 3.6 shows a couple of examples.

Inside an action, we can apply the previous method of loading the selector field at the beginning of the pipeline and specializing actions. Slightly more complex is the use of malleable fields in table matches. Here, the compiler, in addition to matching on `read_var_alt`, replaces the `malleable` match field with |`alts`| instantiated match fields. For example, when the user adds an entry for

`${read_var}` $= 0$, Mantis inserts two entries into `my_table`:

- (`foo`=0, `bar`=*, `read_var_alt`=0)
- (`foo`=*, `bar`=0, `read_var_alt`=1)

Note that this means `exact` matches on a malleable field need to become `ternary` to accommodate the wildcard; `ternary` and `lpm` matches can remain. Also note that using a `malleable` in a table match, on its own, does not necessitate action specialization—specialization is only necessary if it is used within the given action.

Compound usages. While the above examples all include only a single malleable entity, Mantis allows the use of multiple entities in the same program and the same tables/actions.

One place where multiple `malleables` would interact is the initialization at the beginning of the pipeline. To minimize the number of necessary tables and actions, we can reuse a single `init_action` for multiple `malleables` (field or value) by passing in multiple parameters and including multiple assignments in the action body. If the number or aggregate size of the parameters exceeds the limits of a single action, Mantis will create multiple `init` tables. In this case, minimizing the number of tables involves a bin packing problem. Mantis solves this with a simple greedy algorithm in which it sorts the parameters in order of decreasing size and finds the 'first fit'.

The other place where they might interact is in the tables and actions of the P4 program. For malleable values, their composition is trivial as any instance can be directly replaced with the designated metadata field regardless of context. For malleable fields, multiple uses of the same field—whether left-hand or right—can be coalesced; each action needs to be specialized at most one time. For uses of different fields, transformations are applied recursively. For example, two malleable fields used in the same action will require two stages of action specialization that will result in an enumeration of all possible permutation of alternatives. Note an optimization when the fields are read, but not written—loading values in prior stages may result in lower overhead than instantiating all permutations.

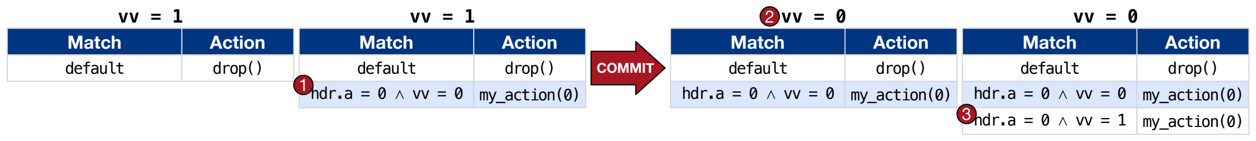


Figure 3.7: Ensuring sequentially consistency for table entry adds using three-phase updates. Multiple entries across multiple tables can be modified in step ❶ before they are atomically committed in step ❷. The rule is mirrored in step ❸ to assist with fast subsequent updates.

3.4.2. Gathering Measurements

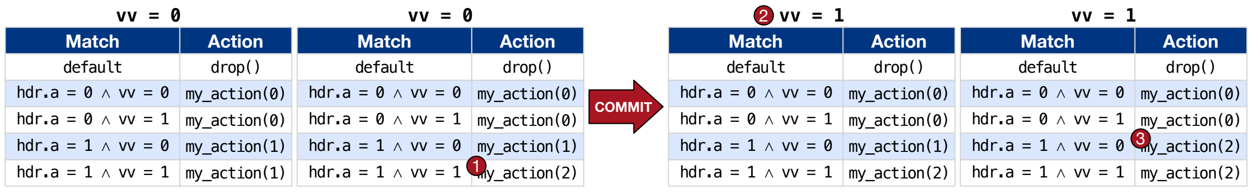
In addition to ensuring that portions of the data plane can be rapidly updated, Mantis also ensures that reaction function parameters can be rapidly read. While P4 provides many ways to read information from the data plane (e.g., digests, counters, the `copy_to_cpu` flag, etc.), to ensure fast reaction time, the chosen mechanisms need to have the following properties:

R1 Measurement should *not* be on a per-packet basis. While we seek fast reaction time, switch CPUs are not equipped and should not be expected to handle line-rate processing.

R2 The measurement schedule should be flexible. While reactions should be as close to real-time as possible, concurrent management tasks and varying reaction execution time mean that the Mantis should tolerate fluctuating measurement intervals.

R3 Measurements should return the most recent data. For instance, the regular export of digests from the data plane would be undesirable as the most recent digests might be head-of-line blocked behind previously unprocessed digests.

Mantis presents an abstraction where the data plane updates measurements on *every* packet, but the control plane only polls those measurements when it is ready to process the next iteration of the reaction loop. Mantis implements this protocol via stateful registers that can be updated in the data plane and polled from the control plane; Mantis stores the polled values in a C variable/array for use in the user-defined reactions. Note that, this pull-based model will only see a subset of updates, thus, users should ensure that any necessary information is retained across packets.



vv = 0

Match	Action
default	drop()
hdr.a = 0 ∧ vv = 0	my_action(0)
hdr.a = 0 ∧ vv = 1	my_action(0)
hdr.a = 1 ∧ vv = 0	my_action(1)
hdr.a = 1 ∧ vv = 1	my_action(1)

vv = 0

Match	Action
default	drop()
hdr.a = 0 ∧ vv = 0	my_action(0)
hdr.a = 0 ∧ vv = 1	my_action(0)
hdr.a = 1 ∧ vv = 0	my_action(1)
hdr.a = 1 ∧ vv = 1	my_action(2)

vv = 1

Match	Action
default	drop()
hdr.a = 0 ∧ vv = 0	my_action(0)
hdr.a = 0 ∧ vv = 1	my_action(0)
hdr.a = 1 ∧ vv = 0	my_action(1)
hdr.a = 1 ∧ vv = 1	my_action(2)

vv = 1

Match	Action
default	drop()
hdr.a = 0 ∧ vv = 0	my_action(0)
hdr.a = 0 ∧ vv = 1	my_action(0)
hdr.a = 1 ∧ vv = 0	my_action(2)
hdr.a = 1 ∧ vv = 1	my_action(2)

Figure 3.8: Ensuring sequentially consistency for table entry updates using three-phase updates. As in Figure 3.7, multiple entries and tables can be modified in step 1. Unaffected entries remain untouched.

Compiler transformations. Header/metadata reaction parameters are collected from every passing packet into generated registers at the end of the pipeline specified by their `ing/egr` annotation; the Mantis compiler places the register after the last modification to the field. User-defined register parameters can be read from the control plane directly, modulo the transformations described in §3.5.2.

Similar to the generation of the init action, Mantis packs header and metadata reaction parameters into as few registers as possible using the sorted-first-fit algorithm discussed previously. The only difference is that parameters from different reactions may be considered separately when packing. Although this may consume more resources than otherwise, it allows Mantis to poll only the most relevant parameters immediately before executing the reaction, which improves the freshness of the measured data.

3.5. Enforcing Isolation[8]

A critical piece of the reaction abstraction is isolation between measurement, modifications, and packet processing. To see why this is important, consider a reaction function that takes as arguments the 5-tuple from a packet. While a user might reasonably expect that the parameters passed into her reaction function all came from a single packet, without isolation, this is unlikely unless no new packets arrive between the first and last measurement.

To address this challenge, *Mantis provides per-pipeline, per-reaction serializable isolation between measurements, malleable entity updates, and packet processing*. Said differently, from the perspective

[8]'Isolation' here refers to same type of isolation used in ACID [201].

of a single packet processing pipeline, the three types of operations—gathering of measurements, application of a reaction, and processing of packets—all appear to execute in some sequential order, despite the inherent parallelism of packet processing.

This particular level of isolation is deliberate as it is both practical and efficient to implement. Stronger guarantees like grouping measurement and updates into a single transaction are useful but difficult to implement in today's switches. Similarly, cross-pipeline guarantees, while potentially useful, would require some type of in-band coordination between all pipelines [187]. We leave an exploration of these stronger models for future work.

3.5.1. Serializable Isolation of Updates

We begin with how Mantis guarantees serializability of reactions' effects before discussing measurement collection in §3.5.2.

3.5.1.1 Updates to fields and values

For malleable fields and values, the generated P4 of §3.4.1 is specifically crafted to be atomically modifiable. In particular, for both types of entities, their value is determined at the beginning of each pipeline, in the `p4r_init_` table. As RMT switches typically guarantee the consistency of a single table entry modification, so long as we can pack all configuration of malleable entities into a single `p4r_init_action_` (see §3.4.1), we can leverage the action as a serialization point. As a concrete example, consider a P4R program with two `malleables` (`value_var` and `field_var`):

```
action p4r_init_action_(value_var, field_var_alt) {
  modify_field(p4r_meta_.value_var, value_var);
  modify_field(p4r_meta_.field_var_alt, field_var_alt);
}
```

A single table entry update can change both atomically. New packets that enter the pipeline will use the updated assignments, while packets that have already passed this stage will continue to use the previous set of assignments.

The above strategy works until the P4R metadata used in a single pipeline of the P4R program exceeds the allowed size of the action (a platform-dependent value). As mentioned in §3.4.1, this case forces the compiler to split the `p4r_init_` table into several, e.g., `p4r_init1_`, `p4r_init2_`, etc. Updates to these tables can be made serializable by treating all except the first as normal, malleable tables and using the method described in the subsequent section.

3.5.1.2 Updates to tables

Handling malleable table modifications is slightly more complex. Conceptually, Mantis's approach is similar to that of [153, 154], but with a few critical differences that stem from Mantis's goal of extremely fast and repeated updates.

More specifically, in [154], Reitblatt et al. guarantee consistent updates in SDN deployments using a two-phase protocol. The protocol assumes that every packet is tagged with the current version number, i. Thus, to install a new configuration, the first step is to add the complete set of new rules across the internal nodes of the network such that the new rules only match on packets with version $i + 1$. The second step is then to, one-by-one, update all ingress nodes to tag packets entering the network with version $i + 1$. After a conservative timeout, the older configuration set is eventually removed from the internal nodes.

There are at least a couple of issues with applying the above protocol directly to Mantis's reaction loop. The first relates to the handling of frequent updates: given how often Mantis can update tables, conservative timeouts and the need to keep around multiple 'in-flight' updates can easily lead to order-of-magnitude increases in necessary table space. The second relates to reaction time: every update in [154] requires an insertion for every table entry in the new configuration, regardless of whether it was changed from the previous version or not.[9] Removal of stale versions doubles the latency overhead when the throughput of the control plane is the bottleneck.

Mantis, in contrast, guarantees serializability of groups of arbitrary and repeated table updates, where the required time is proportional to the number of P4R table interactions and the space overhead is bounded. Mantis's approach relies on a 1-bit version control flag, `vv`, that is set in the

[9]While [154] mentions possible optimizations that only apply the delta between old and new configurations, it does not discuss how to handle more than one such update.

init_action alongside the malleable fields/values; vv is also added as an exact-match field to every malleable table. With the vv field, every entry in every malleable table is doubled: one copy with $\texttt{vv} = 0$ and the other with $\texttt{vv} = 1$. Active entries can be flipped atomically by updating the vv bit.

Note that a 1-bit version flag is sufficient in Mantis as Mantis loads malleable entity configurations and the version control bit at the beginning of each pipeline and deliberately does not guarantee cross-pipeline isolation (e.g., between ingress and egress or across recirculations). Thus, old versions only persist for the maximum latency of a pipeline, which is typically measured in the 100s of nanoseconds. PCIe latency from the control plane is an order of magnitude higher, so the maximum number of active versions is two, regardless of the complexity of the reaction's effects.

Adding a new entry to the table at runtime employs a three-step update, as shown in Figure 3.7. Assume, w.l.o.g., that vv begins at 1. In this configuration, the entries with $\texttt{vv} = 1$ serve as the primary copy, while the entries with $\texttt{vv} = 0$ serve as a shadow copy.

1. The Mantis control plane first *prepares* the entries it wishes to add by adding them to the table, but with the requirement that $\texttt{vv} = 0$. Any number of entries and tables can be modified in this step; meanwhile, all packets will continue to use the $\texttt{vv} = 1$ and the default action.

2. In the second step, Mantis then *commits* all of the added entries by atomically flipping the version control bit, $\texttt{vv} = \texttt{vv} \oplus 1$ by updating the p4r_init_ table. Note that any inflight packets that have already passed the init stage will continue to use the $\texttt{vv} = 1$ copy even after the commit.

3. Finally, so that the entry can withstand a subsequent flip back to $\texttt{vv} = 1$, Mantis *mirrors* updates to the shadow copy. While this step has no visible effect on the network, it amortizes the cost of maintaining the shadow to keep latency more predictable.

Updating an existing table entry proceeds as in Figure 3.8. As mentioned above, init tables beyond the first are handled using the same mechanism. p4r_init1_ (the first table) is considered the 'master' and contains the version control bit; all other init tables will contain two entries (one for each version) just like a malleable table. Thus, when dealing with multiple init tables, the master

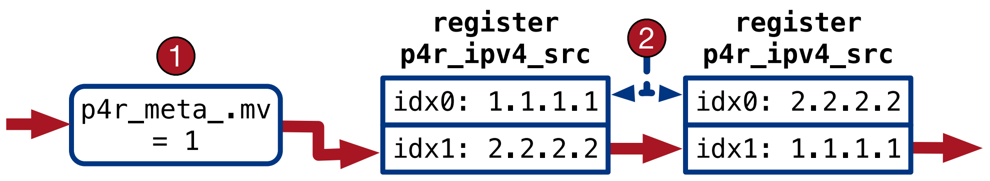


Figure 3.9: Ensuring measurement isolation for measured data plane fields. For `mv` $= \mathbf{1}$, index 1 is the working copy and index 0 is the checkpoint copy. For `mv` $= \mathbf{0}$, vice versa.

is always updated last. Deleting an entry looks similar to adding a table entry, but in reverse: the shadow copy is deleted in the prepare step and the original primary is deleted after the commit. All three types of operations—add, update, and remove—can share a prepare and commit step, even if they touch the same tables and entries. A proof of serializability of this process follows from that of [154].

3.5.2. Serializable Isolation of Measurements

Mantis also guarantees that the polling of the registers that store reaction parameters reflects a serial execution with respect to packet processing. As mentioned, a naïve implementation of register polling would result in an inconsistent view of reaction arguments.

Fields. Mantis ensures that the data plane will not overwrite the registers that store reaction-parameter fields (see §3.4.2) by using a register array rather than an individual register. These arrays have two entries each: a 'working' copy and a 'checkpoint' copy, both gated on a 1-bit `mv` bit that is set in the `p4r_init_` action of each pipeline along with `vv`. We configure the data plane such that it only writes to the working copy.

Figure 3.9 demonstrates an example usage of this mechanism. When the control plane wishes to measure a group of generated field-storing registers, it first ❶ flips the measurement version bit. Assuming that the flip was from $0 \to 1$, index 0 of both registers are now the checkpoint copies and should not be touched by the data plane. Mantis can, therefore, take as much time as it needs to ❷ read those values; meanwhile, the data plane will continue to update the working-copy entries.

Registers and register arrays. Mantis can also collect values contained in stateful elements, e.g., registers, using a ‘double-buffering’ scheme. Specifically, it creates a duplicate version of the register with twice as many instances. In every action that writes to the original user-defined register, Mantis saves the written value (and in the case of a register array, the accessed index) to metadata fields and mirrors the fields to the duplicated register. The written index is the original index prepended by the `mv` bit.

A complicating factor in register measurement (and the reason why we need a duplicate rather than reusing the original) is that not every register will be updated on every packet. In fact, in the case of a register array, at most a single index will be updated per packet. Because of this, the control plane may observe stale values. For example, consider a case where a register R contains the value r_i in both $\texttt{mv} = 0$ and $\texttt{mv} = 1$, and R gets updated to r_{i+1} in the working copy, $\texttt{mv} = 1$. If the control plane flips `mv` twice before another update of R, then it will observe r_{i+1} followed by a stale reading of r_i. Until a new update of R, the measured value will alternate between r_i and r_{i+1}.

The above effect necessitates an additional mechanism. Mantis adds to every duplicated register a ‘timestamp’ register whose entries are incremented every time the associated register’s entries are updated. This allows the Mantis control plane to identify which entries have changed since the copy was last read. The control plane keeps a cache of these values; entries are only replaced when the associated timestamp is updated, ensuring that it holds only the most up-to-date contents of every register entry.

We note a potential optimization when the stateful element is never read within the data plane—a common pattern with switch counters and other statistics. In this case, the original register is not necessary and can be eliminated.

3.6. The Mantis Control Plane

The Mantis control plane runs on a switch’s onboard CPUs and uses the measurements and malleable code described in the previous sections to interact with the switching ASIC. Modern data center switches already use this CPU for tasks such as routing, monitoring, and configuration; however, these interactions are traditionally assumed to be one-off and asynchronous, i.e., ‘on the slow path.’

We pursue a different goal: to, as quickly as possible, poll data plane registers and react to them in a user-defined fashion. Rather than treating each interaction between the data and control plane as an isolated event, Mantis presents an alternative architecture—one where the control plane executes one of a set of predetermined actions repeatedly, and without pause. With a highly optimized control-plane agent and driver, Mantis can execute iterations of the control plane loop at granularities that are on the same order of magnitude as the PCIe latency of the underlying system, and an order of magnitude lower than a typical data center RTT.

Control plane architecture. The operation of the Mantis control plane is split into two phases:

1. *Prologue:* The prologue phase encompasses the initialization of malleable values/fields, populating initial table entries, setting up memoization, and configuring driver sessions of the switch.

2. *Dialogue:* The dialogue phase is where the control plane—as rapidly as possible—polls measurement registers and executes user-defined reactions based on the collected measurement.

Mantis is explicitly optimized for repeated accesses and updates to the same set of reaction parameters and `malleables`; this includes several custom driver modifications that support repeat interactions. Mantis optimizations include precomputation of metadata during the prologue; batching of requests during the dialogue; and caching/memoization of device instructions in the prologue (for statically computable driver operations) as well as the dialogue (for repeated table modifications). The latter is particularly important for speeding up `mv` updates, etc. Thus, control flow is as follows:

```
// prologue
helper_state = precompute_metadata();
memo = setup_cache(helper_state);
run_user_initialization(helper_state, memo);
// dialogue
while(!stopped) {
  updateTable(memo, "p4r_init_", {measure_ver : mv ^ 1});
  read_measurements(memo, mv); mv ^= 1;
  run_user_reaction(memo, helper_state, vv ^ 1);
```

```
    updateTable(memo, "p4r_init_", {config_ver : vv ^ 1});
    fill_shadow_tables(memo, vv); vv ^= 1;
  }
```

The dialogue loop is single-threaded to avoid driver contention and consistency issues; however, if the switch contains multiple disjoint linecards or pipelines, these can be handled by spawning multiple Mantis agent threads, each handling its own component. To minimize latency, Mantis runs as a busy loop in a reserved CPU core, with the option to trade latency for lower CPU utilization.

Stateful dialogue. We note that Mantis allows users to retain state across iterations of the dialogue loop. Examples of behavior that may require this functionality include computing average throughput, tracking buffer depth gradients, or using the reaction loop to sample statistics over time. Mantis supports this behavior intrinsically through C `static` variables, which, when used inside a function, allocate space in the DATA segment of the program's memory and retain their value across function invocations.

Legacy control plane accesses. We also note that Mantis does not preclude legacy control plane accesses, e.g., for routing protocols, handling of higher-level protocols, and manual network operator interaction. Concurrent use is fine as the underlying drivers are typically designed to be thread-safe. Further, due to the poll-based and single-threaded nature of the Mantis agent, at most one reaction is active at any time. Thus, the CPU-ASIC interactions of a legacy application will only need to queue behind at most one set of operations from Mantis. We evaluate this effect in §3.8.2.

3.7. Implementation

We implemented a prototype of Mantis, including a P4R compiler, control plane agent, and modified driver infrastructure. Our prototype runs on a Wedge100BF-32X.

Compiler. The Mantis compiler translates `.p4r` files into a P4 program and C library. In total, the compiler implementation consists of 4,000 lines of C++ and around 1,250 lines of grammar. The compiler's parsing frontend is implemented with Flex/Bison. While parsing, the compiler builds an AST of the input program and, over several translation passes, adds/updates nodes to implement the transformations in §3.4 and §3.5. While parsing the P4R program, the compiler also extracts reaction function definitions. With the help of the P4 compiler, the compiler translates the arguments and malleable entity modifications into executable code that properly handles argument mirroring and isolation.

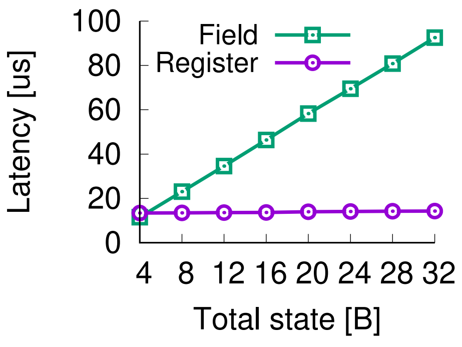
Dynamic loading. The C reaction function is compiled into a shared object with `gcc` so that, at runtime, Mantis can load the user reaction loops via shared objects and dynamic loading. Not only does this separate the implementation of the control plane from that of user code, it also potentially enables users to change their reaction functions without interrupting switch operations.

To signal a change, a user-defined signal will activate a transition flag in the running agent. The flag will break out of the reaction loop after any current dialogue completes, unload the previous dialogue module, and link the new shared object. Users can specify whether the prologue user initialization should be re-executed.

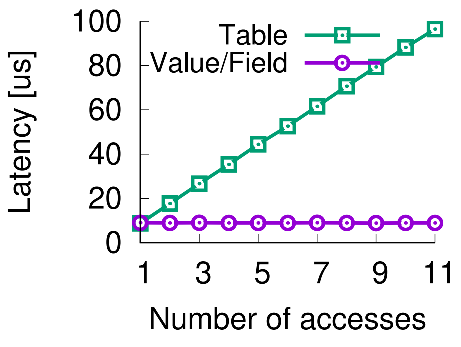
Control plane. Our prototype control plane dynamically unloads/reloads `.so` files that implement the reaction prologues and dialogues, before executing the high-frequency measurement and reaction loop described in §3.6. To ensure fast reaction time, we reserve a core for the reaction loop, configure the loop thread with high priority, and set a `SCHED_FIFO` real-time scheduling policy. As mentioned in §3.6, these can be scaled back in return for increased reaction time. We also modify the existing drivers and control plane interfaces in order to optimize latency.

		Malleables			LoC		Control Flow			Memory		
Example	**Reaction**	**val**	**fld**	**tbl**	**P4R**	**P4**	**Stgs**	**Tbls**	**Regs**	**SRAM**	**TCAM**	**Metadata**
Flow size estimation and DoS mitigation	Reads packet headers from the data plane to derive an estimate of the data sent by all senders in the network. Blocks senders that exceed a threshold rate.			✓	81	95	2	2	1	48KB	1.28KB	263b
Route recomputation	Detects failures using a gray failure detector—marking the link as down if received heartbeats dip below $\delta = \lfloor \eta \frac{T_d}{T_s} \rfloor$ for consecutive loops. Routes are recomputed on detection.			✓	30	158	1	6	6	192KB	0KB	160b
Hash polarization mitigation	Reads queue depth of a set of load-balanced ECMP ports. If there is persistent imbalance, changes the ECMP hashing strategy to prevent polarization.		✓		157	245	3	8	1	160KB	0KB	498b
Reinforcement Learning	Reads packet counts and queue depths for different ports to compute a reward function. Uses RL techniques to optimize DCTCP marking threshold reconfiguration policy.	✓			132	239	2	13	6	192KB	0KB	380b

Table 3.1: Examples of network features that can be formulated as reactions. Evaluation metrics are measured in terms of marginal increase over a basic router. See §3.8 for a more detailed evaluation of these examples.



(a) Argument measurement.



(b) Malleable entity updates.

Figure 3.10: Latency of raw measurements/updates in Mantis. These numbers do not include isolation mechanisms.

3.8. Evaluation

We evaluated Mantis using our prototype implementation. All experiments were run on a hardware testbed consisting of a Wedge100BF-32X switch connected to a set of servers via 25 Gbps links.

3.8.1. Mantis Achieves Fast Reaction Times

To measure the reaction time of Mantis, we microbenchmark raw operations (before the mechanisms of §3.5) in Mantis, from which we can construct a cost model. The microbenchmarks show the latency of reading reaction arguments and writing to `malleables`.

Figure 3.10a plots the latency of measuring the data plane versus the total size of the state that is read. We show results for both 32-bit field arguments and 32-bit register arguments. For field arguments (ingress or egress), the latency of measurement is dependent on the number of packed 32-bit registers that the control plane must read. This value increases linearly modulo packing efficiency. For register arguments, our kernel driver optimizations ensure that reads of multiple entries of a single register array are cheap, each additional byte incurring only 10s of ns of latency. We do not show results for reading from multiple register arrays as the results are identical to field arguments (as field arguments are implemented as registers).

Figure 3.10b shows results for updates of the data plane. Here, we plot latency versus the number of updates. For scalar `malleables` (fields and values), the latency of updates is constant as long as

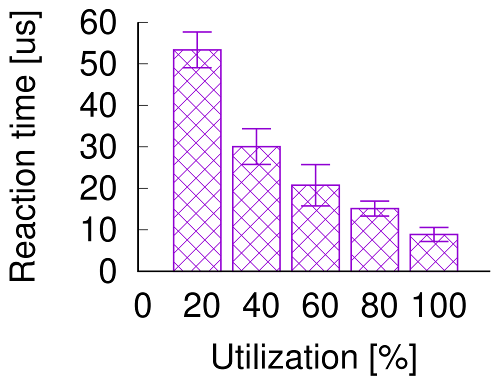


Figure 3.11: CPU utilization and reaction time tradeoff.

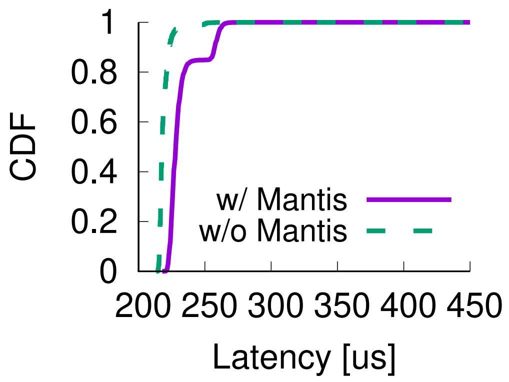


Figure 3.12: Latency of a concurrent legacy table update with and without Mantis.

all of the accesses can be handled within a single `p4r_init_` table (very large in today's switches). After that point, we would need to include the latency of the update protocol. For malleable tables, latency increases linearly with the number of entries modified. Note that, for table insertions, the latency may be more complex [52]; however, we anticipate that most reactions will be updates or involve smaller tables.

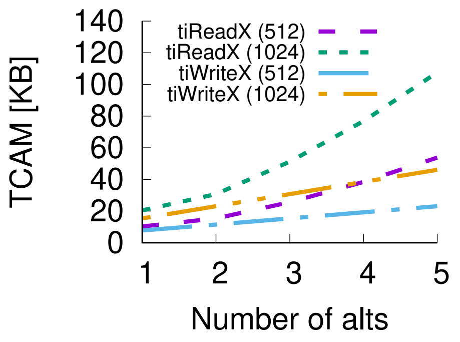
The total latency of a reaction function is, thus, approximately:

$$F_{3.10b}(1 \text{ tblMod}) + \sum_{a \in args} \Big(F_{3.10a}(a)\Big) + C$$
$$+ \sum_{t \in tblMods} \Big(2F_{3.10b}(t)\Big) + 2F_{3.10b}(N_{init} - 1) + F_{3.10b}(1 \text{ tblMod})$$

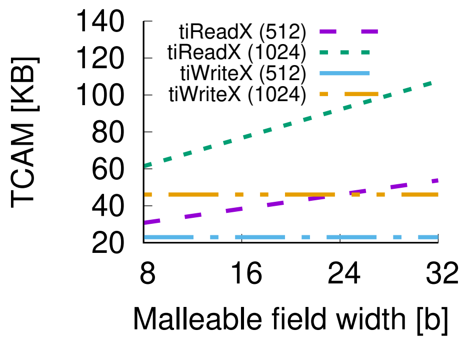
where $F_{3.10a}$ and $F_{3.10b}$ are functions that correspond to Figures 3.10a and 3.10b, respectively; C is the execution time of the reaction logic; and N_{init} is the number of `init` tables in the generated program. The first half of the equation corresponds to the latency of serializable measurement and the reaction logic. The second half corresponds to the latency of serializable updates. For all of the use cases in Table 3.1, end-to-end reaction time was on the order of 10s of µs.

3.8.2. Mantis Can Co-exist with Other Functions

We next explore Mantis's overhead in switch and CPU resources.



(a) $K = 32$.



(b) $A = 5$.

Figure 3.13: Malleable field TCAM usage.

CPU. By default, the Mantis control plane agent occupies one dedicated core for its dialogue loop; however, as mentioned in §3.6, it is possible to reduce this utilization at the cost of slower reaction times. Figure 3.11 shows this tradeoff for the update of a single malleable field with `nanosleep` for pacing. Reducing utilization to 20% still keeps the average reaction time to 10s of µs.

We also evaluate the impact of Mantis's fast reaction loop on concurrent, legacy switch operations. Specifically, we configure a parallel control plane (running on a different core of the switch CPU) that submits a continuous stream of table entry updates to the switch. We note that this is likely more aggressive than most legacy control planes. Mantis does slow down its neighbor, but the impact is relatively small; it mostly comes when the neighbor's update is blocked behind Mantis's current operation, creating a bimodal distribution. Even so, the median and p99 latency of legacy switch operations in the presence of Mantis versus without it are within 4.64% and 6.45%, respectively.

Memory. The other primary overhead of Mantis is switch memory, which is used for init tables, measurement registers, malleable table shadow entries, and transformations for malleable fields. The effect of the first three sources are simple to reason about: init tables add a small number of tables with 1–2 entries each, measurement registers are proportional to the number of arguments, and shadow table entries double the memory required for each malleable table.

The memory cost for malleable field transformations is slightly more complex. To evaluate it, we consider a K-bit malleable field `${X}` with A possible alternatives. We use a table `tiWriteX` that matches on the 5-tuple (all ternary matches) and writes to `${X}` in an action, similar to Figure 3.5. We also use a table `tiReadX` that uses X in an action *and* as a field match, similar to Figure 3.6 but matching on the 5-tuple plus X.

Figure 3.13 shows both tables' TCAM usage (the main bottleneck in this scenario). We evaluate two table occupancies, 512 and 1024. These are the number of user-defined entries, not the number of actual entries, which will be higher to account for the instantiated actions. As shown in Figure 3.13a, for a given field width, TCAM usage scales linearly with A in `tiWriteX`. For `tiReadX`, usage is asymptotically quadratic because the compiler needs to instantiate actions *and* add A extra ternary match columns. Varying the field width, we obtain Figure 3.13b, which shows that for `tiReadX`, usage is proportional to K and `tiWriteX` size is constant with respect to K as, when A is fixed, the number of action instantiations is fixed.

3.9. Mantis, in Context

We also evaluate Mantis in the context of the use cases of Table 3.1. We emphasize that these examples are not complete solutions, nor do they preclude the existence of a future workaround. Rather, they are intended as instruments through which we can understand the utility of Mantis, its relationship to existing data/control-plane alternatives, and the range of what it can express.

3.9.1. Flow Size Estimation and DoS Mitigation

The first use case we examine includes a classic problem in computer networks: flow size estimation. Flow sizes are useful for a variety of tasks, including Heavy Hitters, DDoS victim detection, etc [131]. Unfortunately, given the scale of today's networks, obtaining a precise account of the network's flow sizes is not always feasible. Instead, most modern approaches rely on approximation.

Two solutions are prototypical for this problem. The first is sFlow and its variants [151] where the control plane constructs approximate flow statistics from sampled packets. The second is sketch-based approaches [169] where the data plane records, in a compact representation, statistics over flows. With representatives in both a traditional control plane utility and programmable data plane

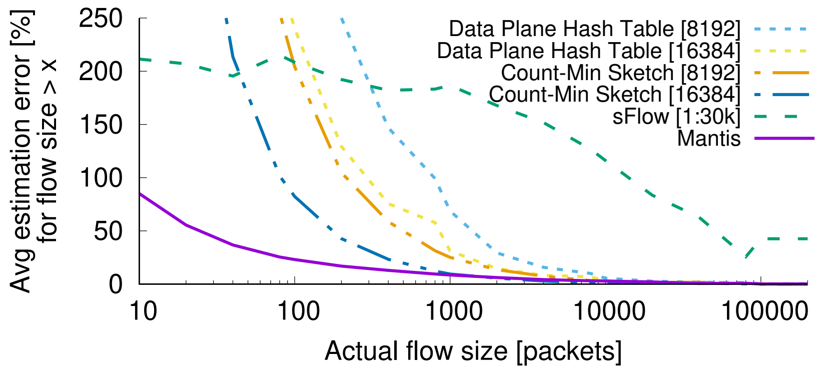


Figure 3.14: Average estimation error for Mantis and several alternatives. Mantis outperforms sFlow by orders of magnitude, and when equalizing the number of stages, beats data plane implementations for small flows as well.

algorithm, this use case is an ideal proving ground for Mantis.

As a reaction, we use a similar setup to Poseidon [202], which among other things, proposed dynamic reinstallation of data plane programs to respond to DDoS attacks. For simplicity, we model their per-sender statistics and rate-limiting defense, but the same techniques would apply to 5-tuples and more complex defenses.

Algorithm. We configure the data plane to track the current packet's source IP and a counter of the total number of bytes received. The reaction takes these two values as parameters and keeps a hash table of all sources. On every iteration, it attributes the marginal increase in total byte count from the previous dialogue to the given source IP. The reaction then computes the rate using $\frac{\hat{f}_t - \hat{f}_{t_0}}{t - t_0}$, where $\hat{f}_t$ is the counter at time t and t_0 is the time immediately prior to the first observation of the flow. To prevent spurious detection of new flows, we impose a minimum duration before a flow becomes eligible for blocking. For our experiments, we set a simple 1 Gbps threshold, but reiterate that arbitrary C is allowed.

Results. To evaluate the accuracy of size estimation in Mantis, we use `tcpreplay` with a CAIDA [10] ISP-backbone trace. For this experiment, Mantis was able to sustain a sampling rate of $\sim$10 µs, corresponding to an average of $\sim$1 in 5 packets.

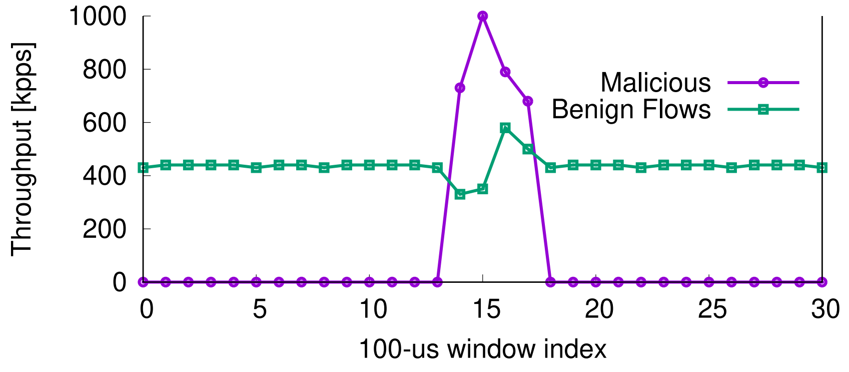
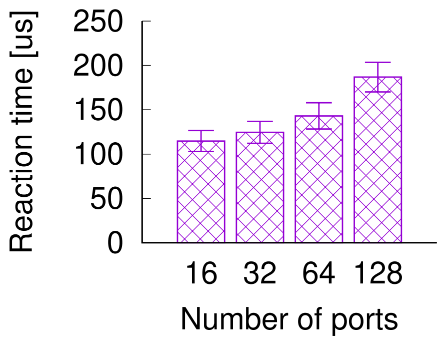


Figure 3.15: Aggregate throughput for legitimate flows from $\boldsymbol{S_{\{1,2,\ldots,250\}}}$ sending to $\boldsymbol{D}$. When $\boldsymbol{S_0}$ begins to flood the network, Mantis detects and suppresses it orders of magnitude faster than similar systems (cf. Figure 14 of [202]).

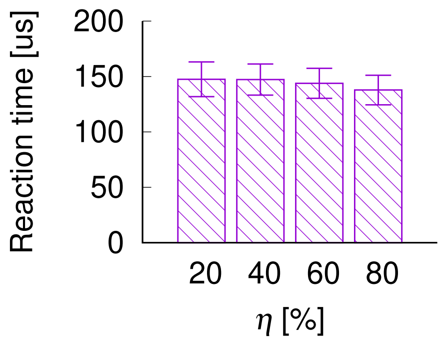
Figure 3.14 shows the average estimation error of Mantis versus the sFlow-based estimator and a pair of data plane implementations. For sFlow, we use the 1:30,000 sampling frequency suggested in [155]. For the data plane implementations, we include a hash table as well as a 2-stage count-min sketch. Following the configuration guidance of [169], we chunk the trace into 20 s blocks (each with around 8.9 M packets and 370 K flows) and configure all tables to have 8,192 entries (the next power of two above their setting of 4,500). We also show data plane results for 16 K entries; Mantis's performance was unchanged.

Compared to sFlow, Mantis is significantly more accurate due to its higher sampling frequency. This effect becomes pronounced as we approach sFlow's sampling granularity. Compared to both data-plane approaches, Mantis provides slightly worse but comparable accuracy for large flows, and orders of magnitude better accuracy for small flows. The overall trend holds across table sizes. The reason is that, in Mantis, inaccuracy is caused primarily by sampling error, which is bounded; in contrast, sketch inaccuracy is due to collisions, which may misattribute arbitrarily many bytes to the wrong flow.

Figure 3.15 shows the reaction in action. 250 legitimate TCP flows utilize 20% of a 10 Gbps bottleneck link before a single malicious sender arrives and blasts UDP traffic at 25 Gbps using a DPDK sending script. The Mantis reaction can install a mitigation rule within $\sim$100 µs (from the timestamp of the first packet of the malicious flow). Accounting for packet delays and TCP mechanisms,



(a) Latency vs ports ($\eta = \mathbf{0.5}$)



(b) Latency vs η (ports $= \mathbf{64}$)

Figure 3.16: The time to accurately detect failures and reroute for a robust Mantis-based gray failure detector.

the benign flows return to steady-state operation within ~500 µs, orders of magnitude faster than traditional reconfiguration.

3.9.2. Route Recomputation on Gray-failures

The second use case leverages Mantis's reaction time more directly via a gray-failure route recomputation scheme. In this use case, the reaction loop measures the frequency of heartbeats from neighboring nodes and triggers a control-plane route recomputation when the frequency drops below a threshold.

Algorithm. Our failure detection scheme is based on a previously proposed gray-failure detector [128] whose original design required specialized hardware support. In our formulation, we install in every node adjacent to the switch a heartbeat generator that produces high-priority packets at a granularity of T_s (1 µs in our tests). The detecting switch accumulates a per-port count of these heartbeats and the current timestamp in the data plane.

By polling (serializably) the counts and timestamp, a Mantis reaction can compare the number of observed heartbeat messages with the number of *expected* messages. More specifically, it can use a threshold $\delta = \lfloor \eta \frac{T_d}{T_s} \rfloor$ where T_d is the time since the last dialogue and $\eta \in [0, 1]$ captures expectations for the successful delivery of heartbeats—a high η will demand a more reliable link and catch failures faster and a low η will allow for more outliers at the cost of reaction time. Two consecutive polling periods with fewer than δ heartbeats trigger recomputation and installation of new routes.

A few features of this use case would be challenging without Mantis. Compared to a traditional control plane solution that polls raw packet counters, Mantis offers fast reaction speed and serializable reads of counters/timestamps that remove inaccuracies due to data/control-plane latency. Compared to a fully data plane solution that computes the threshold and activates detours, Mantis avoids the significant overheads of approximating division when computing δ [161] by offloading it to the control plane. Involving the control plane also opens up the possibility of arbitrary route recomputation and table modifications (data planes are typically limited to static backup paths or inefficient detour protocols [126]).

Results. Figure 3.16 shows the end-to-end reaction time of failure detection and route recomputation in Mantis for various configuration parameters. To emulate link failures, we leveraged switch APIs that disabled physical ports on the switch. Reaction time is defined as the difference between the control-plane timestamps of the link-down event and the installation of the new routing rules.

Figure 3.16a shows that Mantis can restore connectivity within 100–200 µs with low variance. What variance it does have is a result of the position of the failure in the first T_d window—if it occurs right before the ASIC reads the count, detection is faster. Figure 3.16b shows reaction time for different ηs. Overall, the impact of η is low as the majority of the reaction time is due to measuring all of the ports and ensuring isolation. We contrast the above results to typical control plane failure detectors that require 10s of ms to detect failures and an additional few ms to route around them [128]. We also contrast the results to an idealized detection algorithm [88], which would be limited by sampling accuracy rather than detection latency. For example, η=20% and T_s=1 µs implies a minimum reaction time of 15 µs. Waiting for consecutive threshold violations would increase this time. The slightly lower latency comes at the cost of the benefits of control plane route recomputation.

3.9.3. Hash Polarization Mitigation

Our third example is inspired by conversations with production network operators who noted the need to tune ECMP hashing functions for optimal load balancing given a particular packet header distribution. P4R can be used to reconfigure the inputs to a hash function at runtime to shift the function over time.

It can accomplish this by replacing the '5-tuple' input into the ECMP hash function with five malleable fields, each of which can become a reference to alternative fields in the packet's headers. To constrain the number of instantiated field_lists, we apply the optimization from the end of §3.4.1. The reaction function then takes a register array of per-egress packet counters and computes the Median Absolute Deviation (MAD) of the port utilizations. When the MAD differs for a sufficient amount of time, it shifts the inputs to find a better hash configuration for the current workload.

Compared to a control plane implementation that polls egress counters, Mantis provides isolation guarantees, which have been shown to be critical when evaluating ECMP balance [203, 187]. Compared to a data plane implementation that does the comparison in-band, the algorithm requires two operations that are difficult on today's switches. The first is the need to propagate egress counters (where packet counts can be observed) to the ingress (where routing decisions are possible), which often requires a recirculation. The second is the MAD computation, which traditionally requires computing the data's median, then the median difference from that value. A streaming algorithm suitable for use in networks would likely require us to use estimates of the median over only prior data and assume that value does not change significantly over time [4].

3.9.4. Reinforcement Learning

Finally, we note that Mantis's reaction abstraction is a good fit for feedback loops like those of Reinforcement Learning (RL). More formally, in each iteration i, the Mantis agent measures data plane state, s_i, and reacts in some way, a_i. The state then transitions to s_{i+1} resulting in a scalar reward r_i. During execution, Mantis will make observations of the form $e_i = (s_i, a_i, r_i, s_{i+1})$ and adjust to maximize the expected cumulative reward.

Many tasks can fit into the above framework, but as an example, we consider the task of tuning the DCTCP ECN threshold heuristic [33] to optimize the sum of the utilization of the switch with the inverse of queue length. We do this via off-policy Q-learning [175]. Specifically, we cast the ECN marking threshold as a malleable value, configurable from the control plane via reactions a_i, and poll queue depth and a counter register from the egress pipeline as the observed state s_i. At each step, Mantis uses an ϵ-greedy policy to either exploit or explore the space; updates of the state-value function follow the TD control algorithm in [175].

Although others have proposed in-network RL previously, these solutions have tended to rely on custom accelerators [122]. RL is difficult in existing switches both because of the need for a feedback loop and the extremely limited computational ability of switch ALUs. Instead, Mantis-based RL can leverage the CPU and can easily extend to arbitrary models, including neural networks.

3.10. Related Work

Over the years, a sequence of influential work [176, 137, 49] has provided network operators with an increasing amount of control at an increasingly fine granularity. A subset has also looked at control plane latency, though usually in the context of SDNs [171, 52].

Workarounds for data plane limitations. We are not the first to observe the limitations of today's programmable switches [46], and a slew of recent work has proposed data plane approximations/workarounds for specific building blocks [162, 161, 169]. While both innovative and effective, it is not clear that every protocol can be adapted to a pure P4-model, nor is it clear that every operator will have the time/expertise to develop an adaptation. Our work takes a different approach, allowing users to use arbitrary C as long as the application fits in the 'reaction' paradigm.

A subclass of the above workarounds involves the control plane in the workaround for precisely the reasons Mantis does [187, 163, 186, 202]. Mantis is a generalization of these proposals and one that presents a convenient (and potentially finer-grained) abstraction.

Alternative hardware architectures. Similar in spirit is work that has proposed alternative hardware solutions to the problem of data plane expressivity. Some of these propose and use modifications to RMT-style switches [90, 167, 142, 105, 88]; others propose the use of distinct hardware architectures such as FPGAs [132, 66, 96]. Unfortunately, given the current trends of network bandwidth versus compute power, it is unlikely that future switches and routers will be both line-rate and Turing-complete. In contrast, Mantis strives to provide a high degree of expressiveness on today's RMT switches.

Update and measurement isolation. Ensuring consistency and isolation of network updates is a classic problem in traditional and SDN networks, and many solutions have been proposed in those domains [67, 154, 98]. Some of these have also used a two-phase protocol [32, 106, 154]; however, as mentioned in §3.5.1.2 our focus on frequent, repeated updates of the data plane distinguish our design, implementation, and optimizations. Related work in the data plane has instead tended to focus on cross-pipeline consistency [187] and intra-pipeline atomicity [168].

Data plane virtualization. Finally, prior work has also observed the utility of match tables for runtime modifications [80, 200]. As mentioned in §3.3, however, this comes at a high cost. Instead, we target reactions in which most of the data plane is fixed, with only a few parameters dependent on current network conditions.

3.11. Conclusion

In this chapter, we describe Mantis, a framework that reformulates common network tasks as reactions to current network conditions. We show that the Mantis compiler and control plane architecture enable fine-grained RTT-level reaction loops, while the P4R language simplifies the process of designing and implementing reactions.

CHAPTER 4

REDUCING ‘TAX’ OF PARTIAL SNAPSHOTS FOR MANAGING DISTRIBUTED CLOUD SERVICES

All problems in computer science can be solved by another level of indirection.

David Wheeler

This chapter was previously published in press as *Liangcheng Yu, Xiao Zhang, Haoran Zhang, John Sonchack, Dan Ports, and Vincent Liu. Beaver: Practical partial snapshots for distributed cloud services. In 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2024)*. The dissertation author led all phases of the project, from idea development to system prototyping and writing.

Abstract. Distributed snapshots are a classic class of protocols used for capturing a causally consistent view of states across machines. Although effective, existing protocols presume an isolated universe of processes to snapshot and require instrumentation and coordination of all. This assumption does not match today’s cloud services—it is not always practical to instrument all involved processes nor realistic to assume zero interaction of the machines of interest with the external world.

To bridge this gap, this chapter presents Beaver, the first practical partial snapshot protocol that ensures causal consistency under external traffic interference. Beaver presents a unique design point that tightly couples its protocol with the regularities of the underlying data center environment. By exploiting the placement of software load balancers in public clouds and their associated communication pattern, Beaver not only requires minimal changes to today’s data center operations but also eliminates any form of blocking to existing communication, thus incurring near-zero overhead to user traffic. We demonstrate the Beaver’s effectiveness through extensive testbed experiments and novel use cases.

4.1. Introduction

The ability to capture a consistent, global view of a system is a powerful tool. For many tasks—deadlock detection, checkpoints and failure recovery, network telemetry, debugging of distributed software, and many others [188, 112, 43, 118, 179, 130, 44, 65, 22, 189, 159, 31, 8, 23]—a global view, and particularly a consistent one, is essential for correct operation. Without consistency, results are unreliable, and the value of associated tools is questionable.

The classic method for capturing consistent global states is the Chandy-Lamport snapshot algorithm that was proposed almost four decades ago and its subsequent variants [51, 115, 112, 136, 114, 83, 183, 188]. At a high level, these protocols flood snapshot initiation messages throughout the system, triggering local captures of state at every node they pass in a manner that guarantees causal consistency of the recorded values. Some versions (including the original) also include support of capturing messages that are in-flight at the time of the snapshot, i.e., channel state.

While these protocols have been simple, effective, and widely used for decades, they all rely on the fundamental assumption that the set of participants in the protocol is closed under causal propagation. In other words, if any node can both send and receive messages from participants in the protocol, it can propagate Lamport's 'happened-before' relation [115] and must also be a participant in the snapshot. For systems operating in isolation, ensuring full participation is trivial; however, modern cloud deployments are not so utopian.

Today's cloud services are often modular, e.g., structured as microservices, each of which might be developed and maintained by a different user, team, or organization or hosted on otherwise inaccessible infrastructure. Take, for instance, a managed pub/sub messaging layer like Amazon's Simple Notification Service (SNS). As a proprietary and black-box service, users cannot directly propagate snapshot initiation markers through the service. Further, while they might be able to add markers to the application-level content manually, with concurrency, replication, and reordering (e.g., due to prioritization), content-based markers are unlikely to track causal relationships accurately. Even when developers fully control all relevant servers, the clients of the service can also introduce hidden causal relationships, for example, when the user of a generative AI chatbot sends a follow-up message based on the response to the previous prompt. Ultimately, the nature of causal consistency means that a single non-participant can render all snapshots useless.

Observing this gap between classical assumptions and the practicalities of real-world deployments, we ask the question: Can we make distributed snapshots practical in modern cloud data centers, i.e., is it possible to capture a causally consistent snapshot when only a subset of the broader system participates? At first glance, this goal seems far-fetched: With partial participation, we cannot control the messaging behaviors nor instrument any coordination logic for machines external to those of interest. Complicating the issue is the fact that, to be practical, the protocol cannot block, e.g., by buffering or delaying user packets during a snapshot. In essence, this means that hidden causal relationships between participants and external communication partners are unavoidable.

This work presents Beaver[10], the first 'partial' snapshot protocol that extends the capability of distributed snapshots to cloud services with external interactions. Beaver provides the same basic abstraction as other snapshot protocols—for any event whose effects are observed in the snapshot, all other events that 'happened-before' are also included. It achieves this even when the target service communicates with an arbitrary number of external, black-box entities, regardless of their scale, semantics, or placement, and despite potential multi-hop propagation of causal dependencies. Beaver does all of this without blocking or delaying user requests. Beaver tackles this seemingly impossible problem by:

1. Relying on two features found in all of today's largest cloud data centers: (a) Layer-4 Software Load Balancers (SLBs) that interpose on a subset of inbound traffic [147, 61, 146, 18] and (b) servers with low time strata or otherwise stable clocks [15, 56, 2, 145, 124, 141, 117, 78].
2. Eschewing the enforcement of causal consistency in favor of simply *detecting* when violations may have occurred, a mechanism we call Optimistic Gateway Marking (OGM).

Note that for (1b), Beaver does not rely on the traditional notion of clock synchronization that other

[10]The animal species known for their engineering expertise in constructing dams using locally available materials such as rocks and tree branches.

recent systems [56, 15, 124] are founded upon, which requires that the clocks of distinct machines have bounded drift. Instead, it uses a much weaker property [78, 134] over the frequency drift of a single machine[11]. Also note that (2) implies a tradeoff: snapshots are not always successful, but users can be assured of their correctness when they are and retry when they are not.

At a high level, Beaver's approach is based on the observation that when examining the causal consistency of a snapshot, only inbound traffic is relevant and only a small subset therein. More specifically, we can divide inbound traffic into messages that are 'causally irrelevant' (e.g., triggered asynchronously and, thus, are not a part of any transitive causal relationships) and messages that are 'causally relevant' (e.g., triggered by post-snapshot outbound traffic but may not carry any markers of that fact). Beaver's OGM mechanism is an approximate but full-recall detector of causally relevant traffic.

Our prototype[12] of Beaver demonstrates that not only is it possible to build an OGM mechanism, but by leveraging the aforementioned features of today's cloud data centers, we can render the possibility of rejected snapshots minimal (near-zero in many cases). To summarize, this chapter makes the following contributions:

- To the best of our knowledge, we are the first to detail the gap between classical assumptions of distributed snapshots and the practicalities of real-world clouds.

- We propose Beaver, the first partial distributed snapshot primitive for modern cloud services. Beaver presents a unique design point by tightly coupling the protocol with the regularities of the underlying data center environment.

- We evaluate Beaver through end-to-end implementation on a real-world testbed aligned with the production data center settings. We also show that the causally consistent view provided by Beaver enables a spectrum of use cases.

[11]Bounded clock drift to a low-stratum reference server is sufficient to guarantee bounded local frequency drift, but not necessary.

[12]The prototype is available at https://github.com/eniac/Beaver.

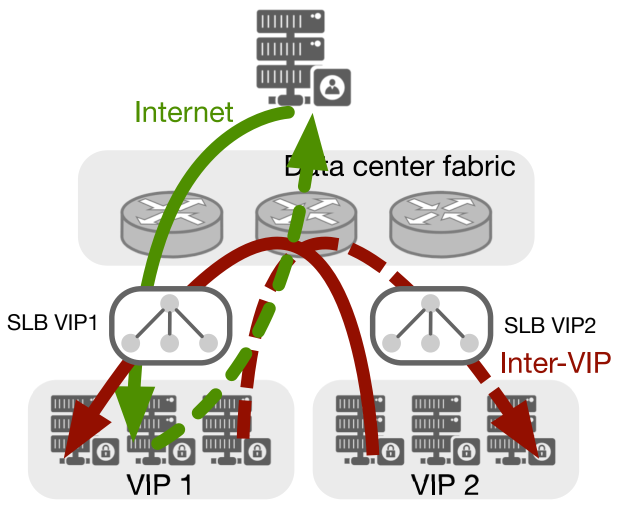


Figure 4.1: Today's public cloud services place SLBs to handle the external traffic to its VIP in the inbound direction (solid lines to VIP 1). The response to inbound messages (dotted lines from VIP 1) typically bypasses its SLB to minimize the SLB traffic load.

4.2. Background and Motivation

We begin by describing the structure of today's cloud services and the data centers in which they reside before we discuss the application of distributed snapshots to these services.

4.2.1. Communication in Public Cloud Data Centers

Today's cloud data centers are massive collections of servers connected by a network fabric that host user services of diverse sizes and scopes. In this context, we can abstract user services as a set of virtual or bare metal machines managed as a single logical entity. Each service is typically assigned a public Virtual IP (VIP) address, and each physical machine a private Direct IP (DIP) address [147, 61].

Software Load Balancers (SLBs). A set of dedicated servers or programmable devices is responsible for translating between VIPs and DIPs. We refer interested readers to prior work [147, 61] for full details, but at a high level, these layer-4 devices act similarly to traditional Network Address Translators (NATs), allocating a new mapping for every new connection and rewriting the headers of every passing packet according to the mapping. In cloud systems, these devices are distributed, replicated, and serve an additional purpose as software load balancers that spread requests over available backend servers. A single service/VIP typically has a dedicated set of SLBs based on its scale (e.g., $\sim$7–20, including replication).

The path of packets in public clouds. In the presence of SLBs, packets can take different paths depending on the relationship between their source and destination (Figure 4.1):

Internet traffic: Incoming packets from the Internet are always routed through an SLB to translate from the service's publicly visible VIP to a relevant internal DIP [147, 61, 198, 138, 70, 69]. Unlike most other NAT-like mechanisms, response packets are usually sent back directly, bypassing the SLB using techniques like Direct Server Return (DSR) [147, 61].

Inter-service traffic: Inbound traffic from other services within the same provider also passes through SLBs [147, 61, 198, 138, 70, 69], which still need to perform the same VIP-to-DIP translation. This is true even if the two service's servers are physically adjacent. Note that, like with Internet traffic, outbound traffic can bypass the responder's SLB; however, even in this case, the packet will still need to pass through the SLB responsible for the destination VIP(s), as shown in Figure 4.1. Note that while cached DIPs have been suggested to bypass inbound SLBs on the fast path [147], this optimization is currently disabled for major classes of production traffic due to load imbalance and cache management issues. The implication is that, at least for public clouds, this need to interpose on all inbound traffic is ubiquitous [61, 198, 45].

Intra-service traffic: Finally, messages between sources and destinations belonging to the same VIP are sent directly, bypassing the SLBs entirely.

Typical service communication patterns. In parallel to the above, we note that modern cloud services rarely operate in isolation. Frontend services typically rely on a wide array of backend services, e.g., to handle storage, analytics, and learning, thus triggering inter-service traffic. The rise of managed cloud service offerings and microservice design patterns have further encouraged modularity and the associated growth in the number of distinct services involved in processing a single

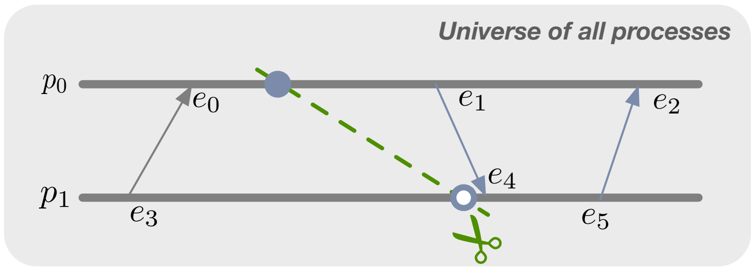


Figure 4.2: A minimal example of a consistent cut for 2 processes p_0, p_1 and 6 events $e_{0,1,\ldots,5}$. The global snapshot formed from the collection of ○ and ● is a 'causal cut' of the event timelines for all processes, where ● and ○ indicate snapshot initiations triggered out-of-band or by receiving marker messages, respectively.

user request. At a more basic level, most cloud services take requests from and return responses to external clients, each with its own internal, causality-carrying logic.

4.2.2. Revisiting the Chandy-Lamport Snapshot

The ability to capture a consistent snapshot of a cloud service's global state is a powerful tool. Indeed, many problems in distributed systems boil down to determining the global state across machines, including distributed logging and debugging, network telemetry, checkpointing and recovery, and deadlock detection [181, 51, 112, 65, 188, 23].

Intuitively, a snapshot is a collection of local states captured from the processes of a system. For simplicity, we omit channel states in our definitions, but the analysis is similar. The snapshot is deemed consistent if the captured states at each process 'cut' the timeline of events in a way that respects the following definition:

Definition 1. (Consistent Snapshot [51, 181]). For a snapshot, let C be the set of events on every process that occurs before the 'cut'. C is causally consistent iff $\forall e \in C$, if $e' \rightarrow e$, then $e' \in C$, where $x \rightarrow y$ denotes that x 'happened before' y.

The seminal Chandy-Lamport algorithm was the first to present a solution for this problem. We refer the interested readers to the original paper [51] or a distributed systems textbook [181, 112] for complete details, but we give a simplified description of the model and the protocol below:

- *Model:* A system involves a set of asynchronous processes $P = \{p_0, p_1, \ldots, p_{N-1}\}$ that interconnect with each other through FIFO message channels. Each process p_i holds state of interest, s_i, that may change in response to local events (e.g., local computation, message sends or receives, etc.). A global snapshot involves a union of states $\{s_i\}$ recorded at different times for all processes.

- *Protocol state machine:* The protocol requires coordination in *all* processes $p \in P$. An initiator process first records its local state and then sends a marker message to all others. The captured state is application-dependent and can range from a single bit representing the state of a lock to all of local memory. When any other process p_i receives a marker message for the first time, it records its state s_i and, to ensure consistency, sends marker messages immediately through all other channels.

Later variants refine the basic algorithm to generalize channel assumptions, allow for concurrent initiation, or reduce message complexity [112, 136, 114, 83, 183, 188]. In particular, the Lai-Yang algorithm permits non-FIFO and lossy channels by having processes piggyback a single marker bit in every sending message [114] rather than sending separate marker messages as in the original protocol. Upon receiving a message with a marker bit set, the receiving process *first* records the local state, processes the payload, and sets the bit for future sending messages. Additional bits can be used to support concurrent snapshots. Figure 4.2 shows a consistent cut with the Lai-Yang algorithm.

4.2.3. A Case for Partial Snapshots

The above snapshot algorithm makes a fundamental and implicit assumption that *all* processes that can communicate with processes in P are themselves in P. Unfortunately, as previously mentioned, today's cloud services are frequently interconnected, with efforts toward modular design and managed solutions promoting increasing complexity in the dependency graph over time. As a rough indication of severity, previous studies have shown that inter-service traffic comprises 10–50% of total traffic in the data center, and Internet traffic accounts for 5–25% [147, 69, 70].

Consider, for instance, a HuggingFace-like ML inference service [26] that hosts a collection of models that can be accessed from external clients. As they are externally visible, the models are frequently used in larger jobs, e.g., as part of an interactive chatbot (where clients submit requests based on

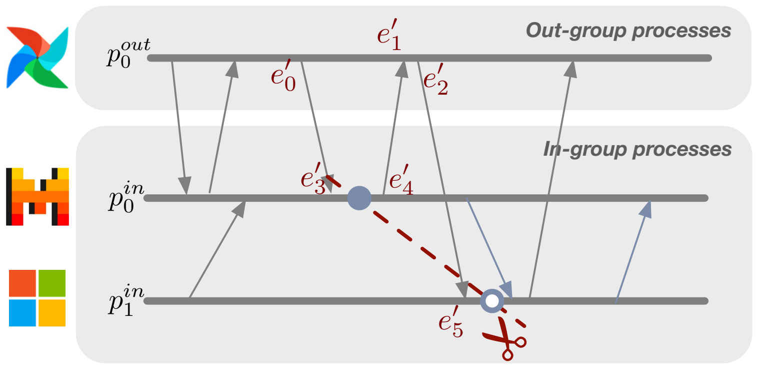


Figure 4.3: An application where a distributed serving system is accessed by an external user (e.g., an Apache Airflow workflow). The out-group process p_0^{out} imposes a hidden causal relationship $e'_4 \rightarrow e'_5$ between events e'_4 and e'_5, rendering a traditional snapshot of only the serving system inconsistent.

prior responses) or more complex Apache Airflow workflows.

The inference service might want to capture a service-wide statistic (e.g., tracking the maximum number of in-flight requests) to decide on the number of servers to provision. Any analysis of the developer's application that does not consider the potential dependencies introduced by external services or clients will miss important causal dependencies.

Figure 4.3 shows a simple example of this, where a single external Airflow job makes requests to multiple models hosted by the inference service such that only one request is outstanding at any given time. Occasional internal messages are for monitoring and coordination. Although there is at most one outstanding request at any given time, a traditional distributed snapshot that only considers the inference service will not respect that bound.

For example, in Figure 4.3, the depicted cut 'observes' two inflight messages because it fails to capture the external interactions ($e'_5 \in C$, yet $e'_4 \notin C$). In fact, for a single client that issues a single request at a time, an n-server snapshot can 'observe' any number of in-flight requests [0, n]. These arbitrary results can cause the developer to waste money and resources on redundant provisioning. More broadly, while the frequency and consequences of consistency violations are application-dependent, there is often a meaningful difference between 'correct' and 'incorrect'.

Although converting all cloud services into participants of the snapshot protocol might be possible given either (a) a well-resourced developer who can implement and manage everything (even if machines are geo-distributed or on the broader Internet) in-house or (b) support from the cloud provider to propagate snapshot markers on all packets, these approaches are not always feasible. For (a), the popularity of managed services demonstrates their importance to low-cost and agile development. For (b), forced instrumentation can lead to overhead and fragmentation for users not involved in the snapshot. Even worse, if the external source of dependencies is a human (e.g., accessing your service through a browser), incorporating her into the snapshot is impractical.

A formal definition of partial snapshots. We seek the design and implementation of a partial snapshot. In a partial snapshot, processes are divided into two groups. The first, *in-group processes* P^{in}, are the machines of the VIP(s) of interest. The second, *out-group processes* P^{out}, includes all other machines, whether in the same data center or the broader Internet.

Given these sets, we refine Definition 1 to obtain a definition of consistent partial snapshots:

Definition 2. (Consistent Partial Snapshot). Consider a universe of processes $P = P^{in} \cup P^{out}$, $P^{in} \cap P^{out} = \emptyset$. Let C_{part} be the set of pre-snapshot events for P^{in}. C_{part} is causally consistent iff $\forall e \in C_{part}$, if $e'.p \in P^{in} \wedge e' \rightarrow e$, then $e' \in C_{part}$.

Similar to traditional snapshots, for a set of in-group processes P^{in}, if a consistent partial snapshot includes the effect of an event e, it must include any event e' at $p \in P^{in}$ that leads to it. Like traditional snapshots, the 'happened before' relation, $\rightarrow$ is transitive and defined over events in the universe of processes. Unlike traditional snapshots, however, the included events only account for in-group events.

4.3. Gateway Marking

This chapter introduces Beaver, a partial snapshot primitive that captures a causally consistent collection of state for cloud services sitting behind one or more operator-specified VIPs.

Fundamentally, the nodes in P^{out} are uncontrollable and, as a result, can introduce arbitrary hidden causal relationships, disrupting the consistency of traditional snapshots. At the core of Beaver is

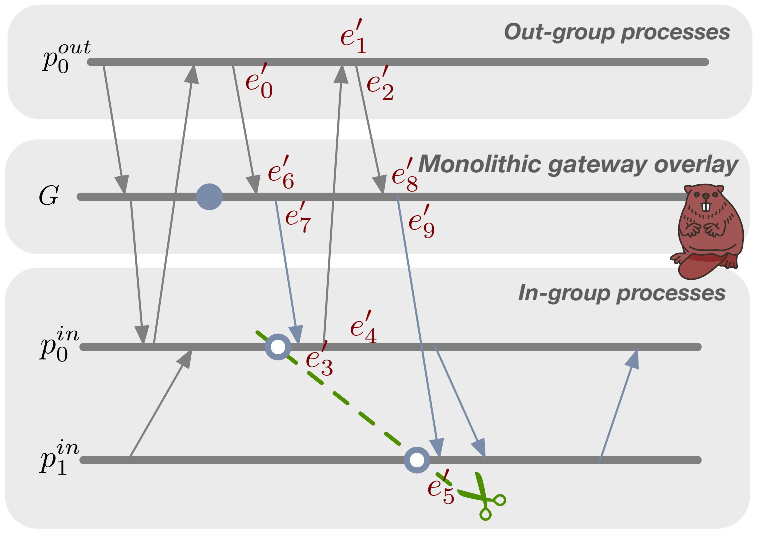


Figure 4.4: With the gateway indirection, Beaver's MGM results in a new frontier at the in-group process p_1^{in} that precedes rather than succeeds the event e'_5 (as in the scenario of Figure 4.3), converging to a consistent partial snapshot.

a primitive called Optimistic Gateway Marking (OGM), which allows Beaver to detect when such causality violations may have occurred. As we show later in §4.5, by combining this primitive with common-case features of today's cloud data centers, Beaver can provide:

- Partial deployability where *only* the in-group machines for the target VIP(s) participate while ensuring high-rate, consistent partial snapshots for the target service(s).
- Minimal cost for data center infrastructure, for example, without switch reconfiguration or additional SLB replicas.
- Near-zero impact on existing data center service traffic.

In this section, we first introduce a strawman version of the primitive before discussing practicalities and how Beaver addresses them with OGM in §4.4.

Strawman: Monolithic Gateway Marking (MGM). Beaver starts with a simple idea: for all packets originating from out-group nodes and destined for in-group nodes, route them through a gateway. The gateway is responsible for two tasks:

1. Tagging incoming packets to in-group nodes with snapshot markers.

Symbol	Description
P	Set of all processes.
P^{in}	Set of in-group processes with states of interest.
P^{out}	Set of out-group processes without any control.
G	Set of gateways handling inbound traffic for P^{in}.
C	Set of pre-snapshot events for a snapshot 'cut'.
e	Event tuple $e = (p, m, t)$.
$e.p$	The process at which an event e occurs.
$e.m$	The message involved in an event e, if any.
$e.t$	Global wall clock time, for ease of discussion.
e^{ss}_{gmax}	The event when the last gateway is in a new snapshot.
e^{ss}_{gmin}	The event when the first gateway is in a new snapshot.
e^{ss}_{g}	The event when $g \in G$ enters a new snapshot.
e^{ss}_{p}	The event triggering $p \in P^{in}$ to enter a new snapshot.
$d(p, q; V)$	One way delay from p to q with intermediate nodes $v \in V$ $(p, q \in (P \cup G), V \subseteq (P \cup G))$ in sequence.
τ_{min}	Min time for an external causal chain to occur.

Table 4.1: Summary of notations in Beaver.

2. Initiating snapshots by tagging all subsequent inbound messages accordingly.

After the gateway initiates a snapshot, the protocol proceeds as a traditional snapshot among the in-group nodes. For the strawman, assume that the gateway is implemented by a single monolithic node. Figure 4.4 shows an example execution using the above protocol and the same application-level communication pattern as Figure 4.3. In contrast to Figure 4.3, indirection and marking via a gateway cause p^{in}_1 to take the snapshot at the correct time. In a way, the gateway node in this protocol can be seen as a stand-in for all nodes in P^{out}. We can prove that MGM produces a consistent partial snapshot.

Theorem 1. With MGM, a partial snapshot C_{part} for $P^{in} \subseteq P$ is causally consistent, that is, $\forall e \in C_{part}$, if $e'.p \in P^{in} \wedge e' \rightarrow e$, then $e' \in C_{part}$.

Proof. Let $e.p = p^{in}_i$ and $e'.p = p^{in}_j$. There are 3 cases:

1. Both events occur in the same process, i.e., $i = j$.
2. $i \neq j$ and the causality relationship $e' \rightarrow e$ is imposed purely by in-group messages.
3. Otherwise, the causality relationship $e' \rightarrow e$ involves *at least* one $p \in P^{out}$.

In cases (1) and (2), the theorem is trivially true using identical logic to proofs of traditional distributed snapshot protocols. We prove (3) by contradiction.

Assume $(e \in C_{part}) \wedge (\exists e' \to e)$ but $(e' \notin C_{part})$. With (3), $e' \to e$ means that there must exist some e^{out} (at an out-group process) satisfying $e' \to e^{out} \to e$. Now, because $e' \notin C_{part}$, we know $e^{ss}_{p^{in}_j} \to e'$ or $e^{ss}_{p^{in}_j} = e'$, that is, p^{in}_j's local snapshot happened before or during e'. Combined with the fact that the gateway is the original initiator of the snapshot protocol, we know that $e^{ss}_g \to e' \to e^{out} \to e$.

We can focus on a subset of the above causality chain: $e^{ss}_g \to e$. From the properties of the in-group snapshot protocol, $e^{ss}_g \to e$ implies $e \notin C_{part}$.

This contradicts our original assumption that $e \in C_{part}$! $\square$

Theorem 1 implications: Beyond correctness, the strawman exhibits several valuable properties:

1. *Obliviousness to out-group semantics:* The proof treats the internals of the out-group processes as a black box. In fact, the protocol remains correct, even if the causal dependency results from multiple network hops through distinct out-group nodes or if an element of the out-group chain is a human.
2. *Obliviousness to outbound messages:* The gateway only needs to observe messages inbound to in-group processes without requiring any visibility or tagging of outbound messages. MGMs achieve this by initiating the snapshot at the gateway, which—as a stand-in for P^{out}—obviates the need to track dependencies carried to the out-group.

SLBs as a candidate for gateway marking. The SLBs described in §4.2.1 are a convenient candidate for implementing gateway marking as VIPs are a natural granularity for service-specific partial snapshots, and SLBs already interpose on all incoming traffic to a VIP—regardless of whether it is from the Internet or a different service. MGM's obliviousness to outbound messages helps here as well, making the system amenable to DSR.

Of course, assuming that a single server can handle all incoming traffic to a service is not feasible. The scale of modern SLBs serves as proof that even for simple gateway processing incoming requests for

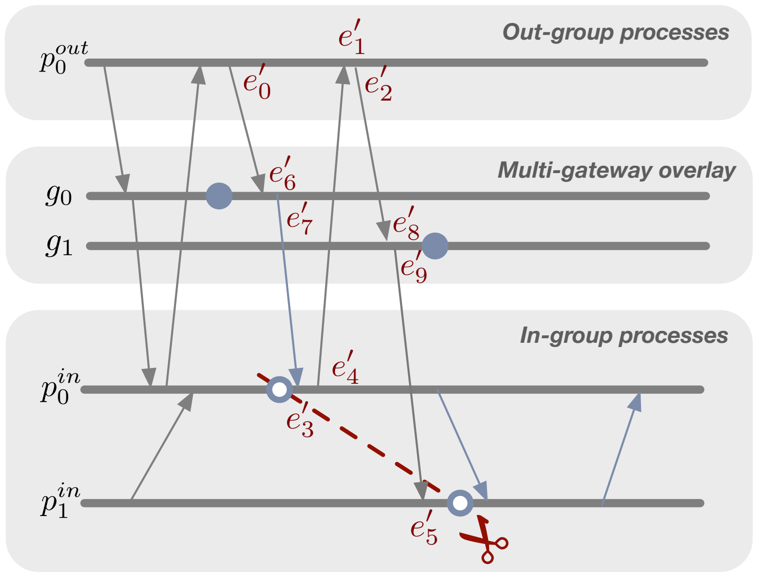


Figure 4.5: An inconsistent partial snapshot using two asynchronous SLBs g_0, g_1. When $e_8'.m$ arrives at g_1, g_1 has not initiated the new snapshot mode to mark the message, thus triggering the violation.

a single service, multiple servers are necessary to handle typical data volumes, load balance among SLBs, and provide fault tolerance.

4.4. Optimistic Gateway Marking (OGM)

Beaver extends gateway marking to practical, distributed environments using OGM. First, to see why asynchronous SLBs could break the consistency guarantee, consider a simple scenario in Figure 4.5 where two SLBs, signaled by an out-of-band controller, initiate a new snapshot. When g_0 initiates snapshot mode and marks $e_6'.m$, it triggers a snapshot at p_0^{in}. However, a new message $e_2'.m$ from p_0^{out} is routed to a different gateway g_1[13], which has not yet entered snapshot mode. This leads to inconsistency: while $e_5' \in C$ and $e_4' \rightarrow e_5'$, $e_4' \notin C$.

To block or not to block? An obvious solution would be to block inbound packets at SLBs during a snapshot and only resume forwarding them after all SLBs have 'committed' to the new snapshot. Unfortunately, this method introduces large overheads—not only to the applications, whose response times will spike while the SLB is blocking requests but also to the cloud providers, where the SLB would require large buffers and overprovisioned capacity to drain said buffers after a snapshot.

[13]This is typical in ECMP routing, where connections even from the same source may reach different SLBs.

Rather than trying to enforce consistency, Beaver seeks a method to (a) detect *in*consistency, (b) reject snapshots when they are potentially inconsistent, and (c) minimize the rejection rate. It seeks to do this with near-zero overhead for applications and cloud infrastructure.

4.4.1. Causal Relevance and Irrelevance

A key idea in Beaver is that, even among the incoming traffic to the in-group, only a subset of that traffic is causally relevant. Using Figure 4.5 to illustrate, an incoming message, m, is **causally relevant** only when (1) an initiated SLB (g_0) sends a marked message to an in-group node (e.g., p_0^{in}), (2) that node interacts directly or indirectly with an out-group node (e.g., p_0^{out}), and (3) that out-group node sends m back to a different in-group node via an uninitiated SLB (e.g., g_1). Other communication patterns, e.g., an m triggered by an uninitiated process, are **causally irrelevant**.

In essence, causally relevant messages are only produced if the message loop: $GW_A \rightarrow IN_A \rightarrow OUT \rightarrow GW_B$ all occurs within the window of time in which the gateways are propagating snapshot initiation. More formally:

Theorem 2. In a system with multiple asynchronous gateways, let the wall-clock time of the first and last gateway initiating snapshots be $e_{gmin}^{ss}.t = \min_{e_g^{ss}}(e_g^{ss}.t)$ and $e_{gmax}^{ss}.t = \max_{e_g^{ss}}(e_g^{ss}.t)$, $\forall g \in G$, respectively. Also let $\tau_{min} = min(d(g, g'; \{p, q\}))$, $\forall g, g' \in G$, $p \in P^{in}$, and $q \in P^{out}$. If $e_{gmax}^{ss}.t - e_{gmin}^{ss}.t < \tau_{min}$, then the partial snapshot is causally consistent.

Proof. We extend the proof of Theorem 1 to a distributed setting. Similar to Theorem 1, there are three cases, with (3) being the one that differs. We again prove it by contradiction.

Assume $(e \in C_{part}) \wedge (\exists e' \rightarrow e)$ but $(e' \notin C_{part})$. As before, there must be some chain $e' \rightarrow e^{out} \rightarrow e^g \rightarrow e$. Because $e' \notin C_{part}$, we have $e_{p_j^{in}}^{ss} \rightarrow e'$ or $e_{p_j^{in}}^{ss} = e'$, that is, p_j^{in} must have been triggered directly or indirectly by an inbound message. Denote the arrival of this inbound message at its marking gateway as $e^{g'}$. By the definition of τ_{min}, we have $e^g.t - e^{g'}.t \geq \tau_{min} > e_{gmax}^{ss}.t - e_{gmin}^{ss}.t$. Thus, at event e^g, the gateway must have already initiated the snapshot and will mark $e^g.m$ before forwarding. This results in $e \notin C_{part}$, a contradiction! □

Theorem 2 implications: Informally, this theorem suggests that if the time gap between the first and last SLB snapshot initiations ($e^{ss}_{gmax}.t - e^{ss}_{gmin}.t$) is sufficiently small, or the minimum time for a message to revisit a gateway (τ_{min}) is long enough, causally relevant messages are impossible and the concerned partial snapshot is provably consistent[14].

Causally relevant messages are rare in the real world. Intuitively and with anecdotal evidence, the inequality $e^{ss}_{gmax}.t - e^{ss}_{gmin}.t < \tau_{min}$ can be satisfied with an exceedingly high probability in real-world contexts:

For the LHS ($e^{ss}_{gmax}.t - e^{ss}_{gmin}.t$): This time gap is essentially the difference in one-way delays between the controller and each of the SLBs. As SLBs share a region with the target service, a well-placed initiator (e.g., equidistant from all target SLBs or one whose messages are forced to travel to the root of the data center fabric) can simultaneously ensure reactive snapshot initiation and $e^{ss}_{gmax}.t - e^{ss}_{gmin}.t$ of near zero.

For the RHS (τ_{min}): This value includes multiple network hops, extending from an SLB to in-group nodes, then to out-group nodes, and back to an SLB. Particularly when out-group nodes are in other data centers or are end-host clients, this value can be orders of magnitude higher than typical values for the LHS—on the order of milliseconds or tens of milliseconds. However, even when the out-group nodes are in the same region or data center as the in-group, we can still expect that this value is higher than any observed delta between initiator-to-SLB one-way delays as it includes *at least three trips* through the data center fabric in addition to processing time at the in/out-group network stacks[15].

4.4.2. Efficiently Verifying Causal Irrelevance

The primary technical challenge of OGM is minimizing the LHS of the above inequality and efficiently/confidently verifying that the resulting inequality held for a given snapshot, even in the presence of message drops, delays, and other sources of unexpected latency. The cloud provider can compute

[14]In principle, another sufficient condition is when the in-group snapshot completes quickly enough. We do not rely on this because it has worse scaling properties than SLB convergence, but it can be added as an optimization.

[15]Even with the detection criteria later described in §4.4.2, LHS entails only 2 trips and encompasses a simpler data path with SLB stacks that are heavily optimized for minimal processing latency and jittering [61, 198, 69, 66].

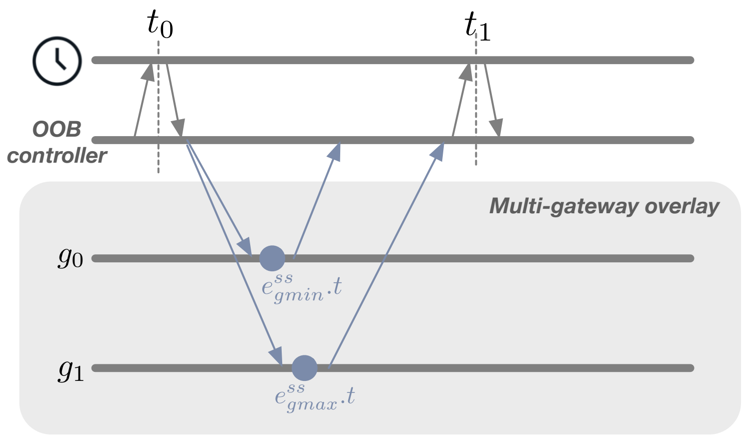


Figure 4.6: The time difference $t_1 - t_0$ as a safe upper bound for $e^{ss}_{gmax}.t - e^{ss}_{gmin}.t$ by querying a single hardware clock source with bounded frequency drift.

the two sides of the inequality separately.

4.4.2.1 Computing a Lower Bound for the RHS

For the RHS, the value can be determined statically, as dynamic network conditions like failures and congestion can only add to the latency of the message sequence. The latency is then equivalent to the sum of each hop's minimum propagation, transmission, and processing delays. These values depend on the relative placements of the in-group nodes, SLBs, and out-group communication partners, but all of those are known at runtime. To ensure a conservative lower bound, operators can and should assume that application-level processing and transmission delays are zero (Figure 4.13).

4.4.2.2 Determining an Upper Bound for the LHS

The LHS is harder to compute statically as failures and congestion mean a true upper bound may not exist[16]. Instead, we need to measure an upper bound online for the observed difference between gateway timestamps ($e^{ss}_{gmax}.t - e^{ss}_{gmin}.t$) for the snapshot in question.

The typical method of measuring time gaps on different machines is via clock synchronization. Although today's clock synchronization techniques can achieve microsecond or sub-microsecond precision, fundamentally, they rely on frequent cross-machine messaging to correct the offset, which is sensitive to congestion and failures, thus impacting the bound on clock drift in the worst case [71,

[16]Beyond the heat death of the universe or at least the life of a data center.

195]. Data center services like TrueTime provide a reliable interface to query time points and calculate their differences. However, a general timing service incurs higher overhead and a typical clock uncertainty range of 1–7 ms [56], much greater than the timescales relevant for Beaver detection.

Synchronization-free approach. Beaver adopts an alternative, customized approach using a single hardware clock to calculate the elapsed time. As depicted in Figure 4.6, the controller queries the start time at t_0 from this clock source with a read t_0^r before initiating a new snapshot. Once the final ACK from the SLBs arrives, it reads the end time t_1^r at t_1 from the source, where t_0, t_1 represents the global wall clock time, and t_0^r, t_1^r the actual clock reads. This hardware clock can be a local hardware clock from either a COTS PCIe NIC [16] or from one equipped with an atomic clock, which are increasingly deployed in production data centers [15, 21].

Note that $t_1 - t_0$ is an upper bound on the LHS as $t_1 > e_{gmax}^{ss}.t$ and $t_0 < e_{gmin}^{ss}.t$. Thus, if $t_1 - t_0 < \tau_{min}$, the partial snapshot under examination is consistent. In practice, the time difference $t_1^r - t_0^r$ is adjusted to account for the maximum frequency drift Δf according to the clock data sheet, to determine an upper bound estimate for the corresponding elapsed time $t_1 - t_0$, thus the detection criteria $(t_1^r - t_0^r) \times (1 + \Delta f) < \tau_{min}$. This method, which relies solely on a single hardware clock to calculate time differences, eliminates issues common in traditional clock synchronization approaches, such as cross-machine message congestion and errors stemming from delays in clock readings due to software interrupts. The frequency drift of a single clock is relatively low and is mainly deterministically affected by temperature, which has low variance in modern data centers [124, 141, 191]. Standard quartz crystal oscillators in production data centers typically drift by ± 100 ppm, or 0.01% error [124, 141, 56, 117, 78]; recent studies are able to reduce this drift of quartz clocks in commodity data center servers to ± 100 ppb (10^{-7} error) by calibrating the offset due to temperature variations. More advanced oscillators (e.g., atomic clocks) can reduce this frequency drift by further orders of magnitude [2, 145] (Table 4.2).

Snapshot invalidation. While ensuring correctness (i.e., no false negatives), our proposed upper bound adds an additional margin to the original time gap. This margin comprises the clock query

	Rubidium	JILA Sr	Quartz	Quartz (calibrated)
Δf	± 0.05 ppb	$\pm 2.1 \times 10^{-18}$	± 100 ppm	± 100 ppb

Table 4.2: Frequency drift (Δf) uncertainty range of today's clocks, ppb (parts per billion) $= 10^{-9}$, ppm (parts per million) $= 10^{-6}$.

latency and the RTT between the controller and the SLBs, which may lead to false positives. In practice, however, we note that many devices support precise hardware timestamping along with the packet data path (i.e., when sending the first notification and when receiving the last notification). Our evaluations on a cloud data center in §4.7 reveal that the resulting snapshot invalidation rate is $< 5\%$ for typical SLB scales today, even in worst-case scenarios when the out-group nodes are in the same data center and under stressed snapshot operation frequencies.

In the end, false positives—while leading to the invalidation of potentially consistent snapshots—are of little concern due to our system's efficient snapshot operations and its ability to achieve a high snapshot rate.

4.5. Beaver's Partial Snapshot Protocol

As mentioned previously, Beaver's snapshot 'quantum' is a single VIP—Beaver can provide snapshots for one or more such VIPs within a single region.

Operation. At a high level, Beaver's partial snapshot protocol distinguishes itself from traditional snapshots in two aspects: (1) its lightweight SLB marking logic for inbound traffic and (2) the snapshot verification process at the controller.

In-group processes: Among in-group processes, Beaver inherits its coordination logic (and the omitted, optional recording of in-flight messages) from prior snapshot algorithms [114, 188] that piggyback 'marker' information per message to handle non-FIFO and lossy channels[17]. Figure 4.7 depicts the core logic: upon receiving a packet, either from an SLB or another in-group process, the current in-group process evaluates if $pkt.sid > csid$. If true, it signals a new snapshot operation: it records

[17]Optional broadcast of marker messages from SLBs to in-group processes may accelerate the snapshot convergence when service traffic is infrequent.

```
• $csid$: Current snapshot ID state for $p \in P^{in}$ or $g \in G$.
– pkt.sid: Snapshot ID ($N$b) in SLB encapsulation header.
– pkt.dst: Destination address of a user packet.
– pkt.src: Source address of a user packet.
1  function IN-OnReceive (pkt):
2      /* Signaled a new snapshot */
3      if $pkt.sid > csid$ then
4          Record the state of interest;
5          Send FIN for $csid + 1, \ldots, pkt.sid$ to the controller;
6          $csid \leftarrow pkt.sid$;
7  function IN-OnSend (pkt):
8      if $pkt.dst \in P^{in}$ then
9          $pkt.sid \leftarrow csid$;
10 function SLB-OnReceive (INIT):
11     if $INIT.sid > csid$ then
12         $csid \leftarrow INIT.sid$;
13         ACK for $csid + 1, \ldots, pkt.sid$ to the controller;
14 function SLB-OnReceive (pkt):
15     /* Mark inbound packet from out-group */
16     if $(pkt.dst \in P^{in}) \wedge (pkt.src \notin P^{in})$ then
17         $pkt.sid \leftarrow csid$;
18     Forward packet to $pkt.dst$;
```

Figure 4.7: Partial snapshot logic (with asynchronous control plane operations) at in-group processes and SLBs.

the relevant state, updates the local $csid$, and asynchronously notifies the controller of completion. For outgoing packets, if the destination address falls within the scope of in-group processes, the process updates $pkt.sid$ to its current $csid$.

SLBs: As discussed in §4.4, Beaver instantiates the gateway overlay with the SLBs. For the set of SLBs handling the target in-group process traffic, Beaver embeds logic for marking inbound messages. On receiving an inbound packet, an SLB first checks if the destination VIP is for the in-group [line 16]—since operators may multiplex a single SLB server for multiple VIPs—and modifies the snapshot ID field accordingly. On the control path, the SLB initializes a new snapshot upon receiving an 'INIT' notification from the controller and subsequently sends the acknowledgment to the controller. This process happens out-of-band to avoid biases in the snapshot verification process. Combined, Beaver's gateway logic requires minimal processing and can be incorporated into existing SLB data planes at line rate, including hardware-accelerated ones.

- • *csid*: The next snapshot ID to initiate at the controller.
- • *receivedFIN*[*sid*][p]: If received FIN from $p \in P^{in}$ for sid.
- • *receivedACK*[*sid*][g]: If received FIN from $g \in G$ for *sid*.
- • $t_0[sid]$: Timestamp t_0 for sid.
- − *FIN.p*: The source process sending the FIN.
- − *FIN.sids*: The associated *sid*(s) of the FIN.
- − *ACK.g*: The source SLB sending the FIN.
- − *ACK.sids*: The associated *sid*(s) of the ACK.

```
function Controller-OnSnapshot():
    num_inflight_ss = $0$, $csid = 0$;
    while num_inflight_ss $< 2^{N-1} - 1$ do
        /* Optional rate-limiting for less greedy snapshots */
        $t_0[csid] = queryClock()$;
        Send $INIT$s ($INIT.sid = csid$) to all $g \in G$;
        num_inflight_ss += 1, $csid$ += 1;
function Controller-OnReceive(FIN):
    for $sid \in FIN.sids$ do
        receivedFIN[$sid$][$FIN.p$] = 1;
        /* Check all FINs received with bitwise negation */
        if $\sim$ receivedFIN[$sid$][$\cdot$] == 0 then
            num_inflight_ss −= 1;
            receivedFIN[$sid$][$\cdot$] = 0;
function Controller-OnReceive(ACK):
    for $sid \in ACK.sids$ do
        receivedACK[$sid$][$ACK.g$] = 1;
        /* If all ACKs received */
        if $\sim$ receivedACK[$sid$][$\cdot$] == 0 then
            if $(queryClock() - t_0[sid])(1 + \Delta f) < \tau_{min}$ then
                /* Accept the snapshot */
            else
                /* Invalidate the snapshot */
            receivedACK[$sid$][$\cdot$] = 0;
```

Figure 4.8: Main controller logic for continuous snapshots.

Controller: With Beaver, operators can designate any server with direct or indirect access to a stable clock source, preferably located near the pertinent SLBs, as the controller. The core logic to initiate snapshots, shown in Figure 4.8, involves continuously sending INIT commands to SLBs to initiate new snapshots. The protocol maintains the number of snapshots in flight and controls the snapshot frequency. The detection of invalid snapshots follows the methodology outlined in §4.4.2: The controller queries the clock read for t_0 before sending notifications [line 5] and uses the clock reads upon receiving the last ACK to determine the snapshot's validity [line 20]. It the local NIC supports hardware time-stamping capabilities, `queryClock()` can occur along the data path during the send of the first INIT notification and the receive of the last ACK response.

Handling packet loss, delay, and reordering. Beaver is robust to faults in data- and control-plane communications.

Data plane: Unlike the original Chandy-Lamport protocol, which relies on separate marker messages, Beaver draws inspiration from subsequent variants [114, 188] to incorporate marker information by piggybacking it into existing traffic. This piggybacking makes Beaver inherently resilient to 'marker' losses and reordering on the data path, whether these occur within the network core or the host networking stacks.

Control plane: Although timely and reliable delivery of control messages can be beneficial (e.g., through an alternate port that is dedicated to control tasks) Beaver does not depend on it for its core functionality. It operates effectively even with unreliable transport protocols such as UDP and it requires only a negligible number of control messages: $|P^{in}|$ FIN messages (or less as members of the in-group, P^{in}, can batch updates in a single ACK on the increments in prior snapshots), $|G|$ INIT commands, and $|G|$ ACK responses for each snapshot.

While delays or losses of the above messages might slow down the snapshot rate—a minimal impact as observed in our evaluation—they do not compromise the correctness of Beaver. The controller, in response to any delays or losses, simply invalidates the affected snapshot.

Handling failures. One important problem is how to handle failures of the SLBs and backend servers. Fortunately, most public clouds today already apply central management mechanisms that ensure fault tolerance and state consistency during changes in membership of machines for each VIP[18][61, 147, 45, 69]. Operating on top of the abstraction, Beaver's controller coordinates with the SLBs and backend servers belonging to the requested VIP (as indicated by the current central state), incurring minimal additional costs and deployment complexity. To handle failure events during a snapshot, Beaver incorporates a single ACK mechanism (Figure 4.8): if the controller does not receive the ACK from an SLB or an in-group process, Beaver simply invalidates the snapshot or drops affected states while guaranteeing correctness.

Supporting parallel snapshots. Many cases, such as event-driven or telemetry tasks, require higher-frequency state capture [188, 193]. Rather than waiting for the completion of one snapshot before initiating another, limiting the snapshot rate to the slowest component in the snapshot convergence process, Beaver can initiate snapshots concurrently. The controller ensures that the number of packets in flight remains within $2^N - 1$ [line 3 in Figure 4.8], the maximum concurrent snapshots supported by the header field sid. The extra $^{-1}$ in the exponent is to eliminate ambiguities in comparator operations at in-group processes [line 3 in Figure 4.7] under worst-case wrap-around conditions.

Beaver also supports parallel snapshots for distinct groups of VIPs without needing extra metadata. This is facilitated by the SLBs' ability to naturally segregate operations based on VIP information. Consequently, the same sid header space can be utilized for simultaneous snapshots across groups with non-overlapping VIPs.

4.6. Implementation

We implement a Beaver prototype on a cloud data center [59] (Figure 4.9) that aligns with a production setup [147, 18, 61].

Supporting SLB-associated functionalities. We implement an end-to-end workflow to mirror the behaviors associated with SLBs in production data centers [61, 18, 147]. Additionally, our system

[18]Unlike the DIP caching feature in §4.2.1, the consistency mechanism was originally absent in [147], but later incorporated as an essential component.

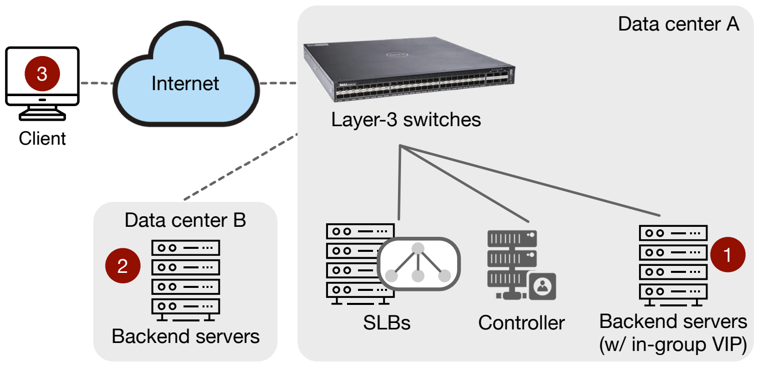


Figure 4.9: Evaluation setup considering three different out-group locations: within the same data center, data center of a different region, or on the Internet (from a local laptop).

facilitates automated service discovery operations through an out-of-band controller server.

SLB implementation: Our setup configures DELL EMC PowerSwitch S4048-ON [17] for layer-3 ECMP forwarding based on service VIPs to SLBs. Emulating prior work [61, 18], we implement the core SLB functions with DPDK [7], involving around 1860 lines of C/C++ code. Each SLB maintains an in-memory connection flow state, employs consistent hashing on the 5-tuple of each packet to determine the appropriate backend server, and caches the decision for future decisions. Then, the SLBs encapsulate the inbound packet's header and forward it to the backend server with the destination DIP. To maximize utilization of SLB servers, we perform load balancing across different CPU cores using RSS.

Backend servers: To maintain transparency for the upper-layer applications, we implement the recomputation of checksums, NAT caching in a shared eBPF map, and the de-encapsulation of incoming packets from the SLB via XDP [84]. For outbound packets, we instrument the Linux `tc` to look up the NAT entries and perform the header transformations to replicate Direct Server Return (DSR). In total, they involve 1040 lines of C/C++ code.

Topology. Our testbed supports typical communication patterns, encompassing a variety of out-group positions, including other VIPs within the same data center, VIPs in other data centers, and Internet clients—all through the layer-3 switches and SLBs, along with DSR on the return path. We scale up to 16 SLB servers, each capable of supporting 64 in-group processes, due to limits in resource availability. Our current testbed servers are equipped with Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz and dual-port ConnectX-4 Lx NICs.

Integrating the Beaver protocol. We implement Beaver's partial snapshot protocol from §4.5. The SLBs append a snapshot ID to inbound packet headers that encapsulate the destination DIP and the source SLB IP. The in-group processes and SLBs embed Beaver's snapshot logic from Figure 4.7 through XDP and DPDK. The additional logic involves 68 lines of C++ for SLB data-path logic and 102 lines of C codes for eBPF at in-group processes. The controller server, following Figure 4.8, automates the initiation, control, collection, and verification of snapshots. We use UDP for bi-directional control messages with SLBs and unidirectional messages from in-group servers. The controller currently exploits local NIC hardware timestamping (`SOF_TIMESTAMPING_RAW_HARDWARE`) for precise timing of INIT and ACK messages on their data path [20].

4.7. Evaluation

Our evaluation focuses on exploring the following questions.

- Can Beaver sustain fast snapshot rates? How does the scale of the in-group nodes and SLBs affect? (§4.7.1)
- What about effective snapshot rates? How often do Beaver invalidate snapshots in cloud data centers? (§4.7.2)
- Does Beaver's distributed coordination affect the existing service traffic? (§4.7.3)
- How does Beaver help real-world services? (§4.8)

4.7.1. Beaver Supports Fast Snapshot Rates

To stress-test Beaver, unless otherwise specified, our evaluation runs Beaver at very high snapshot frequencies. To further ensure that our performance/overhead results are conservative, state capture in the snapshots are NOPs. Real local record operations (which are application-dependent and orthogonal to the study of distributed snapshot protocols) will only result in less contention and overhead.

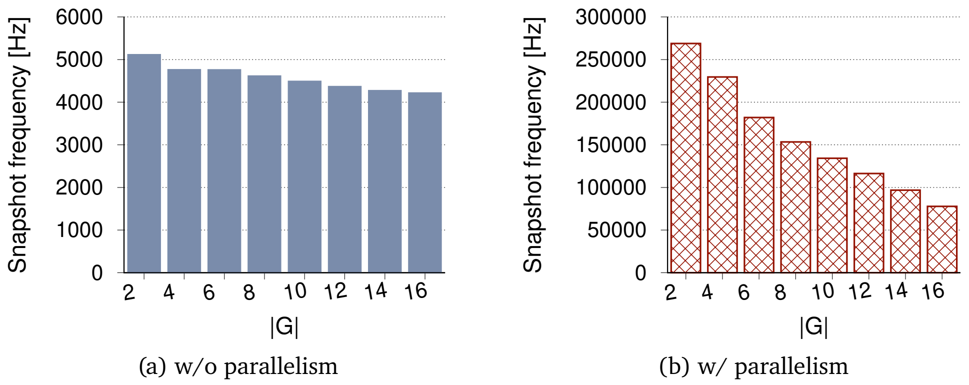


Figure 4.10: Beaver's sustained snapshot frequency versus a strawman approach with blocking operations at varying scales of SLBs and backend processes.

As a measure of Beaver's efficiency and scalability, even at these high rates, Beaver exhibits good performance. Figure 4.10 shows the maximum snapshot rate compared to a strawman approach, which waits for completion before initiating another. The maximum rate is determined by increasing the snapshot frequency until we observe backlogs in the ACK and INIT message notification queue. We vary the number of gateways ($|G|$) up to 16, aligning with typical values for SLBs assigned to a VIP.

The baseline is limited by the snapshot convergence time, which depends on factors such as scale, traffic pattern, and topology. In contrast, Beaver's parallel snapshot capability significantly enhances the rate and shifts the bottlenecks to the processing power of the controller's CPU. Even at the maximum scale, Beaver reaches a snapshot rate of > 77000 Hz, $> 18\times$ that of the strawman. In practical applications, leveraging a more powerful processor or scaling the controller server could further improve its speed.

4.7.2. Beaver Invalidates Snapshots Infrequently

With a high snapshot frequency, how does Beaver perform in terms of effective snapshot rates? Recall in §4.4.2, Beaver uses an upper bound $t_1 - t_0$ for the time gap between SLB initiations ($e^{ss}_{gmax}.t - e^{ss}_{gmin}.t$) to eliminate the need for time synchronization, it invalidates a snapshot if the bound is greater than τ_{min}, the minimum time to for an external causal chain to occur. While this upper bound ensures correctness, it may reject snapshots and reduces the effective snapshot rate.

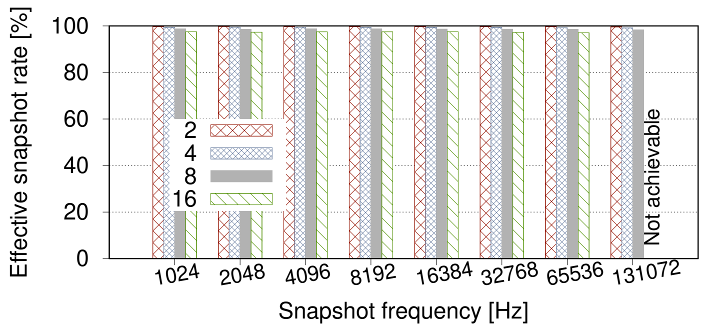
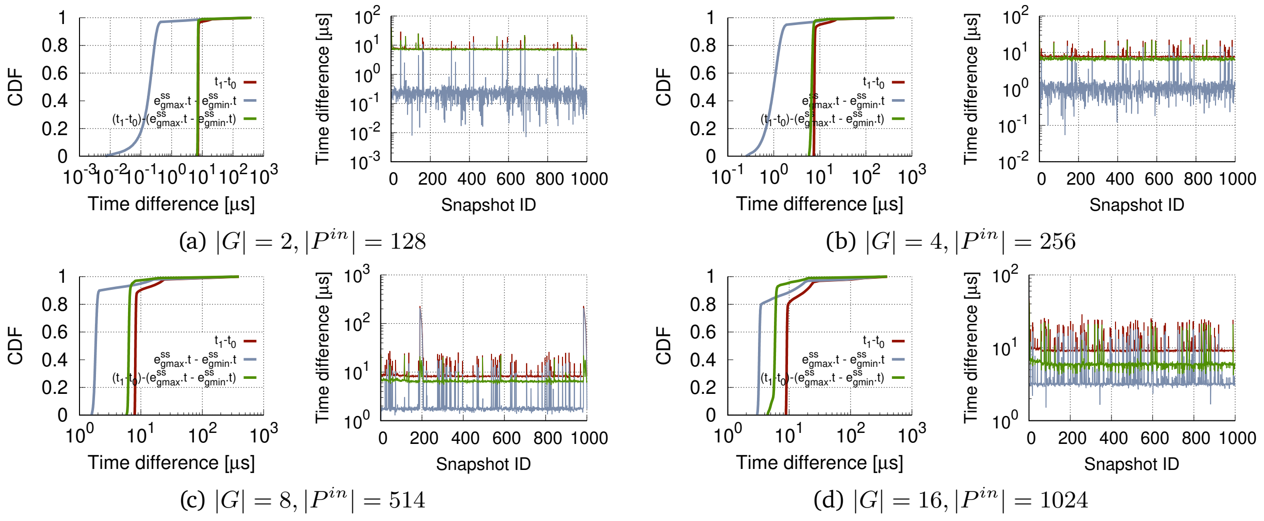


Figure 4.11: Beaver's effective snapshot rates under varying snapshot frequencies and in-group process scale.



(a) $|G| = 2, |P^{in}| = 128$

(b) $|G| = 4, |P^{in}| = 256$

(c) $|G| = 8, |P^{in}| = 514$

(d) $|G| = 16, |P^{in}| = 1024$

Figure 4.12: CDF of Beaver's upper bound $t_1 - t_0$ with the ground truth $(e^{ss}_{gmax}.t - e^{ss}_{gmin}.t)$ for > 10M snapshots and a zoom in to its snapshot series, under stressed scenario with 65536 Hz snapshot frequency and varying number of SLBs/processes.

To measure the time for an external causal chain to occur, we consider three distinct scenarios for out-group process locations in Figure 4.9. In each scenario, we set up a worst-case condition where, immediately following an SLB's snapshot initiation, the SLB forwards an inbound packet to the closest in-group node. The in-group node then loopbacks an immediate message to out-group node with the shortest path, which bounces the packet back to any SLB. Figure 4.13 shows that the intra-DC scenario results in the shortest time window, resulting in τ_{min} as 33μs. This value is robust because, even though varying cloud conditions often cause latency spikes, they primarily affect the tail rather than the minimum.

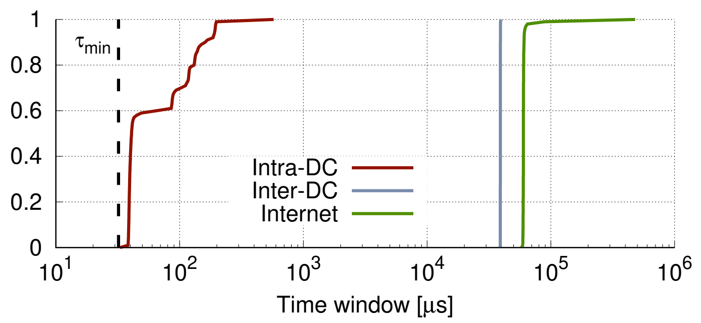
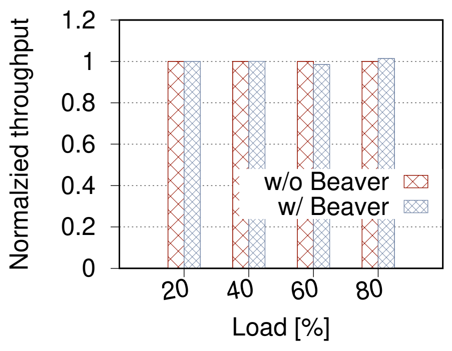


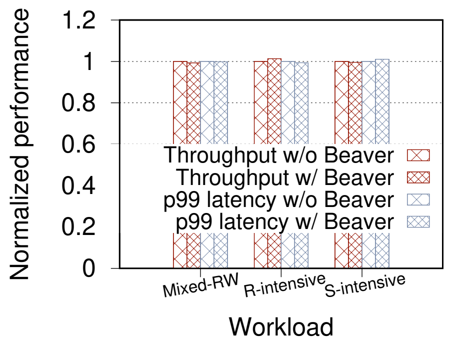
Figure 4.13: Measurement of the minimum time window for a external causal chain to occur under worst case conditions.

To stress test Beaver's performance, we focus on the worst-case scenarios with out-group processs located within the same data center. For other scenarios, τ_{min} is significantly greater, leading to 100% effective snapshot rates across 10M snapshot operations. We execute Beaver in various experimental settings, including scale and snapshot frequencies. For each configuration, we calculate the effective rate based on more than 10 million snapshots. The results, as in Figure 4.11, reveal that the proportion of snapshots invalidated by Beaver is remarkably low even under the maximum operating frequencies and scales of our testbed.

To better understand the results, we compare the recorded upper bound estimation of $t_1 - t_0$ with the true ground truth $e^{ss}_{gmax}.t - e^{ss}_{gmin}.t$. As the two events e^{ss}_{gmax} and e^{ss}_{gmin} occur on separate SLB machines, we synchronize the clocks of all SLBs to controller's PTP master clock over symmetric paths without contending traffic, which reports maximum 50 ns offsets during the ground truth measurement. This step, meant solely to understand the behavior, should not be confused with Beaver's clock-synchronization-free approach. Figure 4.12 shows the comparison over > 10M snapshots when Beaver operates at a frequency of 65536 Hz. Overlapping tails of $e^{ss}_{gmax}.t - e^{ss}_{gmin}.t$ and the heads of $t_1 - t_0$ are expected—the cdf of the pairwise calculation of $(t_1 - t_0) - (e^{ss}_{gmax}.t - e^{ss}_{gmin}.t)$ for each snapshot clearly demonstrates that the upper bound is strictly higher than the ground truth SLB initiation time gap. The observed outliers in $t_1 - t_0$ are typically due to queueing in our manager's processing queue at high rates or asynchrony in SLB initiations. Furthermore, the margin introduced by $t_1 - t_0$ over $e^{ss}_{gmax}.t - e^{ss}_{gmin}.t$ is due to the RTT between the controller and the SLBs, which is used to ensure the theoretical upper bound without clock synchronization.



(a) Stressed workloads



(b) YCSB benchmarks

Figure 4.14: Performances with and without Beaver's overhead, normalized to the value without Beaver.

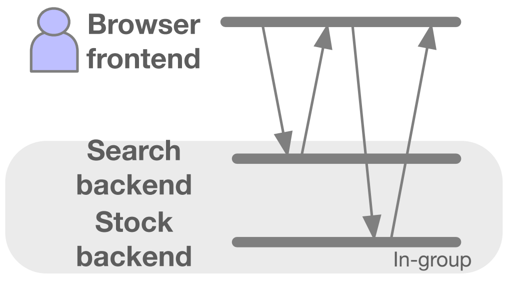
4.7.3. Beaver Incurs Near-zero Impact

We also stress test the overhead of Beaver on user traffic. Figure 4.14a compares throughput with and without Beaver under the 65536 Hz snapshot frequency and the max scale of our testbed. `iperf` clients send traffic with varying degree of the total consumed bandwidth capacity of the 16 SLBs. We also run YCSB benchmark workloads [14] with varying mix of read, update, and scan operations, as shown in Figure 4.14 for backend servers running CassandraDB [13]. The requests follow zipfian distributions, and the scan length adheres to the uniform distribution.

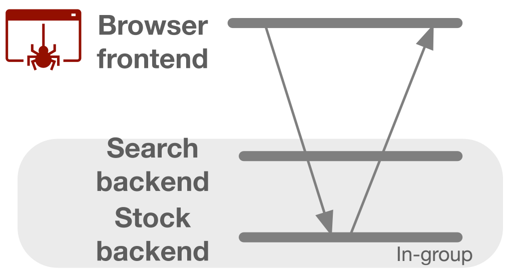
The results of various performance metrics are almost identical, confirming that Beaver has a near-zero effect on service traffic. This is because Beaver, by design, eliminates any delay or blocking operations on the data path for distributed coordination, and the lightweight control path messages are orthogonal.

4.8. Use Cases

We also examine several use cases of Beaver. These examples are intended as instruments through which we can understand its potential utility, differences versus traditional snapshots, and the semantics of its causal consistency guarantee under partial deployments that were previously impossible.



(a) Benign



(b) Bot access

Figure 4.15: Example benign and bot access patterns.

4.8.1. Detecting Anomalous Access

Web applications often feature a JavaScript browser frontend for user interaction and a backend providing service APIs. Consider a legitimate user access in an e-commerce application (Figure 4.15a). The frontend calls a Search API `fetch("example.com/api/v1/search")`, followed by a Stock API `fetch("example.com/api/v1/get_stock")` for product details. However, malicious traffic, such as web scrapers, might bypass the initial search stage and directly query the stock backend, potentially overwhelming the server. This type of traffic can be challenging to detect as it differs from legitimate traffic in intent rather than content [100, 64].

Beaver can help detect such anomaly patterns, as its partial snapshot can capture the external dependency of these requests, even though it occurs through communication with the Internet. To illustrate, we run a varying mixture of benign and illegal bot clients on our testbed. The backend servers maintain per-client request count in a BPF map through double buffering, so as to 'freeze' the current state through a single switch of the pointer and minimize the impact of blocking local record calls. Table 4.3 shows detection results calculated against the ground truth. We find that Beaver can accurately recognize the interdependence between the accesses. For example, when all clients are benign, Beaver consistently results in true negatives, aligning with the ground truth. However, a polling-based approach and traditional snapshots (L-Y) can result in false positives due to interpretations of erroneous capture of higher counts at the Search backend than at the Stock backend.

Method	Bot ratio = 0% TP, FP, TN, FN	Bot ratio = 5% TP, FP, TN, FN	Bot ratio = 10% TP, FP, TN, FN
Polling	0, 0.005, 0.995, 0	0.005, 0.062, 0.874, 0.059	0.069, 0.136, 0.666, 0.129
L-Y	0, 0.005, 0.995, 0	0.001, 0.058, 0.886, 0.055	0.011, 0.105, 0.783, 0.101
Beaver	0, 0, 1, 0	0.053, 0, 0.947, 0	0.113, 0, 0.887, 0.001

Table 4.3: Beaver's detection accuracy versus (1) polling-based approach using time synchronization, and (2) Lai-Yang algorithm, a state-of-the-art global snapshot protocol.

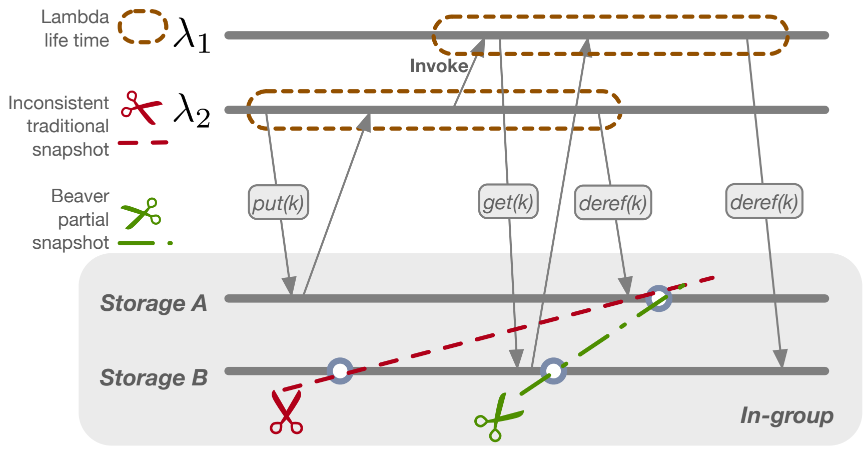


Figure 4.16: Garbage collection for the ephemeral storage for serverless analytics.

4.8.2. Serverless Garbage Collection

Backend services that support serverless applications are also a natural fit, as requests to serverless functions rely on schedulers and logic that are not visible to the backend services or the serverless functions themselves. Consider an application that provides storage for a serverless analytics job and uses reference counting for garbage collection [111]. The storage service deploys multiple servers for scalability and supports three primary APIs: `get()`/`put()`, which fetch/upload the object and increment the reference counter, and `deref()`, which indicates that the previously fetched object is no longer in use and decrements the reference counter.

Beaver's consistent partial snapshots can support safe garbage collection decisions. To illustrate, we instantiate two serverless functions through [97] that follow the workflow of Figure 4.16 on our testbed. The backend storage maintains an in-memory state of reference counters for each KV object. When a reference counter reaches 0 in a snapshot, the controller informs the backends to recycle the

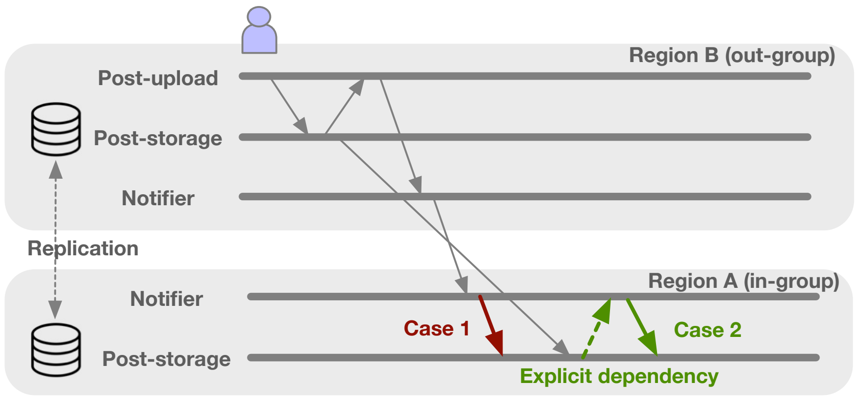


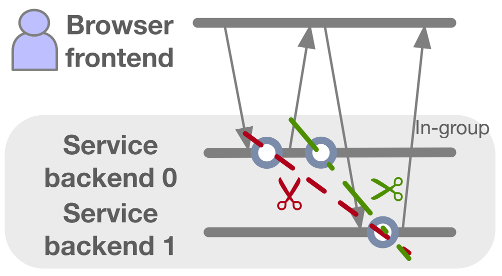
Figure 4.17: A simplified example of geo-distributed social media application [65] which includes distinct services such as post-upload, post-storage, and notifier.

corresponding object. During invocations, we also record the incident counts of invalid `get()` access or `deref()` calls. We find that, across invocations, the Lai-Yang algorithm may produce inconsistent snapshots (shown in Figure 4.16) that indicate *no* open references to the object—however, λ_1 is still keeping a reference to it. This leads to unsafe decisions to recycle the object associated with the key and results in an observed invalid call percentage of 23–29%. In contrast, Beaver's partial snapshots guarantee causal consistency even in the presence of external communication that ensures safe reclamation of the object and consistently results in 0 invalid calls.

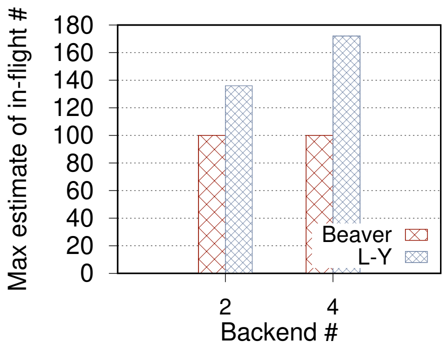
4.8.3. Integration Testing

Integration testing, commonly used in CI/CD pipelines [75], extends the coverage of testing to inter-service logic. Unfortunately, applying it to distributed applications can be challenging. Consider the example shown in Figure 4.17, a violation of the application specification occurs when followers in a region receive a notification and request the storage DB (case 1) before the cross-region protocol actually replicates the post data. Recent solutions [65, 165] address the inconsistencies by forming explicit dependencies (case 2). However, the involvement of auxiliary services and additional dependencies make it difficult to capture a holistic snapshot.

Beaver offers a practical abstraction to test distributed applications by enabling partial deployment and capturing causal dependencies relevant to the local service. By snapshotting states in post-storage and notifier services, developers can write test cases to verify the crucial invariant above: the



(a)



(b)

Figure 4.18: (a) Example snapshots for in-flight message tracking. (b) Comparison of estimated number of in-flight requests with and without Beaver.

presence of a post in the storage must always precede its corresponding notification in the notifier service. In particular, Beaver's guarantee of causal consistency means that if a canary solution is correct, a partial snapshot observing a log in the notifier must have captured the data entry of the corresponding version in post-storage. Therefore, a single violating test case will suggest the presence of bugs.

4.8.4. In-flight Message Tracking

We also revisit the example in Figure 4.3. As mentioned in §4.2.2, a useful query is to estimate the number of concurrent requests, which can inform resource provision decisions. Figure 4.18a illustrates a scenario with only one active request. In theory, traditional snapshots, which fail to capture the causality between the client's follow-up request and the prior response, can give an overestimation of 2 in-flight messages (indicated by the cut in red). Beaver, in contrast, can capture the external causality and results in an estimation of no more than 1 message in flight (indicated by the cut in green).

To validate the behavior in practice, we run 100 clients concurrently that conform to the poisson arrival pattern on our testbed. Each backend process maintains a total request and response count value using a BPF map. Thus, the difference between the two counters indicates the number of messages in flight. The controller then collects the snapshot of counter values and then obtains the aggregate estimate. Figure 4.18b shows that traditional snapshots can overestimate the number of

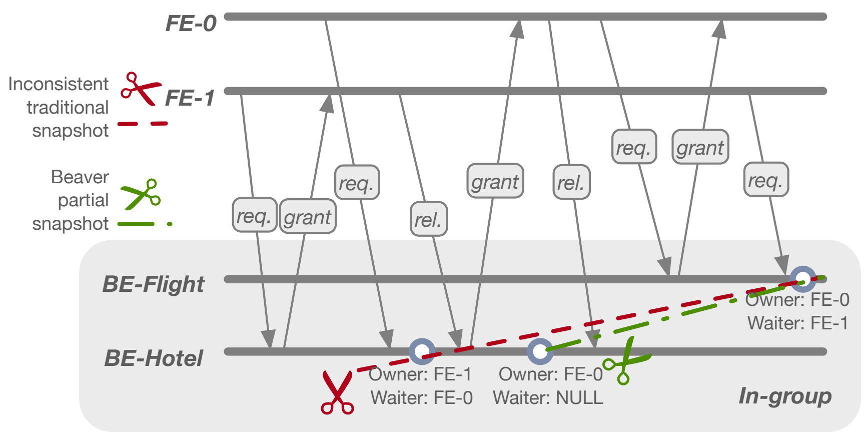
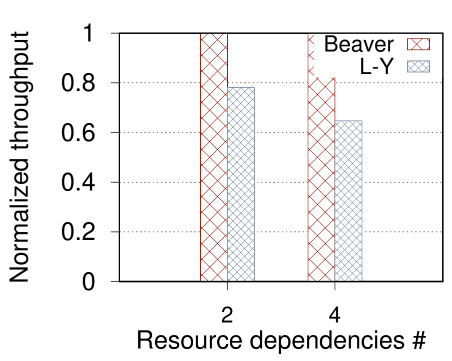
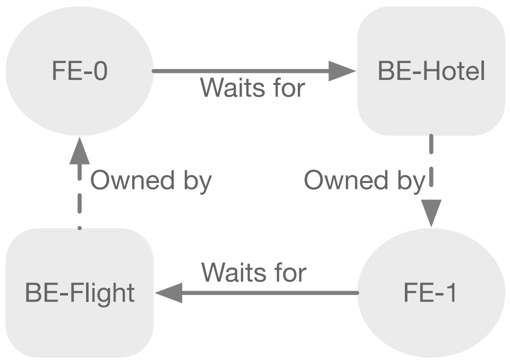


Figure 4.19: An example of deadlock detection using distributed snapshots.



(a)



(b)

Figure 4.20: (a) Comparison of transaction throughput (normalized to Beaver). (b) WFG for the inconsistent snapshot in Figure 4.19.

concurrent requests by more than 30%, while Beaver's result consistently matches the ground truth. Worse, a higher number of backends will lead to an overestimation further divorced from reality.

4.8.5. Distributed Deadlock Detection

A classic use of distributed snapshots is deadlock detection, a fundamental problem in distributed systems. Consider the scenario in Figure 4.19, where the machines of a frontend service interact with a reservation microservice to book flights and hotels on behalf of its clients. Here, a frontend server acquires a lock from the backend server for a target resource ID and releases it after completing its transaction. A deadlock may occur when a client requests resources that are held by others, forming a directed cycle in the resource dependency graph (known as Wait-For Graphs or WFGs). As these systems (such as those used by Airbnb and Uber) encompass thousands of microservices, each with its own sovereignty [206, 68], global snapshots are challenging and expensive to enforce.

Beaver, however, is amenable to only taking partial snapshots of the reservation service. To illustrate, we run backend processes that maintain the ID list of client(s) currently owning/waiting for the local resources in memory. When the controller detects a deadlock based on a snapshot, it informs backend processes to abort the current transaction. We emulate clients that request backend resources in random order and measure the resulting transaction throughput. Figure 4.20a shows that the traditional snapshot algorithm can suffer from more than 20% throughput drops compared to Beaver. This is because, without accounting for the external message dependencies, it can render a snapshot that is inconsistent (Figure 4.19), which leads to false deadlocks (Figure 4.20b) and the unnecessary costs of deadlock resolution operations. Beaver, on the other hand, guarantees safe detection.

4.9. Discussion

Instantiating Beaver gateways. Beaver focuses on public clouds, which already contain SLBs, imposing minimal changes and costs to integrate its functionality. We argue that these are where partial snapshots are most important as smaller private clouds are easier to modify wholesale [157]. Without cloud providers' support, cloud tenants could also deploy their own Beaver-compatible gateways on virtual machines (e.g., Network Virtual Appliances (NVAs) [24]) to ensure consistency under external communication with clients and human users. This involves additional costs and complexities and can be suitable if NVAs are already in use, e.g., to provide firewall functionality.

Optimizing local record operations. Similar to classic distributed snapshot protocols (§4.2.2), Beaver is agnostic to the semantics of local record operations. An interesting problem—orthogonal to the core mechanism of Beaver—is to enable efficient local-state capturing mechanisms, especially when the user desires a large target state or a high snapshot frequency. Besides application-specific practices in §4.8, we postulate that a more generic and opportunistic approach may minimize their online impacts by focusing on state changes during IDLE times of the application. We leave a complete exploration for future work.

4.10. Related Work

Distributed snapshots. This work builds on the large array of classic distributed snapshot algorithms [51, 114, 136, 188, 181, 112, 83, 183]. To the best of our knowledge, Beaver formalizes, designs, and implements the first partial snapshot primitive that extends their capabilities for practical usage.

Cloud data centers. Beaver is also related to works on various facets of cloud data centers, including layer-4 load balancers [147, 198, 61, 138, 69, 45] and its clock services [56, 124, 195, 141, 191, 71, 15, 21, 117, 78]. For the former, Beaver integrates its gateway marking logic based on the behaviors of SLBs fundamental to cloud data center services and implements a practical prototype aligned with today's setups. Meanwhile, Beaver builds on extensive measurement studies that highlight the reliable properties of frequency drifts of a single clock. Combined, Beaver presents a unique design without making any assumptions about clock synchronization that ensures consistent, high-rate partial snapshots under external interactions while incurring minimal changes and impacts to current operations and service traffic.

4.11. Conclusion

This chapter rethinks the classic distributed snapshots and observes the mismatch of their assumptions with today's cloud services. With it, we present Beaver, the first partial snapshot primitive that advances the capabilities of existing snapshots for practical usage in distributed cloud services. Central to Beaver is the design and instantiation of a novel optimistic gateway marking primitive. Beaver presents a unique design point by tightly integrating the protocol with the regularities of data center networks. Our evaluation demonstrates that Beaver not only can capture partial snapshots at high speed, but it also incurs near-zero costs to existing service traffic.

CHAPTER 5

CONCLUSION

Waste is a tax on the whole people.

Albert W. Atwood

This dissertation characterizes zero-waste designs (§1.2 and §1.3) and presents three instantiations for control and monitoring functions (Chapter 2, 3, and 4) that play a crucial role in supporting user applications: (1) Beaver, which reduces additional overhead from the outset when extending network management capabilities; (2) OrbWeaver, which reuses idle Ethernet cycles that would otherwise go wasted for high-resolution in-band control protocols; and (3) Mantis, which recycles and couples underutilized on-board CPU cores and the PCIe channel for localized, sub-RTT reactive transactions. Central to our approach is the harvesting of in-network waste, enabling us to sidestep the traditional trade-off between performance and cost. These designs demonstrate that it is possible to integrate these functions at near-zero cost while maintaining their efficacy.

The dissertation research also includes other collaborative efforts on zero-waste designs. Examples of works already published (listed in §1.4): (a) InvisiFlow extends OrbWeaver's capability to dynamically identify and reuse idle capacity in the whole network, thereby minimizing both the loss rate of telemetry data and overheads on user traffic; (b) Cebinae simplifies existing mechanisms to mitigate unfairness in public networks for a scalable design compatible with a diverse range of end-host transport protocols; (c) PrintQueue minimizes the SRAM resources with novel data-plane data structures to accurately track the provenance of packet-level delays for diagnosing performance anomalies; and (d) Cowbird exploits in-network resources such as SmartNICs, spot VMs, and programmable switches to offload the control of all network communication, thereby saving compute nodes' CPU tax overheads for application threads in the memory disaggregation paradigm.

In the broader context, the impetus for this thesis is twofold: the relentless increase in application demands, and the technology scaling slowdowns characterized by the plateauing of Moore's law and Dennard scaling. Propelled by these factors, coupled with the pressing environmental concerns related to energy consumption and carbon emissions in our industry[19], this thesis envisions zero-waste networked systems, where we adhere to zero-waste designs (§1.2) to maximize the utility of the resulting network from high-efficiency designs. More broadly, we advocate for addressing the grand challenge of pushing waste in our computing infrastructure to their limits.

Before concluding, we summarize the lessons learned from practicing zero-waste designs (§5.1) and outline a series of open questions to further this endeavor (§5.2).

5.1. Lessons Learned

This section shares the insights learned from exploring zero-waste designs.

Wastes in networked systems take many forms. Wastes can manifest in various forms across spatial network components and time scales. For instance, in OrbWeaver, idle cycles appear not only in familiar diurnal patterns but also in microsecond granularities. One crucial lesson is to constantly seek out and identify waste. Often, what is overlooked becomes a design blind spot, and harnessing it can reveal the sweet spot in the trade-offs between performance and costs. In fact, upcycling waste not only has practical utility but also facilitates cost-effective solutions, as demonstrated in the Mantis case, which enabled the fast deployment of its agent in large-scale production networks (§1.3).

Waste is often abundant and its absence can be also informative. One surprising finding is that, while periods of no idle resources do occur, such conditions are rare when considering both time *and* space dimensions, as shown in InvisiFlow. Even so, the absence of idle resources itself provides concrete information, such as indicating congestion for idle Ethernet cycles. Moreover, even for background functions requiring strict service level objectives (SLOs), these resources can be utilized opportunistically under common conditions, combined with practices such as resource priority escalation for worst-case scenarios.

[19]Annual ICT energy demand is projected to exceed 100 exajoules over the next decade, reaching approximately 15% of global energy production, with electricity use growing faster than renewable energy capacity [79, 116].

Zero-waste designs have wide conceptual extensions[20]. Zero-waste designs go beyond practices such as reusing underutilized resources and should be tailored to specific system requirements and constraints. For instance, sometimes it can be more beneficial to trim the waste itself, e.g., by reducing power consumption using power-saving techniques or offloading tasks to more efficient hardware. Incorporating waste reduction considerations early in the design process is also valuable, as demonstrated in Beaver, which reduces the need for server hardware procurement. More broadly, the concept of zero-waste encompasses a wider range of practices beyond the three-Rs presented, such as the five-Rs in the waste management hierarchy and more.

In addition, the applications of zero-waste designs extend beyond control and monitoring tasks. Auxiliary tasks associated with application requests also add to processing overheads, as in the Cowbird case, because they often fall outside user or application concerns. Pushing these 'taxes' toward their lower bounds, possibly engaging idle channels while preserving performance, is thus beneficial.

5.2. Future Directions

With the relentless increase in application demands, technology scaling slowdowns, and growing environmental constraints, it is particularly crucial to explore methods to push toward systems with minimal waste. We highlight several directions to improve the visibility of waste, innovate design practices, and address practical implications toward the vision.

Augmenting the visibility into 'wastes' from their cradle to their grave. To minimize waste, we need a comprehensive understanding of their patterns to gauge the distance to the ideal and evaluate the efficacy of a design. Current ad-hoc measurements provide a glimpse, but a holistic, granular, and synchronous view is lacking. This is particularly pertinent in today's heterogeneous networked systems, where resource use is multi-dimensional. A clearer perspective would help us understand the interplay among diverse resources such as compute, storage, and networking, especially with network speeds reaching the terabit scale.

Additionally, today's evaluation of computing systems often focuses on their performance, with en-

[20]The set of objects satisfying the properties (intension) connoted by the concept.

ergy use or environmental impact receiving less attention. With a growing emphasis on reducing carbon footprints, it is crucial to develop methods for complexity analysis and carbon attribution in multi-tenant environments. This would guide informed and environmentally responsible design decisions.

Exploring designs to minimize waste for the emerging workloads and contexts. While this dissertation focuses on zero-waste designs, minimizing waste for emerging applications remains a problem. Traditional layering principles, while effective in separating developer concerns, can introduce inefficiencies by imposing fewer assumptions about the specifics of the underlying components. Thus, it is appealing to simplify current system stacks, adhering to Occam's Razor principles. Emerging elephant workloads, such as machine learning, present not only an opportunity to customize these stacks but also a dire need to reduce their footprint, as the exponential growth in ML resource demand outpaces the growth in total resource capacity, such as power. Similar practices also have great potential for microservices, where the curse of generality has also been noted for their sidecars. A question is what does a clean-slate, 'post-layering' architecture—that reduces the system stack footprint while maintaining functional modularity—look like?

The notable trend of faster networking speeds makes data movement to interconnected devices cheaper and enables new paradigms such as memory disaggregation. Exploring new practices, such as process migrations, is also promising. Moreover, while our focus has been on core data center networking, emerging edge clouds also present vast opportunities, as user traffic patterns in these environments are less predictable and more difficult to multiplex.

Navigating through real-world implications of zero-waste designs. Zero-waste designs bridge the gap by minimizing residual waste from applications. However, operating at full utilization can imply potential unexpected behaviors, such as those caused by increased heat generation or hardware bugs rarely encountered today. For example, adaptive cooling systems can fail, resulting in thermal throttling and degraded performance. Although modern processors and switching devices provide sufficient thermal margins, increased utilization increases the risk of violations. Additionally, security concerns, such as attacks on cooling systems, must be considered, even though they are orthogonal to utilization. These reliability considerations are crucial when pushing toward zero-waste systems.

Another implication of zero-waste designs is to extend the lifespan of devices when considering embodied carbon, an increasing contributor to carbon emissions in the industry. Such an emphasis of embodied carbon can change the inflection point of trade-off decisions; for example, simply discarding old components becomes more wasteful—since the embodied carbon arises from the manufacturing process, these costs remain even if the devices are left idle. Consequently, it becomes important to handle more heterogeneous hardware, which introduces challenges such as increased failure rates when devices are kept in use longer than they are today.

APPENDIX A

ORBWEAVER

A.1. Applications of OrbWeaver

Table A.1 surveys 11 applications that can benefit from an OrbWeaver implementation, belonging to four distinct classes. We describe several implementations in Section 2.4. All applications can be expressed as OrbWeaver P4 programs with the basic architecture shown in Figure 2.8.

Across all applications, we find that there are two overarching benefits to an OrbWeaver implementation:

1. OrbWeaver's weaved stream allows data plane applications to infer information about network conditions, such as the presence of congestion or failures in an upstream path.

2. OrbWeaver's IDLE packet abstraction lets data plane applications disseminate information without consuming user bandwidth. IDLE packets are useful for data transfer between directly connected switches (e.g., to synchronize the context tables of a switch-to-switch packet-header compressor [174]) or across the wider network (e.g., to disseminate information about network faults [129], congestion [204], or even user query metrics [143]).

We note that our focus of these applications and this paper is in-network communication. However, end hosts may also be able to benefit from OrbWeaver, e.g., by examining the output of the weaved stream coming from host-facing ports of ToR switches. Efficient end-host generation of a weaved stream may also be possible, but we leave a full exploration to future work.

A.1.1. Balancing Multiple Applications

IDLE packets are generated and weaved entirely by the OrbWeaver framework. Applications only embed information and extract it in the receiver. IDLE packets can carry the information of multiple applications. For example, a time synchronization application that needs 12B to carry 4 timestamps can co-exist with a failure detection protocol that needs 48B. In this paper, we assume minimum-sized packets but, in principle, IDLE packets can be MTU-sized with the only effect being a proportionally increased worst-case packet delay. Of course, there are fundamentally a limited number of bytes in each IDLE packet; OrbWeaver leaves the decision on how to allocate these bytes to network architects and operators.

Application Class	**System**	**Weaved Inference?**	**IDLE Messaging?**	**Description**
Traffic Engineering	Flowlet load balancing [105, 36]	✓	✓	Section 2.4.3.
	Performance-aware routing [89]		✓	Propagate route updates in customizable distance-vector routing algorithms using IDLE packets.
	Micro-burst detection [204]	✓	✓	Detect micro-bursts from weaved stream, provide feedback to upstream switches with IDLE packets.
Fault Tolerance	Fast failure recovery [207]	✓	✓	Detect failures (Section 2.4.1), alert upstream switches with IDLE packets for fast data-plane mitigation [54].
	Consistent replicas [110, 199]		✓	Synchronize eventually-consistent distributed state, e.g., for distributed firewalls, with IDLE packets.
Monitoring	Packet forensics [81]		✓	Transfer packet postcards in IDLE packets to reduce overhead of packet history tracking.
	Network queries [143, 77]	✓	✓	Support queries over both flow and weaved stream statistics, export query results in IDLE packets.
	Latency localization [76]	✓	✓	Measure latency in network core using weaved stream, disseminate measurements with IDLE packets.
Network Services	Clock synchronization [103]	✓	✓	Section 2.4.2.
	Header compression [174, 92]		✓	Synchronize state of point-to-point packet header compressors with IDLE packets.
	Event-based network control [172]		✓	Carry network control events in IDLE packets.

Table A.1: OrbWeaver use cases. A diverse range of data-plane applications can use OrbWeaver's weaved stream to learn about conditions in the network and/or communicate via IDLE packets that consume no data-packet bandwidth.

A.1.2. Preventing Starvation

The primary goal of the paper is to explore the opportunistic use of IDLE cycles for in-network coordination. Because of our opportunistic approach, there may be cases where IDLE packets get starved by user packets; however, as previously noted, two factors mitigate the issue:

- The lack of IDLE packets itself reveals concrete information of the network condition (per R1 guarantee of the weaved stream predictability).

- Prior works observed that persistent user traffic is rare, instead, IDLE cycles (every 10s or 100s of µs) are ubiquitous.

A wide range of applications can be implemented with only opportunistic communication. Of course, some applications may need additional guarantees, e.g., applications requiring a strict, real-time guarantee w.r.t. minimum rate (i.e., maximum inter-IDLE-packet gap); or applications that need more aggregate bandwidth than the weaved stream can guarantee in a timely fashion.

In these cases, networks can apply a priority escalation mechanism by adding a single register of N (number of ports) slots and check the elapsed time since last seen IDLE packet. Applications can seamlessly escalate the priority of IDLE packets when too much time passes (per the applications' guaranteed rate SLO). In these situations, OrbWeaver still eliminates nearly all overhead in the presence of (micro)bursts, but may impose a fixed overhead during extended periods of congestion.

A.2. Generalization to Other Platforms

Our focus in this paper was on the Tofino family of programmable switches. While a detailed taxonomy and analysis of every programmable platform is out of the scope of this paper, there is reason to believe that other programmable platforms have similar features or can emulate the features needed to implement OrbWeaver.

In particular, OrbWeaver leverages three hardware features of Tofino switches: (1) packet generation, (2) multicast, and (3) packet prioritization. Among these, support for the latter two can be found in almost every modern forwarding device that is designed to handle the Ethernet protocol.

Support for onboard packet generation is not as universal; however, one potential solution is to connect a port on each switch to a simple device/CPU responsible for generating regular, periodic packets. Of course, a CPU, even with real-time scheduling optimizations, may not be as dependable as the Tofino packet generator. This may necessitate additional tolerances.

Finally, our conversations with switch vendors indicate that OrbWeaver's mechanisms will scale to future switches with both increased bandwidth and port counts. Part of this is due to the fact that most of OrbWeaver's components scale with the clock rate of the switch and/or are independent to each pipeline. The notable exception is packet generation; however, we note that OrbWeaver currently has more than an order of magnitude of headroom (Section 2.3.2.1). If MTU transmission time does eventually outpace packet generation latency, OrbWeaver's properties will degrade gracefully.

A.3. Energy-Efficient Ethernet (EEE)

The Ethernet standard contains an optional EEE mechanism [173], which allows switches to transition links into a Low-Power Idle (LPI) mode when there is no data to send.

OrbWeaver may be able provide compatibility by turning off the IDLE stream on a per-port, per-direction basis if there is no user traffic during the past S seconds. Each packet flowing between two OrbWeaver switches would then need a single bit reserved as an 'LPI' indicator. Upon receiving an IDLE packet with the 'LPI' indicator set, a receiver will change its expectation from requiring a packet every τ_i seconds to requiring one every τ_i' seconds ($\tau_i' \gg \tau_i$). The very first user packet after the low-power idle mode will be sent with the 'LPI' indicator unset. Loss can be addressed by again emulating EEE and sending several indicator packets in a row.

Enabling this feature may impact the responsiveness of OrbWeaver applications, but we note that all of the use cases studied can make do with less frequent but still regular coordination. OrbWeaver may be able to synchronize these low-power updates with existing synchronization-maintenance events in the PHY.

A.4. Proof of Priority-effect on User Traffic

Theorem. For an arbitrary user packet size distribution and arrival process, with strict priority scheduling and a measurement time window $T \gg \Delta t$ (Δt denotes transmission time of a single IDLE packet), the throughput of the user traffic is unaffected by the IDLE stream.

Proof. Consider a packet sequence $p_1, \ldots, p_n$ with size $\Delta t_1, \ldots, \Delta t_n$ and original schedule $t_1, \ldots, t_n$, denote the new schedule upon the coexistence of IDLE stream as $t'_1, \ldots, t'_n$.

We first prove $\forall i \in [1, n-1]$, $t'_i \leq (t_i + \Delta t) \rightarrow t'_{i+1} \leq (t_{i+1} + \Delta t)$. The case for preemptive scheduling is trivially true. We focus on the case of non-preemptive scheduling.

Base case with p_1: the worst case delay of the transmission is when right at t_1, an IDLE packet is scheduled to transmit and with strict priority p_1 is scheduled right next to it. Hence $t'_1 \leq (t_1 + \Delta t)$.

For the inductive step, given the new schedule of p_i satisfying $t'_i \leq (t_i + \Delta t)$, we need to show that $t'_{i+1} \leq (t_{i+1} + \Delta t)$. There are three cases for the next packet p_{i+1}:

- $t_{i+1} > (t_i + \Delta t_i + \Delta t)$: at t_{i+1}, the previous packet has finished transmission in the new schedule since $t_{i+1} > (t_i + \Delta t_i + \Delta t) \geq t'_i + \Delta t_i$. The worst case delay is when IDLE packet is scheduled right at t_{i+1} and the transmission is delayed by Δt, i.e., $t'_{i+1} \leq (t_{i+1} + \Delta t)$ holds.

- $t'_i + \Delta t_i \leq t_{i+1} \leq (t_i + \Delta t_i + \Delta t)$: at t_{i+1}, p_i finishes transmitting in the new schedule, similar to the previous case, the worst case is Δt when right at t_{i+1}, IDLE packet gets scheduled, hence $t'_{i+1} \leq (t_{i+1} + \Delta t)$ holds.

- $t_i + \Delta t_i \leq t_{i+1} < t'_i + \Delta t_i$: p_{i+1} has been queued since p_i is still transmitting until $t'_i + \Delta t_i$ in the new schedule. With strict priority, p_{i+1} will start transmission right at $t'_i + \Delta t_i$ ignoring the IDLE packet. Hence, $t'_{i+1} = t'_i + \Delta t_i \leq t_i + \Delta t + \Delta t_i \leq (t_{i+1} + \Delta t)$.

By induction, we have $t'_n \leq (t_n + \Delta t)$, that is, the latency impact is tightly bounded by Δt for an arbitrary user packet and won't accumulate across packets. Given such fixed workload, consider the impact of the IDLE stream over the original transmission time $T = t_n + \Delta t_n - t_1$. For the new

Configuration	SRAM	TCAM	Metadata	Tbls	Regs
16×100 Gbps	80 KB	1.28 KB	85 b	3	1
32×25 Gbps	80 KB	1.28 KB	53 b	3	1

Table A.2: Additional data plane resources for OrbWeaver's weaved stream generation over an L2 forwarding switch. Ports are binned into groups of 2 and 4, and only 256 multicast groups reserved.

transmission time window $[t'_1, t'_n + \Delta t_n]$, the duration $T' = t'_n + \Delta t_n - t'_1 \leq \max(t'_n) + \Delta t_n - \min(t'_1) \leq t_n + \Delta t + \Delta t_n - t_1$. Hence, $T' - T \leq \Delta t$. Since $T \gg \Delta t$, the throughput of the high priority user packet stream is not impacted. $\square$

A.5. Probability of Notification in Use Case #1

We can formally express the probability that a notification is sent before the flow is evicted. Consider the case where there is a drop in flow f and user packets are all MTU-sized, i.e., there is one packet per period, τ. Assume that the flow cache holds N records and 3 can be packed in each IDLE.

$$\begin{aligned} P(\text{notified}) &= \frac{P(\text{IDLE contains } f)}{P(\text{IDLE contains } f) + P(\text{new } f' \text{ replaces } f)} \\ &= \frac{\frac{3}{N} P(\text{IDLE})}{\frac{3}{N} P(\text{IDLE}) + \frac{1}{N}(1 - P(\text{IDLE}))P(\text{new flow})} \\ &= \frac{P(\text{IDLE})}{P(\text{IDLE}) + (1 - P(\text{IDLE}))P(\text{new flow})/3} \end{aligned}$$

where $P(\text{IDLE})$ is the probability that an IDLE packet was sent during a given period τ, and $P(\text{new flow})$ is the probability that a user packet's flow cannot be found in the cache. Smaller packets multiply the second term in the denominator; a larger N decreases it by improving cache hit rates. The probability that a flow record is evicted *before* it is sent (i.e., that we miss the loss) is 1 less the above value.

A.6. OrbWeaver Data Plane Resource Overhead

§2.3 details the overhead of OrbWeaver's weaved stream generation on user traffic and energy usage. We note that OrbWeaver also uses data plane resources for IDLE seed packet filtering and replication, as shown in Table A.2. For each category, OrbWeaver only occupies a small fraction of the total switch resources (for instance $< 1\%$ of both SRAM and TCAM).

BIBLIOGRAPHY

[1] Google data center pue performance. URL https://www.google.com/about/datacenters/efficiency/.

[2] Iqrb-1 high-performance rubidium oscillators. URL https://www.digikey.com/en/product-highlight/i/iqd-frequency-products/iqrb-1-high-performance-rubidium-oscillators.

[3] Ieee standard 1588-2008, 2008. URL https://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4579757.

[4] Online algorithm for mean absolute deviation and large data set, 2010. URL https://stats.stackexchange.com/q/3378.

[5] Network Time Protocol Version 4: Protocol and Algorithms Specification. RFC 5905, 2010. URL https://rfc-editor.org/rfc/rfc5905.txt.

[6] IEEE 802.1Qbb: Priority-based flow control, 2011. URL https://1.ieee802.org/dcb/802-1qbb/.

[7] Data plane development kit (DPDK), 2015. URL http://www.dpdk.org.

[8] Ontario man 'heartbroken' after ticketmaster cancels 'double-sold' seat, 2018. URL https://www.ticketnews.com/2018/03/paul-simon-fan-scored-floor-seats-had-them-revoked-by-ticketmaster-after-seat-was/.

[9] The p4 language specification, 2018. URL https://p4.org/p4-spec/p4-14/v1.0.5/tex/p4.pdf.

[10] The caida ucsd statistical information for the caida anonymized internet traces, 2019. URL https://www.caida.org/data/passive/passive_trace_statistics.xml.

[11] Project natick: Microsoft's underwater datacenter, 2020. URL https://news.microsoft.com/source/features/sustainability/project-natick-underwater-datacenter/.

[12] Latency numbers every programmer should know, 2020. URL https://colin-scott.github.io/personal_website/research/interactive_latency.html.

[13] Apache cassandra, 2021. URL https://cassandra.apache.org/.

[14] Yahoo! cloud serving benchmark (ycsb), 2021. URL https://github.com/brianfrankcooper/YCSB.

[15] Precision time protocol at meta, 2022. URL https://engineering.fb.com/2022/11/21/production-engineering/precision-time-protocol-at-meta/.

[16] Connectx-6 dx ethernet adapter cards datasheet, 2023. URL https://www.nvidia.com/content/dam/en-zz/Solutions/networking/ethernet-adapters/connectX-6-dx-datasheet.pdf.

[17] Dell emc powerswitch s4048-on support page, 2023. URL https://www.dell.com/support/home/en-us/product-support/product/force10-s4048-on/docs.

[18] Katran: A high performance layer 4 load balancer, 2023. URL https://github.com/facebookincubator/katran.

[19] Building Software-Defined, High-Performance, and Efficient vRAN Requires Programmable Inline Acceleration, 2023. URL https://developer.nvidia.com/blog/building-software-defined-high-performance-and-efficient-vran-requires-programmable-inline-acceleration/.

[20] Time stamping - nvidia networking docs, 2023. URL https://docs.nvidia.com/networking/display/ofedv502180/time-stamping.

[21] Time appliance project, 2023. URL https://opencomputeproject.github.io/Time-Appliance-Project/.

[22] Apache flink: Stateful computations over data streams, 2024. URL https://flink.apache.org/.

[23] Apache kafka, 2024. URL https://kafka.apache.org/.

[24] Azure network virtual appliance, 2024. URL https://aviatrix.com/learn-center/cloud-security/azure-network-virtual-appliance/.

[25] 800G/1.6T Ethernet: Innovations and Challenges, 2024. URL https://community.fs.com/article/800g-16t-ethernet-innovations-and-challenges.html.

[26] Hugging face models, 2024. URL https://huggingface.co/models.

[27] Juniper precision time protocol overview, 2024. URL https://www.juniper.net/documentation/us/en/software/junos/time-mgmt/topics/concept/ptp-overview.html.

[28] Terabit Ethernet, 2024. URL https://en.wikipedia.org/wiki/Terabit_Ethernet.

[29] Zero waste, 2024. URL https://en.wikipedia.org/wiki/Zero_waste.

[30] Saksham Agarwal, Arvind Krishnamurthy, and Rachit Agarwal. Host congestion control. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 275–287, 2023.

[31] Remzi Can Aksoy and Manos Kapritsos. Aegean: replication beyond the client-server model. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 385–398, 2019.

[32] Richard Alimi, Ye Wang, and Y. Richard Yang. Shadow configuration as a network management primitive. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, page 111–122, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605581750.

[33] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). *SIGCOMM Comput. Commun. Rev.*, 40(4):63–74, August 2010. ISSN 0146-4833.

[34] 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.

[35] 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, pages 503–514, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2836-4. doi: 10.1145/2619239.2626316.

[36] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, et al. Conga: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 503–514, 2014.

[37] Pradeep Ambati, Inigo Goiri, Felipe Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, Marcus Fontoura, and Ricardo Bianchini. Providing slos for resource-harvesting vms in cloud platforms. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 735–751. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/ambati.

[38] Amin Vahdat. Reinventing Computing: How necessity will transform next-generation infrastructure, 2023. URL https://www.youtube.com/watch?v=9lBbqH_1KS4&t=1175s.

[39] Amin Vahdat. Coming of age in the fifth epoch of distributed computing, accelerated by machine learning, 2024. URL https://cloud.google.com/blog/topics/systems/the-fifth-epoch-of-distributed-computing.

[40] Serhat Arslan, Yuliang Li, Gautam Kumar, and Nandita Dukkipati. Bolt:{Sub-RTT} congestion control for {Ultra-Low} latency. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 219–236, 2023.

[41] Serhat Arslan, Sundararajan Renganathan, and Bruce Spang. Green with envy: Unfair congestion control algorithms can be more energy efficient. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, pages 220–228, 2023.

[42] 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. ISBN 978-1-939133-01-4. URL https://www.usenix.org/conference/nsdi18/presentation/arzani.

[43] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. The potential dangers of causal consistency and an explicit solution. In *Proceedings of the Third ACM Symposium on Cloud Computing*, pages 1–7, 2012.

[44] Deepak Bansal, Gerald DeGrace, Rishabh Tewari, Michal Zygmunt, James Grantham, Silvano Gai, Mario Baldi, Krishna Doddapaneni, Arun Selvarajan, Arunkumar Arumugam, et al. Disaggregating stateful network functions. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1469–1487, 2023.

[45] Tom Barbette, Chen Tang, Haoran Yao, Dejan Kostić, Gerald Q Maguire Jr, Panagiotis Papadimitratos, and Marco Chiesa. A high-speed load-balancer design with guaranteed per-connection-consistency. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 667–683, 2020.

[46] R. Ben-Basat, X. Chen, G. Einziger, and O. Rottenstreich. Efficient measurement on programmable switches using probabilistic recirculation. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, pages 313–323, 2018.

[47] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. Pint: Probabilistic in-band network telemetry. SIGCOMM '20, page 662–680, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450379557.

[48] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280, 2010.

[49] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and et al. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014. ISSN 0146-4833.

[50] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM international symposium on microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[51] K Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985.

[52] Huan Chen and Theophilus Benson. Hermes: Providing tight control over high-performance sdn switches. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '17, page 283–295, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450354226.

[53] Xinyi Chen, Liangcheng Yu, Vincent Liu, and Qizhen Zhang. Cowbird: Freeing cpus to compute by offloading the disaggregation of memory. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 1060–1073, 2023.

[54] M. Chiesa, R. Sedar, G. Antichi, M. Borokhovich, A. Kamisiński, G. Nikolaidis, and S. Schmid. Fast reroute on programmable switches. *IEEE/ACM Transactions on Networking*, pages 1–14, 2021. doi: 10.1109/TNET.2020.3045293.

[55] Marco Chiesa, Roshan Sedar, Gianni Antichi, Michael Borokhovich, Andrzej Kamisiński, Georgios Nikolaidis, and Stefan Schmid. Purr: A primitive for reconfigurable fast reroute: Hope for the best and program for the worst. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, pages 1–14, 2019.

[56] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22, 2013.

[57] Intel Corporation. P4-16 intel tofino native architecture – public version. Application Note 631348-0001, Intel Corporation, March 2021.

[58] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, Jesse Alpert, Jing Ai, Jon Olson, Kevin DeCabooter, Marc de Kruijf, Nan Hua, Nathan Lewis, Nikhil Kasinadhuni, Riccardo Crepaldi, Srinivas Krishnan, Subbaiah Venkata, Yossi Richter, Uday Naik, and Amin Vahdat. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 373–387, Renton, WA, April 2018. USENIX Association. ISBN 978-1-939133-01-4. URL https://www.usenix.org/conference/nsdi18/presentation/dalton.

[59] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019. URL https://www.flux.utah.edu/paper/duplyakin-atc19.

[60] Thomas G. Edwards and Warren Belkin. Using sdn to facilitate precisely timed actions on real-time data streams. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, page 55–60, New York, NY, USA, 2014. Association for Computing

Machinery. ISBN 9781450329897.

[61] Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 523–535, 2016.

[62] Marco Faltelli, Giacomo Belocchi, Francesco Quaglia, Salvatore Pontarelli, and Giuseppe Bianchi. Metronome: adaptive and precise intermittent packet retrieval in dpdk. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, pages 406–420, 2020.

[63] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. Power provisioning for a warehouse-sized computer. *ACM SIGARCH computer architecture news*, 35(2):13–23, 2007.

[64] Maryam Feily, Alireza Shahrestani, and Sureswaran Ramadass. A survey of botnet and botnet detection. In *2009 Third International Conference on Emerging Security Information, Systems and Technologies*, pages 268–273. IEEE, 2009.

[65] João Ferreira Loff, Daniel Porto, João Garcia, Jonathan Mace, and Rodrigo Rodrigues. Antipode: Enforcing cross-service causal consistency in distributed applications. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 298–313, 2023.

[66] Daniel Firestone, Andrew Putnam, Hari Angepat, Derek Chiou, Adrian Caulfield, Eric Chung, Matt Humphrey, Kalin Ovtcharov, Jitu Padhye, Doug Burger, Dave Maltz, Albert Greenberg, Sambhrama Mundkur, Alireza Dabagh, Mike Andrewartha, Vivek Bhanu, Harish Kumar Chandrappa, Somesh Chaturmohta, Jack Lavier, Norman Lam, Fengfen Liu, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Kushagra Vaid, and David A. Maltz. Azure accelerated networking: Smartnics in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, April 2018.

[67] K. Foerster, S. Schmid, and S. Vissicchio. Survey of consistent software-defined network updates. *IEEE Communications Surveys Tutorials*, 21(2):1435–1461, Secondquarter 2019. ISSN 2373-745X. doi: 10.1109/COMST.2018.2876749.

[68] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[69] Rohan Gandhi, Hongqiang Harry Liu, Y Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. Duet: Cloud scale load balancing with hardware and software. *ACM SIGCOMM Computer Communication Review*, 44(4):27–38, 2014.

[70] Rohan Gandhi, Y. Charlie Hu, Cheng kok Koh, Hongqiang (Harry) Liu, and Ming Zhang. Rubik: Unlocking the power of locality and end-point flexibility in cloud scale load balancing. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 473–485, Santa Clara, CA, July 2015. USENIX Association. ISBN 978-1-931971-225. URL https://www.usenix.org/conference/atc15/technical-sessions/presentation/gandhi.

[71] 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 *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 81–94, 2018.

[72] Soudeh Ghorbani, Zibin Yang, P. Brighten Godfrey, Yashar Ganjali, and Amin Firoozshahian. Drill: Micro load balancing for low-latency data center networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 225–238, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450346535.

[73] 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.

[74] 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 ACM SIGCOMM 2009 conference on Data communication*, pages 51–62, 2009.

[75] Boris Grubic, Yang Wang, Tyler Petrochko, Ran Yaniv, Brad Jones, David Callies, Matt Clarke-Lauer, Dan Kelley, Soteris Demetriou, Kenny Yu, et al. Conveyor: One-tool-fits-all continuous software deployment at meta. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation*, 2023.

[76] 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 *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 139–152, 2015.

[77] 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*, SIGCOMM '18, page 357–371, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355674.

[78] Eashan Gupta, Prateesh Goyal, Ilias Marinos, Chenxingyu Zhao, Radhika Mittal, and Ranveer Chandra. Dbo: Fairness for cloud-hosted financial exchanges. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 550–563, 2023.

[79] Udit Gupta, Young Geun Kim, Sylvia Lee, Jordan Tse, Hsien-Hsin S Lee, Gu-Yeon Wei, David

Brooks, and Carole-Jean Wu. Chasing carbon: The elusive environmental footprint of computing. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 854–867. IEEE, 2021.

[80] David Hancock and Jacobus van der Merwe. HyPer4: Using p4 to virtualize the programmable data plane. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, page 35–49, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342926. doi: 10.1145/2999572.2999607. URL https://doi.org/10.1145/2999572.2999607.

[81] 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.

[82] 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. Association for Computing Machinery. ISBN 9781450346535.

[83] Jean-Michel Helary. Observing global states of asynchronous distributed applications. In *Distributed Algorithms: 3rd International Workshop Nice, France, September 26–28, 1989 Proceedings 3*, pages 124–135. Springer, 1989.

[84] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David 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*, pages 54–66, 2018.

[85] Thomas Holterbach, Edgar Costa Molero, Maria Apostolaki, Alberto Dainotti, Stefano Vissicchio, and Laurent 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, 2019.

[86] 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. ISBN 9781450320566.

[87] 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.

[88] Kuo-Feng Hsu, Ryan Beckett, Ang Chen, Jennifer Rexford, and David Walker. Contra: A pro-

grammable system for performance-aware routing. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 701–721, Santa Clara, CA, February 2020. USENIX Association. ISBN 978-1-939133-13-7. URL https://www.usenix.org/conference/nsdi20/presentation/hsu.

[89] Kuo-Feng Hsu, Ryan Beckett, Ang Chen, Jennifer Rexford, and David Walker. Contra: A programmable system for performance-aware routing. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 701–721, 2020.

[90] Stephen Ibanez, Gianni Antichi, Gordon Brebner, and Nick McKeown. Event-driven packet processing. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, HotNets '19, page 133–140, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450370202.

[91] Romain Jacob, Jackie Lim, and Laurent Vanbever. Does rate adaptation at daily timescales make sense? In *Proceedings of the 2nd Workshop on Sustainable Computer Systems*, pages 1–7, 2023.

[92] Van Jacobson. Compressing tcp/ip headers for low-speed serial links. Technical report, RFC 1144, February, 1990.

[93] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, and et al. B4: Experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 3–14, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450320566.

[94] 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. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 3–14, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450320566.

[95] 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 *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 947–960, Renton, WA, July 2019. USENIX Association. ISBN 978-1-939133-03-8. URL https://www.usenix.org/conference/atc19/presentation/jeon.

[96] Vimalkumar Jeyakumar, Mohammad Alizadeh, Yilong Geng, Changhoon Kim, and David Mazières. Millions of little minions: Using packets for low latency network programming and visibility. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 3–14, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328364.

[97] Zhipeng Jia and Emmett Witchel. Nightcore: Efficient and scalable serverless computing

for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 152–166, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383172. doi: 10.1145/3445814.3446701. URL https://doi.org/10.1145/3445814.3446701.

[98] John P. John, Ethan Katz-Bassett, Arvind Krishnamurthy, Thomas Anderson, and Arun Venkataramani. Consensus routing: The internet as a distributed system. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, page 351–364, USA, 2008. USENIX Association. ISBN 1119995555221.

[99] Bea Johnson. *Zero Waste Home: The ultimate guide to simplifying your life*. Penguin UK, 2013.

[100] Srikanth Kandula, Dina Katabi, Matthias Jacob, and Arthur Berger. Botz-4-sale: Surviving organized ddos attacks that mimic flash crowds. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 287–300, 2005.

[101] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Proceedings of the 42nd annual international symposium on computer architecture*, pages 158–169, 2015.

[102] Qiao Kang, Lei Xue, Adam Morrison, Yuxin Tang, Ang Chen, and Xiapu Luo. Programmable in-network security for context-aware BYOD policies. In *29th USENIX Security Symposium (USENIX Security 20)*, Boston, MA, August 2020. USENIX Association.

[103] 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*, pages 8–20, 2019.

[104] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design guidelines for domain specific languages. *CoRR*, abs/1409.2378, 2014.

[105] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. Hula: Scalable load balancing using programmable data planes. In *Proceedings of the Symposium on SDN Research*, pages 1–12, 2016.

[106] Naga Praveen Katta, Jennifer Rexford, and David Walker. Incremental consistent updates. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, page 49–54, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450321785.

[107] Junaid Khalid and Aditya Akella. Correctness and performance for stateful chained network functions. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 501–516, Boston, MA, February 2019. USENIX Association. ISBN 978-1-931971-49-2. URL https://www.usenix.org/conference/nsdi19/presentation/khalid.

[108] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. In-band network telemetry via programmable dataplanes. In *Demo paper at SIGCOMM '15*, 2015.

[109] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. Tea: Enabling state-intensive network functions on programmable switches. 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 90–106, 2020.

[110] Daehyeok Kim, Jacob Nelson, Dan R. K. Ports, Vyas Sekar, and Srinivasan Seshan. Redplane: Enabling fault-tolerant stateful in-switch applications. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 223–244, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383837.

[111] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, 2018.

[112] Ajay D Kshemkalyani, Michel Raynal, and Mukesh Singhal. An introduction to snapshot algorithms in distributed computing. *Distributed systems engineering*, 2(4):224, 1995.

[113] 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. Association for Computing Machinery. ISBN 9781450379557. doi: 10.1145/3387514.3406591. URL https://doi.org/10.1145/3387514.3406591.

[114] Ten H Lai and Tao H Yang. On distributed snapshots. *Information Processing Letters*, 25(3): 153–158, 1987.

[115] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*, pages 179–196. 2019.

[116] Benjamin C Lee, David Brooks, Arthur van Benthem, Udit Gupta, Gage Hills, Vincent Liu, Benjamin Pierce, Christopher Stewart, Emma Strubell, Gu-Yeon Wei, et al. Carbon connect: An ecosystem for sustainable computing. *arXiv preprint arXiv:2405.13858*, 2024.

[117] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. Globally synchronized time via datacenter networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 454–467, 2016.

[118] Yiran Lei, Liangcheng Yu, Vincent Liu, and Mingwei Xu. Printqueue: performance diagnosis via queue measurement in the data plane. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 516–529, 2022.

[119] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 1–14, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341936.

[120] Chunjie Li, Tjeerd-Jan Stomph, David Makowski, Haigang Li, Chaochun Zhang, Fusuo Zhang, and Wopke van der Werf. The productive performance of intercropping. *Proceedings of the National Academy of Sciences*, 120(2):e2201886120, 2023.

[121] Xuesong Li, Wenxue Cheng, Tong Zhang, Jing Xie, Fengyuan Ren, and Bailong Yang. Power efficient high performance packet i/o. In *Proceedings of the 47th International Conference on Parallel Processing*, pages 1–10, 2018.

[122] Youjie Li, Iou-Jen Liu, Yifan Yuan, Deming Chen, Alexander Schwing, and Jian Huang. Accelerating distributed reinforcement learning with in-switch computing. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 279–291, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450366694.

[123] 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, 2019.

[124] Yuliang Li, Gautam Kumar, Hema Hariharan, Hassan Wassel, Peter H Hochschild, Dave Platt, Simon Sabato, Minlan Yu, Nandita Dukkipati, Prashant Chandra, et al. Sundial: Fault-tolerant clock synchronization for datacenters. 2020.

[125] Jiaxin Lin, Tao Ji, Xiangpeng Hao, Hokeun Cha, Yanfang Le, Xiangyao Yu, and Aditya Akella. Towards accelerating data intensive application's shuffle process using smartnics. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 7(2):1–23, 2023.

[126] Junda Liu, Aurojit Panda, Ankit Singla, Brighten Godfrey, Michael Schapira, and Scott Shenker. Ensuring connectivity via data plane mechanisms. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'13, pages 113–126, Lombard, IL, 2013. USENIX Association. ISBN 978-1-931971-00-3.

[127] Junda Liu, Aurojit Panda, Ankit Singla, Brighten Godfrey, Michael Schapira, and Scott Shenker. Ensuring connectivity via data plane mechanisms. In *10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pages 113–126, 2013.

[128] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. F10: A fault-tolerant engineered network. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI'13, page 399–412, USA, 2013. USENIX Association.

[129] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. F10: A fault-tolerant engineered network. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 399–412, 2013.

[130] Yunhe Liu, Nate Foster, and Fred B Schneider. Causal network telemetry. In *Proceedings of the 5th International Workshop on P4 in Europe*, pages 46–52, 2022.

[131] 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 ACM SIGCOMM Conference*, SIGCOMM '16, pages 101–114, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4193-6. doi: 10.1145/2934872.2934906.

[132] John W. Lockwood, Nick McKeown, Greg Watson, Glen Gibb, Paul Hartke, Jad Naous, Ramanan Raghuraman, and Jianying Luo. Netfpga–an open platform for gigabit-rate network switching and routing. In *Proceedings of the 2007 IEEE International Conference on Microelectronic Systems Education*, MSE '07, page 160–161, USA, 2007. IEEE Computer Society. ISBN 076952849X.

[133] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. Existential consistency: measuring and understanding consistency at facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 295–310, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450338349. doi: 10.1145/2815400.2815426. URL https://doi.org/10.1145/2815400.2815426.

[134] Jennifer Lundelius and Nancy Lynch. An upper and lower bound for clock synchronization. *Information and control*, 62(2-3):190–204, 1984.

[135] Krishna T Malladi, Benjamin C Lee, Frank A Nothaft, Christos Kozyrakis, Karthika Periyathambi, and Mark Horowitz. Towards energy-proportional datacenter memory with mobile dram. *ACM SIGARCH Computer Architecture News*, 40(3):37–48, 2012.

[136] Friedemann Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of parallel and distributed computing*, 18(4):423–434, 1993.

[137] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[138] 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, pages 15–28, 2017.

[139] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[140] 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. Association for Computing Machinery. ISBN 9781450355674.

[141] Ali Najafi and Michael Wei. Graham: Synchronizing clocks by leveraging local clock properties. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 453–466, 2022.

[142] 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*, SIGCOMM '17, pages 85–98, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4653-5.

[143] 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.

[144] Sergiu Nedevschi, Lucian Popa, Gianluca Iannaccone, Sylvia Ratnasamy, and David Wetherall. Reducing network energy consumption via sleeping and rate-adaptation. In *NsDI*, volume 8, pages 323–336, 2008.

[145] Travis L Nicholson, SL Campbell, RB Hutson, G Edward Marti, BJ Bloom, Rees L McNally, Wei Zhang, MD Barrett, Marianna S Safronova, GF Strouse, et al. Systematic evaluation of an atomic clock at $2\times$ 10- 18 total uncertainty. *Nature communications*, 6(1):6896, 2015.

[146] Tian Pan, Nianbing Yu, Chenhao Jia, Jianwen Pi, Liang Xu, Yisong Qiao, Zhiguo Li, Kun Liu, Jie Lu, Jianyuan Lu, Enge Song, Jiao Zhang, Tao Huang, and Shunmin Zhu. Sailfish: Accelerating cloud-scale multi-tenant multi-service gateways with programmable switches. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 194–206, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383837.

[147] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, et al. Ananta: Cloud scale load balancing. *ACM SIGCOMM Computer Communication Review*, 43(4):207–218, 2013.

[148] David Patterson. A decade of machine learning accelerators: Lessons learned and carbon

footprint, 2022. URL https://eecs.berkeley.edu/research/colloquium/220907-2/.

[149] David Patterson, Joseph Gonzalez, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeff Dean. Carbon emissions and large neural network training. *arXiv preprint arXiv:2104.10350*, 2021.

[150] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A centralized "zero-queue" datacenter network. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 307–318, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328364.

[151] P. Phaal, S. Panchen, and N. McKee. InMon corporation's sFlow: A method for monitoring traffic in switched and routed networks. RFC 3176 (Informational), 2001.

[152] Abhiram Ravi, Nandita Dukkipati, Naoshad Mehta, and Jai Kumar. Congestion Signaling (CSIG). Technical report, 2024. URL https://datatracker.ietf.org/doc/draft-ravi-ippm-csig/01/.

[153] Mark Reitblatt, Nate Foster, Jennifer Rexford, and David Walker. Consistent updates for software-defined networks: Change you can believe in! In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, pages 1–6, 2011.

[154] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, page 323–334, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450314190.

[155] 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, page 123–137, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450335423.

[156] 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.

[157] Harshit Saokar, Soteris Demetriou, Nick Magerko, Max Kontorovich, Josh Kirstein, Margot Leibold, Dimitrios Skarlatos, Hitesh Khandelwal, and Chunqiang Tang. {ServiceRouter}: Hyperscale and minimal cost service mesh at meta. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 969–985, 2023.

[158] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling distributed machine learning with In-Network aggregation. In *18th USENIX Symposium on Networked*

Systems Design and Implementation (NSDI 21), pages 785–808. USENIX Association, April 2021. ISBN 978-1-939133-21-2. URL https://www.usenix.org/conference/nsdi21/presentation/sapio.

[159] Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed computing*, 7(3):149–174, 1994.

[160] Korakit Seemakhupt, Brent E Stephens, Samira Khan, Sihang Liu, Hassan Wassel, Soheil Hassas Yeganeh, Alex C Snoeren, Arvind Krishnamurthy, David E Culler, and Henry M Levy. A cloud-scale characterization of remote procedure calls. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 498–514, 2023.

[161] Naveen Kr. Sharma, Antoine Kaufmann, Thomas Anderson, Arvind Krishnamurthy, Jacob Nelson, and Simon Peter. Evaluating the power of flexible packet processing for network resource allocation. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 67–82, Boston, MA, March 2017. USENIX Association. ISBN 978-1-931971-37-9. URL https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/sharma.

[162] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. Approximating fair queueing on reconfigurable switches. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 1–16, Renton, WA, April 2018. USENIX Association. ISBN 978-1-939133-01-4.

[163] Naveen Kr. Sharma, Chenxingyu Zhao, Ming Liu, Pravein G Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. Programmable calendar queues for high-speed packet scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 685–699, Santa Clara, CA, February 2020. USENIX Association. ISBN 978-1-939133-13-7.

[164] Justine Sherry. The i/o driven server: From smartnics to data movement controllers. Technical Report 3, New York, NY, USA, October 2023.

[165] Xiao Shi, Scott Pruett, Kevin Doherty, Jinyu Han, Dmitri Petrov, Jim Carrig, John Hugg, and Nathan Bronson. Flighttracker: Consistency across read-optimized online stores at facebook. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 407–423, 2020.

[166] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, 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. *SIGCOMM Comput. Commun. Rev.*, 45(4):183–197, August 2015. ISSN 0146-4833.

[167] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh,

Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 15–28, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4193-6.

[168] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 15–28, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341936.

[169] 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*, SOSR '17, page 164–176, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349475.

[170] John Sonchack. *Balancing Performance and Flexibility in Hybrid Network Telemetry Systems*. PhD thesis, University of Pennsylvania, 2020.

[171] John Sonchack, Jonathan M. Smith, Adam J. Aviv, and Eric Keller. Enabling practical software-defined networking security applications with OFX. In *23rd Annual Network and Distributed System Security Symposium*, NDSS '16. Internet Society, 2016.

[172] John Sonchack, Devon Loehr, Jennifer Rexford, and David Walker. Lucid: A language for control in the data plane. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 731–747, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383837.

[173] Charles E. Spurgeon. *Ethernet: The Definitive Guide*. O'Reilly & Associates, Inc., USA, 2000. ISBN 1565926609.

[174] Nik Sultana, John Sonchack, Hans Giesen, Isaac Pedisich, Zhaoyang Han, Nishanth Shyamkumar, Shivani Burad, André DeHon, and Boon Thau Loo. Flightplan: Dataplane disaggregation and placement for p4 programs. In *18th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 21)*, pages 571–592, 2021.

[175] Richard S Sutton et al. *Introduction to reinforcement learning*, volume 135.

[176] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, Jan 1997. ISSN 1558-1896. doi: 10.1109/35.568214.

[177] David Tilman. Benefits of intensive agricultural intercropping. *Nature Plants*, 6(6):604–605, 2020.

[178] Muhammad Tirmazi, Adam Barker, Nan Deng, Md Ehtesam Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: the next generation. In *EuroSys'20*, Heraklion, Crete, 2020.

[179] Peter Van Roy and Angel Bravo Gestoso. Saturn: A distributed metadata service for causal consistency. In *EuroSys 2017 Conference*, 2017.

[180] Maarten Van Steen and Andrew S Tanenbaum. *Distributed systems*. Maarten van Steen Leiden, The Netherlands, 2017.

[181] Maarten van Steen and Andrew S. Tanenbaum. *Distributed Systems*. distributed-systems.net, 3 edition, 2017.

[182] Erico Vanini, Rong Pan, Mohammad Alizadeh, Parvin Taheri, and Tom Edsall. Let it flow: Resilient asymmetric load balancing with flowlet switching. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, page 407–420, USA, 2017. USENIX Association. ISBN 9781931971379.

[183] S Venkatesan. Message-optimal incremental snapshots. In *Proceedings. The 9th International Conference on Distributed Computing Systems*, pages 53–54. IEEE Computer Society, 1989.

[184] Dingming Wu, Ang Chen, T. S. Eugene Ng, Guohui Wang, and Haiyong Wang. Accelerated service chaining on a single switch asic. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, HotNets '19, page 141–149, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450370202.

[185] 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.

[186] Jiarong Xing, Wenqing Wu, and Ang Chen. Architecting programmable data plane defenses into the network with fastflex. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, HotNets '19, page 161–169, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450370202.

[187] Nofel Yaseen, John Sonchack, and Vincent Liu. Synchronized network snapshots. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 402–416, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355674.

[188] Nofel Yaseen, John Sonchack, and Vincent Liu. Synchronized network snapshots. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 402–416, 2018.

[189] Nofel Yaseen, Behnaz Arzani, Ryan Beckett, Selim Ciraci, and Vincent Liu. Aragog: Scalable runtime verification of shardable networked systems. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 701–718. USENIX Association, November 2020.

[190] Nofel Yaseen, John Sonchack, and Vincent Liu. tpprof: A network traffic pattern profiler. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 1015–1030, Santa Clara, CA, February 2020. USENIX Association. ISBN 978-1-939133-13-7. URL https://www.usenix.org/conference/nsdi20/presentation/yaseen.

[191] Nofel Yaseen, Behnaz Arzani, Krishna Chintalapudi, Vaishnavi Ranganathan, Felipe Frujeri, Kevin Hsieh, Daniel S Berger, Vincent Liu, and Srikanth Kandula. Towards a cost vs. quality sweet spot for monitoring networks. In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*, pages 38–44, 2021.

[192] Liangcheng Yu, Abhiram Ravi, Weida Huang, Tyler Griggs, KK Yap, Milad Hashemi, Vincent Liu, Neal Cardwell, Arvind Krishnamurthy, and Nandita Dukkipati. Civet: A computational approach to efficient, high-resolution switch telemetry. *In Submission*.

[193] Liangcheng Yu, John Sonchack, and Vincent Liu. Mantis: Reactive programmable switches. In *Proceedings of the ACM SIGCOMM 2020 Conference*, pages 296–309, 2020.

[194] Liangcheng Yu, John Sonchack, and Vincent Liu. Cebinae: scalable in-network fairness augmentation. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 219–232, 2022.

[195] Liangcheng Yu, John Sonchack, and Vincent Liu. Orbweaver: Using idle cycles in programmable networks for opportunistic coordination. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1195–1212, 2022.

[196] Liangcheng Yu, Xiao Zhang, Haoran Zhang, John Sonchack, Dan Ports, and Vincent Liu. Beaver: Practical partial snapshots for distributed cloud services. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 233–249, Santa Clara, CA, July 2024. USENIX Association. ISBN 978-1-939133-40-3. URL https://www.usenix.org/conference/osdi24/presentation/yu.

[197] Yifan Yuan, Omar Alama, Jiawei Fei, Jacob Nelson, Dan RK Ports, Amedeo Sapio, Marco Canini, and Nam Sung Kim. Unlocking the power of inline floating-point operations on programmable switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 683–700, 2022.

[198] Chaoliang Zeng, Layong Luo, Teng Zhang, Zilong Wang, Luyang Li, Wenchen Han, Nan Chen, Lebing Wan, Lichao Liu, Zhipeng Ding, et al. Tiara: A scalable and efficient hardware acceleration architecture for stateful layer-4 load balancing. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1345–1358. USENIX Association, 2022.

[199] Lior Zeno, Dan RK Ports, Jacob Nelson, and Mark Silberstein. Swishmem: Distributed shared state abstractions for programmable switches. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, pages 160–167, 2020.

[200] Cheng Zhang, Jun Bi, Yu Zhou, Abdul Basit Dogar, and Jianping Wu. Hyperv: A high performance hypervisor for virtualization of the programmable data plane. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–9. IEEE, 2017.

[201] Irene Y. Zhang. Operation ordering in systems, 2017. URL https://irenezhang.net/research/consistency.html.

[202] Menghao Zhang, Guan-Yu Li, Shicheng Wang, Chang Liu, Ang Chen, Hongxin Hu, Guofei Gu, Qianqian Li, Mingwei Xu, and Jianping Wu. Poseidon: Mitigating volumetric ddos attacks with programmable switches. 2020.

[203] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. High-resolution measurement of data center microbursts. In *Proceedings of the 2017 Internet Measurement Conference*, IMC '17, pages 78–85, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5118-8.

[204] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. High-resolution measurement of data center microbursts. In *Proceedings of the 2017 Internet Measurement Conference*, pages 78–85, 2017.

[205] Yinda Zhang, Liangcheng Yu, Gianni Antichi, Ran Ben Basat, and Vincent Liu. Enabling silent telemetry data transmission with invisiflow. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, 2025.

[206] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. Crisp: Critical path analysis of large-scale microservice architectures. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 655–672, 2022.

[207] 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.

[208] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y. Zhao, and Haitao Zheng. Packet-level telemetry in large datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 479–491, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3542-3. doi: 10.1145/2785956.2787483.

[209] 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.

[210] 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.