EMPOWERING CLOUD DATA CENTERS WITH NETWORK PROGRAMMABILITY

by
Zhuolong Yu

A dissertation submitted to The Johns Hopkins University in conformity with the requirements for the degree of Doctor of Philosophy

Baltimore, Maryland
January, 2023

Abstract

Cloud data centers are a critical infrastructure for modern Internet services such as web search, social networking and e-commerce. However, the gradual slow-down of Moore's law has put a burden on the growth of data centers' performance and energy efficiency. In addition, the increasing of millisecond-scale and microsecond-scale tasks also bring higher requirements to the throughput and latency for the cloud applications. Today's server-based solutions are hard to meet the performance requirements in many scenarios like resource management, scheduling, high-speed traffic monitoring and testing.

In this dissertation, we study these problems from a network perspective. We investigate a new architecture that leverages the programmability of new-generation network switches to improve the performance and reliability of clouds. As programmable switches only provide very limited memory and functionalities, we exploit compact data structures and deeply co-design software and hardware to best utilize the resource. More specifically, this dissertation presents four systems:

(i) NetLock: A new centralized lock management architecture that co-designs programmable switches and servers to simultaneously achieve high performance and rich policy support. It provides orders-of-magnitude higher throughput than existing systems with microsecond-level latency, and supports many commonly-used policies such as performance isolation.

(ii) HCSFQ: A scalable and practical solution to implement hierarchical fair queueing

on commodity hardware at line rate. Instead of relying on a hierarchy of queues with complex queue management, HCSFQ does not keep per-flow states and uses only one queue to achieve hierarchical fair queueing.

(iii) AIFO: A new approach for programmable packet scheduling that only uses a single FIFO queue. AIFO utilizes an admission control mechanism to approximate PIFO which is theoretically ideal but hard to implement with commodity devices.

(iv) Lumina: A tool that enables fine-grained analysis of hardware network stack. By exploiting network programmability to emulate various network scenarios, Lumina is able to help users understand the micro-behaviors of hardware network stacks.

Thesis Readers

Dr. Xin Jin (Advisor)
Assistant Research Professor
Department of Computer Science
Johns Hopkins University

Dr. Vladimir Braverman (Advisor)
Associate Professor
Department of Computer Science
Johns Hopkins University

Dr. Mosharaf Chowdhury
Associate Professor
Department of Electrical Engineering & Computer Science
University of Michigan

Acknowledgements

Foremost, I would like to express my gratitude to my advisors, Xin Jin and Vladimir (Vova) Braverman, for their guidance, patience and support during my PhD journey. Xin focuses on system research and Vova is more on the theory side, while both of them share the same passion for cross-discipline research with solid systems and efficient algorithms. During these years, Xin and Vova supported me in every way they could. Their guidance on how to do research has greatly benefited me in thinking and tackling problems. Besides the guidance on research, Xin and Vova also connected me with many collaborations and internships. Having Xin and Vova as my advisor has been one of the luckiest things I have had in my life.

I would like to thank Mosharaf Chowdhury who is on my dissertation committee. I started the collaboration with Mosharaf in my first year and finished two projects with him: NetLock and AIFO. I really appreaciate the chance to work with him. Mosharaf has always been energetic and enthusiastic throughout the projects. He and his group are the best collaborators I can imagine. Their valuable and timely input has made our collaboration enjoyable. I also want to thank Ryan Huang and Tamás Budavári who are on my GBO committee together with Xin, Vova and Mosharaf. The suggestions they give me during my GBO examination are very helpful.

I have had the pleasure of working with Wei Bai during my internship at Microsoft Research. He has taught me a lot about how to do research from his industrial perspective. He also gave me lots of career advice and provided great help with my job search. I would like to thank Jean Tourrilhes and Puneet Sharma for hosting my

internship at HP Labs. Jean and Puneet are great researcher and very kind person. I really enjoyed working with them and their team.

I have been fortunate to work with many other great collaborators—Xiaoqi Chen, Chuheng Hu, Nikita Ivkin, Zhenhua Liu, Chunming Qiao, Shachar Raindel, Jennifer Rexford, Ion Stoica, Xiao Sun, Liguang Xie, Ying Xiong, Hongli Xu, Jingfeng Wu, Qianyu Zhang, Yiwen Zhang, Gongming Zhao, Yangming Zhao, Samson Zhou.

I also want to thank our staff members at Hopkins—Zack Burwell, Kim Franklin, and others. Their efforts have made everything so smooth. I feel very lucky to have my group mates—Hang Zhu, Zhihao Bai, Zhen Zhang, Jingfeng Wu, Haoran Li, Xuan Wu, Aditya Krishnan, Nikita Ivkin and Zaoxing Liu. Discussing research and sharing fun stuff at Malone 235 are my precious memories.

Finally, I would like to express my greatest gratitude to my family for their continuous and unconditional love, understanding and support.

To my family.

Contents

List of Figures

Chapter 1

Introduction

Cloud data centers are a critical infrastructure for modern society as they power large portions of Internet services like search, e-commerce and social networking. The performance and energy efficiency are important for large data centers. While in recent years, the slow-down of Moore's law brings challenges: the processors' frequency and power efficiency are not increasing as fast as before. It puts a burden on the growth of data centers' performance and energy effciency. Besides, the rise of millisecond-scale and even microsecond-scale tasks bring unprecedented requirements on the throughput and latency for the cloud applications. Today's server-based solutions cannot meet the performance requirements in many scenarios like resource management, scheduling, high-speed traffic monitoring and testing.

To tackle the challenges, various of domain-specific accelerators (DSAs) such as GPU, TPU, smartNIC, programmable switches etc., are widely adopted by data centers. Domain-specific accelerators have shown significant improvement over general-purpose processors in terms of performance and energy efficiency. However, the performance improvement comes with a price: domain-specific accelerators normally focus on specific functionalities and lack generality compared with general-purpose processors. To fully utilize the performance gain from DSAs, the co-design between general CPUs and DSAs, software and hardware is essential for data centers.

In this dissertation, we investigate the problem from a network perspective. Specifically, we focus on DSAs for networking—programmable networking ASICs, and study how can programmable networking empower data centers.

1.1 Domain-Specific Accelerator for Networking

Domain-specific accelerators enable developers to program domain-specific tasks flexibly and efficiently. They are suitable for performing a narrow range of intensive computation tasks, such as GPUs for graphics processing tasks and TPUs for machine learning tasks. DSAs have demonstrated the ability to balance flexibility (better than fixed-function ASICs) and performance (better than general-purpose processors) for computing systems. Languages and programming APIs have been designed for popular DSAs as they get widely used in the industry.

In recent years, the network community also proposed domain-specific accelerators for networking. Among them, programmable switches have drawn great attention from both research community and industry. One of the most popular architectures for programmable switches is called PISA (Protocol Independent Switch Architecture). PISA-based programmable switch normally has a control plane CPU and a data plane programmable ASIC connected by PCIe. There are two major components in the switch data plane—programmable parser and programmable match-action pipeline. The parser can parse customized key-value fields in the packet. The match-action pipeline has multiple stages and stateful ALUs that can read and update key-value fields at line rate. PISA-based programmable switches have been available commercially and can be easily programmed with the specialized language named P4 [1]. With the support of PISA and P4, network owners can easily write self-defined network functions based on their own needs. The network functions can be flexibly compiled and deployed to the switches, which greatly save the time cost when we need

to add/change some features on the switch, compared with the traditional fashion [1].

1.2 Opportunities and Challenges

Cloud data center is typically owned by a single administrative entity (cloud provider like Amazon, Microsoft, Google) that controls both software and hardware. It's relatively easy for a cloud provider to adopt new system architectures. For example, follwing the notion of software-defined networking (SDN) [2], Google built their private software defined WAN called B4 ten years ago, which largely improves the efficiency and flexibility of traffic engineering. Similarly, other companies like Microsoft and Meta (Facebook) have also proposed their SDN solutions. While with SDN, network owners can control the network by programming the control plane module, programmable networking takes one step further: network owners can have the control of packet processing in the data plane. Programmable network has the potential to enable new features and provide extraordinary performance for cloud data centers in various of aspects. In this dissertation, we explore new architectures with the power of new-generation programmable switches. We prove that the new architectures can provide advanced features and exceptional performance improvement for cloud data centers, compared with conventional architectures.

While the new architecture directly benefits from switch hardware for high performance, integrating programmable networking devices into the current data center architecture is nontrivial. There are three main challenges:

- **Limited on-chip memory and lack of memory management mechanism, memory access pattern.** In recent years, the computation tasks are becoming more and more memory hungry. Unlike modern commodity servers that can

[1] Traditionally, when the network owners want a new feature, they need to inform the device vendors. The vendors need to let their software team write the software for the new feature. After that, the hardware team will start to design the new ASIC. In order to have a stable hardware with the feature, the whole process normally takes two or three years.

have hundreds of GBs to several TBs of memory, network switches normally only have 10–100MBs of on-chip memory due to the consideration of cost and performance. Besides, switch memory is usually organized in a manner suitable for packet processing: the memory is normally splitted among multiple stages in the processing pipeline as the packets are processed stage by stage. A memory region (register) can only be accessed by its own stage and a register array can only be accessed once when a packet is processed in the pipeline. As a result, network programs should be dedicatedly designed to fully utilize the switch memory to realize the functionalities.

- **Limited functionalities.** Because of strict timing and resource requirements, in a processing pipeline, a network switch is only able to support a small number of operations from a limited operation set. Some operations that may seem straightforward on general CPUs such as multiplication, division, floating-point operations, sorting, etc., are complex for ALUs on switches. Besides, the commodity programmable switches lack the programmability in their traffic manager or MMU (Memory Management Unit)—they are using multiple first-in first-out (FIFO) queues to manage packet scheduling just like what traditional switches do. However, this also gives us the chances to think out of the box: what traffic manager do we need? In fact, we will describe some of our findings in Chapter 3 and Chapter 4.

- **Difficult co-design across devices.** There are two approaches to overcome the challenge of limited memory and limited functionalities. The first approach is to use approximations. For example, instead of storing everything on the switch memory, we can use sketches to largely save memory footprint. Besides, some operations like multiplication can be composed by several simple operations (such as addition, subtraction, bit shifting). However, this approach does not always work as it can hurt the performance and accuracy, and sometimes it is not possible to approximate

all operations well. The second approach is to co-design programmable switch with other devices (other switches and servers). Such co-design requires efficient memory allocation among multiple devices and offloading in-network primitives from general processors to switches. When doing the approximations and/or co-designs, there are several questions we need to consider: How to identify which operation/primitive to approximate or offload? Does the co-design make the whole system brittle, is it error-prone to packet loss and other failures? Do the approximation and co-design work in a larger scale? How to conduct realistic experiments to evaluate the effect of the changes we make?

1.3 Contributions

In this dissertation, we present four novel systems that leverage flexible network programming to realize advanced features and provide exceptional performance improvement.

NetLock: Fast and Centralized Lock Management Using Programmable Switches. Lock managers are widely used by distributed systems. Traditional centralized lock managers can easily support policies between multiple users using global knowledge, but they suffer from low performance. In contrast, emerging decentralized approaches are faster but cannot provide flexible policy support. Furthermore, performance in both cases is limited by the server capability.

We present NetLock in Chapter 2, a new centralized lock manager that co-designs servers and network switches to achieve high performance without sacrificing flexibility in policy support. The key idea of NetLock is to exploit the capability of emerging programmable switches to directly process lock requests in the switch data plane. Due to the limited switch memory, we design a memory management mechanism to seamlessly integrate the switch and server memory. To realize the locking functionality

in the switch, we design a custom data plane module that efficiently pools multiple register arrays together to maximize memory utilization We have implemented a NetLock prototype with an Intel Tofino switch and a cluster of commodity servers. Evaluation results show that NetLock improves the throughput by 14.0–18.4×, and reduces the average and 99% latency by 4.7–20.3× and 10.4–18.7× over DSLR, a state-of-the-art RDMA-based solution, while providing flexible policy support.

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

In Chapter 3, we argue that two emerging trends are making CSFQ relevant again: (*i*) cloud computing which makes it feasible to change the protocol within the same datacenter or across datacenters owned by the same provider, and (*ii*) programmable switches which can implement sophisticated packet processing at line rate. To this end, we present the first realization of CSFQ using programmable switches. In addition, we generalize CSFQ to a multi-level hierarchy, which naturally captures the traffic in today's datacenters, e.g., tenants at the first level and flows of each tenant at the second level of the hierarchy. We call this scheduler Hierarchical Core-Stateless Fair Queueing (HCSFQ), and show that it is able to accurately approximate hierarchical fair queueing. HCSFQ is highly scalable: it uses just a single FIFO queue, does not perform per-packet scheduling, and only needs to maintain state for the interior nodes of the hierarchy. We present analytical results to prove the lower bounds of HCSFQ. Our testbed experiments and large-scale simulations show that CSFQ and HCSFQ can provide fair bandwidth allocation and ensure isolation.

AIFO: Programmable packet scheduling with a single queue. Programmable

packet scheduling enables scheduling algorithms to be programmed into the data plane without changing the hardware. Existing proposals either have no hardware implementations for switch ASICs or require multiple strict-priority queues.

Chapter 4 presents Admission-In First-Out (AIFO) queues, a new solution for programmable packet scheduling that uses only a *single* first-in first-out queue. AIFO is motivated by the confluence of two recent trends: *shallow* buffers in switches and *fast-converging* congestion control in end hosts, that together leads to a simple observation: the decisive factor in a flow's completion time (FCT) in modern datacenter networks is often *which* packets are enqueued or dropped, not the *ordering* they leave the switch. The core idea of AIFO is to maintain a sliding window to track the ranks of recent packets and compute the relative rank of an arriving packet in the window for admission control. Theoretically, we prove that AIFO provides bounded performance to Push-In First-Out (PIFO). Empirically, we fully implement AIFO and evaluate AIFO with a range of real workloads, demonstrating AIFO closely approximates PIFO. Importantly, unlike PIFO, AIFO can run at line rate on existing hardware and use minimal switch resources—as few as a single queue.

Lumina: Fine-grained Analyzation Tool for Hardware Offloaded Network Stacks. Hardware offloaded network stacks are widely adopted in modern datacenters to meet the demand for high throughput, ultra-low latency and low CPU overhead. To best utilize the superb performance, network developers need to have in-depth understandings of their behaviors. Recent years, there have been various of testing tools helping users to test and understand software network stacks. However, hardware network stacks are left behind as its kernel bypass nature and high performance make testing challenging. In Chapter 5, we present Lumina, a tool to test the correctness and performance of hardware network stacks. The key idea of Lumina is exploiting network programmability to emulate various network scenarios and dumping the complete packet trace for offline analysis. Lumina supports injecting deterministic events with

user-friendly interfaces, thus enabling developers to write precise reproducible tests. Due to the limited resource and flexibility of programmable network devices, we mirror and dump traffic to a pool of dedicated servers for offline analysis. We start with RDMA NIC as the testing target and prototype Lumina with our testbed. We evaluate Lumina with microbenchmark experiments and share our findings on three widely used RDMA NICs using Lumina.

Chapter 2

NetLock: Fast, Centralized Lock Management Using Programmable Switches.

2.1 Introduction

As more and more enterprises move their workloads to the cloud, they are increasingly relying on databases provided by public cloud providers, such as Amazon Web Services [3], Microsoft Azure [4], and Google Cloud [5]. *Performance* and *policy support* are two important considerations for cloud databases. Specifically, cloud databases are expected to provide high performance for many tenants and enable rich policy support to accommodate tenant-specific performance and isolation requirements, such as starvation freedom, service differentiation, and performance isolation.

Lock managers are a critical building block of cloud databases. They are used by multiple concurrent transactions to mediate access to shared resources in order to achieve high-level transactional semantics such as serializability. With recent advancements that exploit fast RDMA networks and in-memory databases to significantly improve the performance of distributed transactions [6, 7] (i.e., decrease *think time*), the overhead of acquiring and releasing locks is now a major component in the end-to-end performance of cloud-based enterprise software [8].

Existing lock manager designs (both centralized and decentralized) face a *trade-off* between performance and policy support (Figure 2-1). The traditional centralized approach uses a server as a *central* point to grant locks [9, 10]. With the global view of all lock operations in the server, this approach can easily support various policies, such as starvation freedom and fairness [10–13]. The drawback is that the lock server, especially its CPU, becomes the performance bottleneck as transaction throughput increases.

To mitigate the CPU bottleneck, recent decentralized solutions leverage fast RDMA networks to achieve high throughput and low latency [7, 8, 14, 15]. Clients acquire and release locks by updating the lock information on the lock server through RDMA, without involving the server's CPU. However, since the locking decisions are made by the clients in a decentralized manner, it is hard to support and enforce rich policies [8].

We present NetLock, a new approach to design and build lock managers that sidesteps the trade-off and achieves both high performance and rich policy support. We observe that compared to the actual data stored in a database, the lock information is only a small amount of metadata. Nonetheless, the metadata requires *high-speed*, *concurrent* accesses. Network switches are specifically designed and optimized for high-speed, concurrent data input-output workloads, making them a natural place to accelerate lock operations.

The key idea of NetLock is to leverage this observation and co-design switches and servers to build a fast, centralized lock manager. Switches provide orders-of-magnitude higher throughput and lower latency than servers. By using switches to process lock requests in the switch data plane, NetLock avoids the CPU bottleneck of server-based centralized approaches, and achieves high performance. By using a centralized design, NetLock avoids the drawback of decentralized approaches and can support many essential policies.

Realizing this idea is challenging for at least two reasons. First, switches only

have limited on-chip memory. Although the size of lock information is orders-of-magnitude smaller than that of the actual storage data, it can still exceed the switch memory size for large-scale cloud databases. While previous work [16] has proposed the idea of extending the switch memory with the server memory, it does not consider the characteristics of locking and does not provide a concrete solution for memory management. To address this challenge, we design a mechanism to seamlessly integrate the switch and server memory to store and process lock requests. NetLock only offloads the *popular* locks to the switch and leaves other locks to servers. We formulate the problem as an optimization problem and design an optimal algorithm for memory allocation.

Second, switches only have limited functionalities in the data plane and cannot process lock requests. Prior work [17] has shown how to build a key-value store in switches and solved the fault-tolerance problem, but a key-value store is not a fully functional lock manager that can support different types of locks and support policies. To address this challenge, we leverage the capability of emerging programmable switches to design a data plane module to implement necessary features required by NetLock. To maximize memory utilization and avoid memory fragmentation, we design a shared queue data structure to pool the register arrays in multiple data plane stages together and allocate it to the locks. Each lock owns an adjustable, continuous region in the shared queue to store its requests. We design custom match-action tables in the data plane to support both shared and exclusive locks with common policies.

NetLock is incrementally deployable and compatible with existing datacenter networks. It is well-suited for cloud providers that have dedicated racks for database services. It only needs to augment the Top-of-Rack (ToR) switches of these database racks with a custom data plane module for processing lock requests. Since the custom module is only invoked by lock messages, other packets are processed by switches as before. NetLock does not change other switches in the network, and it is compatible

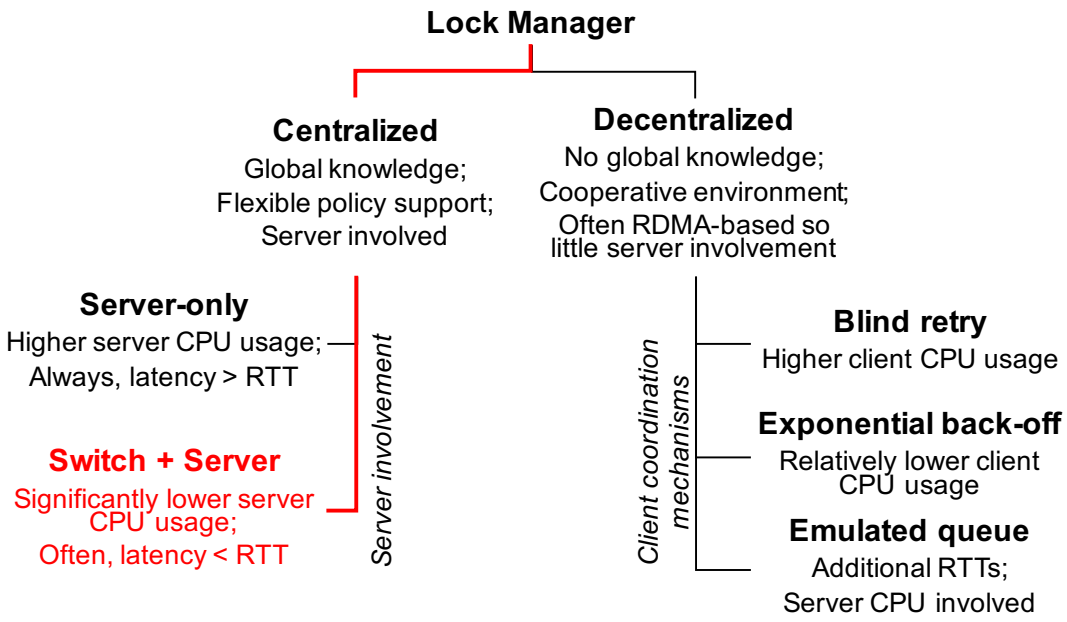


Figure 2-1. Design space for lock management.

with existing routing protocols and network functions.

Recently there is a surge of interest in in-network computing. While it is arguable whether applications should be moved to the network and to what extent, NetLock takes a modest approach to make the network more application-aware. Assisting locking in the network is not a radical deviation from traditional network functionalities. We emphasize that the application (i.e., transaction processing) is still running on servers. NetLock provides locks with switches to resolve contentions and enforce policies for concurrent transactions, which is similar to using switch-based signals like Random Early Detection (RED) and Explicit Congestion Notification (ECN) to resolve congestion and enforce fairness for concurrent flows, but in a more application-aware way for databases. Furthermore, compared to changing all NICs and redesigning applications to leverage RDMA, replacing only the switch and transparently updating the lock manager provides a competitive alternative to high-performance database applications. NetLock can provide better performance and lower the cost by reducing the lock servers.

In summary, we make the following contributions.

- We propose NetLock, a new centralized lock manager architecture that co-designs programmable switches and servers to achieve high performance and flexible policy support.
- We design a memory management mechanism to seamlessly integrate the switch and server memory, and a custom data plane module for switches to store and process lock requests.
- We implement a NetLock prototype on an Intel Tofino switch and commodity servers. Evaluation results show that NetLock improves transaction throughput by 14.0–18.4×, and reduces the average and 99% latency by 4.7–20.3× and 10.4–18.7× over the state-of-the-art DSLR, while providing flexible policy support.

The code of NetLock is open-source and available at https://github.com/netx-repo/NetLock.

2.2 Background and Motivation

In this section, we first provide background on the design of lock managers. Then we motivate the usage of programmable switches to design lock managers, by identifying potential benefits and discussing its feasibility.

2.2.1 Background on Lock Management

Lock managers are used by distributed systems to mediate concurrent access to shared resources over the network, where locks are typically held in servers. There are two main approaches, i.e., centralized and decentralized, as shown in Figure 2-1.

Centralized lock management. A centralized lock manager uses a server as a *central* point to grant locks [9, 10]. Because the server has the global view of all lock

requests and grant decisions, it can easily enforce policies to provide many strong and useful properties, such as starvation-freedom and fairness [10–13].

A centralized lock manager can be *distributed* across multiple servers, by having each server be responsible for a subset of lock objects. There is a distinction between distributed and decentralized. Centralized and decentralized approaches differ in how the decisions to grant locks are made, i.e., whether they are made by the central lock manager or by the clients in a decentralized manner. Both approaches can be made distributed to scale out.

The lock manager can either be co-located with the storage server that actually stores the objects or be in a separate server. In the former case, the lock manager daemon would consume the resources of the storage server, which can be otherwise used to process storage requests such as transactions. In the latter case, lock managers for multiple storage servers can be consolidated to a few dedicated servers.

Decentralized lock management. Centralized lock managers suffer from low performance, as the server CPUs become the bottleneck to handle a large number of lock requests from clients [8]. Decentralized lock managers often leverage fast RDMA networks to address the performance problem [7, 8, 14, 15]. A decentralized lock manager still has a designated server to maintain necessary information for each lock in a lock table, e.g., the current transaction ID that holds the lock and whether the lock is shared or exclusive. Different from centralized ones, a decentralized lock manager relies on clients to make decisions in a distributed manner. The lock table at the lock server is updated by the clients using RDMA verbs, such as SEND, RECV, READ, WRITE, CAS, and FA. This approach reduces CPU utilization at the lock server.

There are a few different strategies for the clients to acquire locks in this approach. The simplest one is *blind fail-and-retry*, where each client tries to acquire

a lock independently, and retries after a timeout if not succeed [7]. This strategy has high client CPU usage, and can cause starvation and hence long tail latencies. *Exponential back-off* can be used to reduce the CPU usage, but it further increases latencies. More advanced ones use *distributed queues* to emulate centralized lock managers [14]. Such strategies, while avoiding starvation, incur extra network round-trips and lose the benefit of high performance. The most recent solution in this category, DSLR [8], adapts Lamport's bakery algorithm [18] to order lock requests and guarantees first-come-first-serve (FCFS) scheduling; this reduces starvation and achieves high throughput.

Decentralized lock managers typically use *advisory locking*, where clients *cooperate* and follow a distributed locking protocol. This is because the clients use RDMA verbs to interact with the lock table in the lock server without involving the server's CPU. It is different from *mandatory locking* used by centralized lock managers that can *enforce* a locking protocol, as the lock manager is solely making locking decisions. Besides the difficulty to enforce a protocol, decentralized lock managers cannot flexibly support various policies such as isolation, without significantly degrading performance using an expensive distributed protocol.

2.2.2 Exploiting Programmable Switches

Providing both high performance and policy support. Traditional server-based approaches make a *trade-off* between performance and policy support. Centralized approaches provide flexible policy support, but have low performance; decentralized approaches achieve the opposite. The goal of this chapter is to design a solution that sidesteps the trade-off and provides both high performance and policy support. Our key idea is to design a centralized solution with fast switches, which can benefit from switches to achieve high performance while still providing flexible policy support as being a centralized approach. Moreover, since switches provide orders-of-magnitude

higher throughput and lower latency than servers, this solution is even faster than decentralized, RDMA-based approaches. This is especially important for emerging fast transaction systems based on RDMA networks and in-memory storage [6, 7]. In these systems, the transactions themselves are executed in memory, and thus the execution cost is comparable to the locking and unlocking cost, meaning that the system needs to spend a considerable amount of server resources for lock managers as for the storage servers themselves. Leveraging switches to build faster lock managers can both improve the transaction performance and reduce the system cost.

Building lock managers with programmable switches. While traditional switches are fixed-function, emerging programmable switches, such as Intel Tofino [19], Broadcom Trident [20] and Cavium XPliant [21], make it feasible to design, build and deploy switch-based lock managers. Leveraging programmable switches provides orders-of-magnitude higher performance than FPGA-based (e.g., SmartNICs) or NPU-based solutions. While this chapter focuses on programmable switches, the mechanisms designed for NetLock can also be applied to programmable NICs.

Programmable switches allow users to develop custom data plane modules, which can parse custom packet headers, perform user-defined actions, and access the switch on-chip memory for stateful operations [1, 22]. With this capability, we can program the switch data plane to parse lock information embedded in a custom header format, to perform lock and unlock actions, and to store the lock table in the switch on-chip memory.

2.3 NetLock Architecture

In this section, we first give the design goals of NetLock, and then provide a system overview of NetLock.

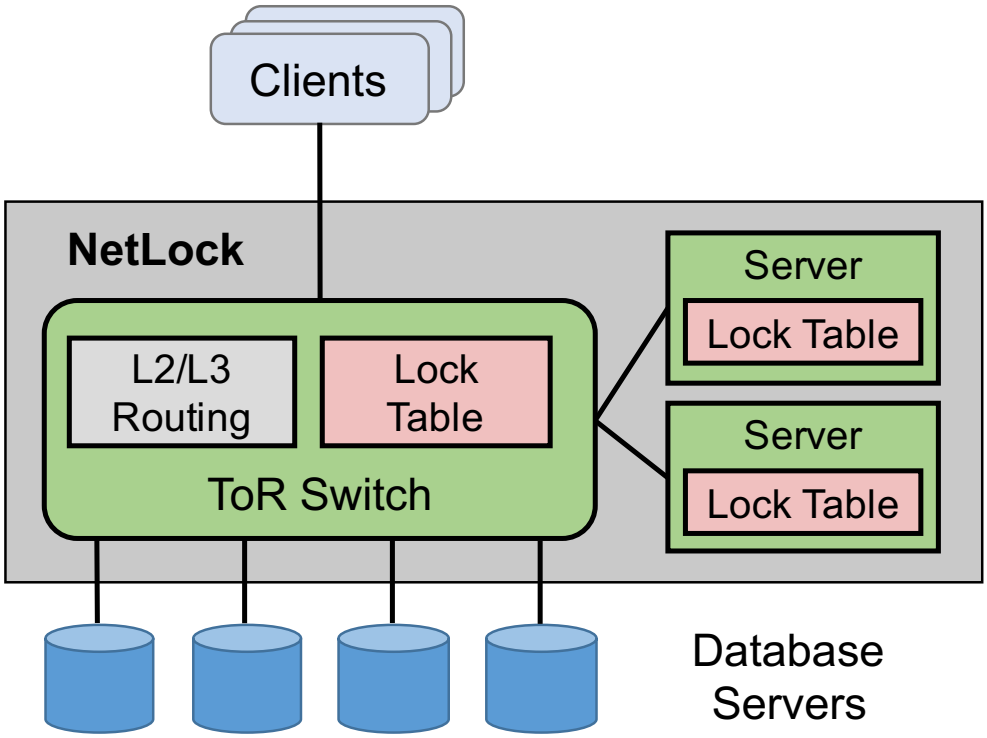


Figure 2-2. NetLock architecture.

2.3.1 Design Goals

NetLock is a fast, centralized lock manager. It is designed to meet the following goals.

- **High throughput.** State-of-the-art distributed transaction systems can process hundreds of millions of transactions per second (TPS) with a single rack [6, 23, 24], and each transaction can involve a few to tens of locks. To avoid being the performance bottleneck of fast distributed transaction systems, the lock manager should be able to process up to a few billion lock requests per second (RPS).

- **Low latency.** Given the tens of microseconds transaction latency enabled by fast networks and in-memory databases [6, 23, 24], the lock manager should provide low latency to process lock requests, in the range of a few to tens of microseconds.

- **Policy support.** For a cloud environment, the lock manager should provide flexible policy support to accommodate tenant-specific requirements. Specifically,

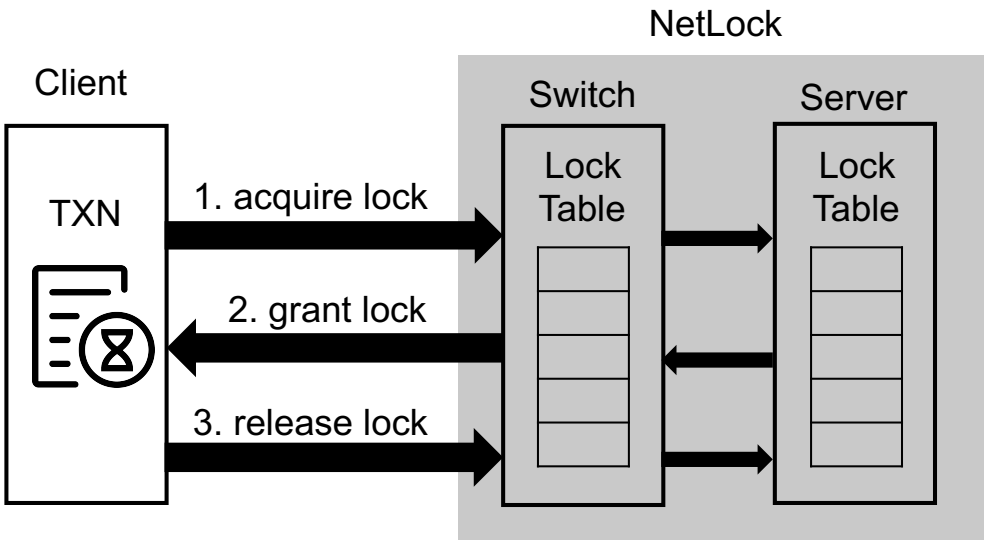


Figure 2-3. Lock request handling in NetLock. The switch directly processes most lock requests.

we consider common policies including starvation freedom, service differentiation, and performance isolation.

2.3.2 System Overview

A NetLock lock manager consists of one switch and multiple servers in the same rack (as shown in Figure 2-2), where the round-trip time (RTT) between machines within the same switch is typically single-digit microsecond. The switch is the ToR switch of a dedicated database rack that is specifically provisioned for database services, which is common in public clouds. Different database racks have their own NetLock instances. Besides adding a new data plane module for NetLock to the ToR switch, no other changes are made to the datacenter network. The ToR switch only invokes the NetLock module to process lock requests, and it processes other packets as usual. NetLock does not affect existing network functionalities.

At a high level, clients send lock requests to NetLock without knowing whether the requests will be processed by a switch or a server. Behind the scene, NetLock processes lock requests with a combination of switch and servers. It integrates the

Algorithm 1 ProcessLockRequest(req)

```
1: if req.lock ∈ switch.locks() then
2:     if req.type == acquire then
3:         if switch.CanGrant(req) then
4:             Grant req.lock to req.client
5:         else if switch.CanQueue(req) then
6:             Queue req at switch
7:         else
8:             Forward req to server
9:     else
10:        Release req.lock, and grant it to pending requests
11: else
12:    Forward req to server
```

switch and server memory to store and process lock requests. When a lock request arrives at the switch, the switch checks whether it is responsible for the lock. If so, it invokes the data plane module to process the lock; otherwise, it forwards the lock requests to the server. The switch only stores and processes the requests on popular locks, while the lock servers are responsible for the requests on unpopular locks. The lock servers also buffer the requests on popular locks when the queues in the switch are overflowed.

2.4 NetLock Design

In this section, we describe the design of NetLock that exploits programmable switches for fast, centralized lock management.

2.4.1 Lock Request Handling

As shown in Figure 2-3, to acquire a lock for a transaction, the client first sends a lock request to NetLock and waits for NetLock to grant the lock. NetLock directly processes most lock requests with the lock switch and only leaves a small portion to the lock servers. After the lock is granted, the client executes its transaction and sends a release notification to NetLock if the lock is no longer needed.

Algorithm 1 shows the pseudocode of the switch. Since the switch is the ToR

switch of the database rack and is on the path for a request to reach the lock servers, the switch can always process the request first. If the switch is responsible for the corresponding lock object (line 1), it checks the lock availability and policy. If the lock can be granted, the switch directly responds to the client (line 3-4). If the lock cannot be granted immediately, the switch queues the request if it has enough memory (line 5-6). If the switch is not responsible for the lock object or does not have sufficient memory, it forwards the request to the lock server based on the destination IP (line 8 and 12). The locks are partitioned between the lock servers. The client obtains the partitioning information from an off-the-shelf directory service in datacenters [25, 26], and sets the destination IP to that of the server responsible for the lock. After the client releases the lock, NetLock can further grant the lock to other requests (line 10). The performance benefit of NetLock comes from that most requests can be directly processed by the switch, without the need to visit a lock server.

One-RTT transactions. In the basic mode, a client gets a grant from NetLock (taking 0.5 RTT by the lock switch or 1 RTT by the lock server) and then issues another request to fetch the data from a database server (taking 1 RTT) to finish the transaction, which takes 1.5–2 RTTs in total. Some recent distributed transaction systems (e.g., DrTM [7], FARM [27] and FaSST [23]) combine lock acquisition and data fetching in a single request to a database server, and thus are able to finish a transaction in 1 RTT. NetLock can apply the same idea to achieve one-RTT transactions. Specifically, after a lock is granted, instead of replying to the client, NetLock forwards the request to the corresponding database server to fetch the item, making lock acquisition and data fetching in one RTT. More importantly, unlike existing solutions (e.g., DrTM, FARM and FaSST) that rely on fail-and-retry which may lead to low throughput and high latency, all requests to the database servers can *successfully* fetch data, because the locks have already been granted by NetLock. This is critical under high-contention scenarios to reduce overhead at both clients

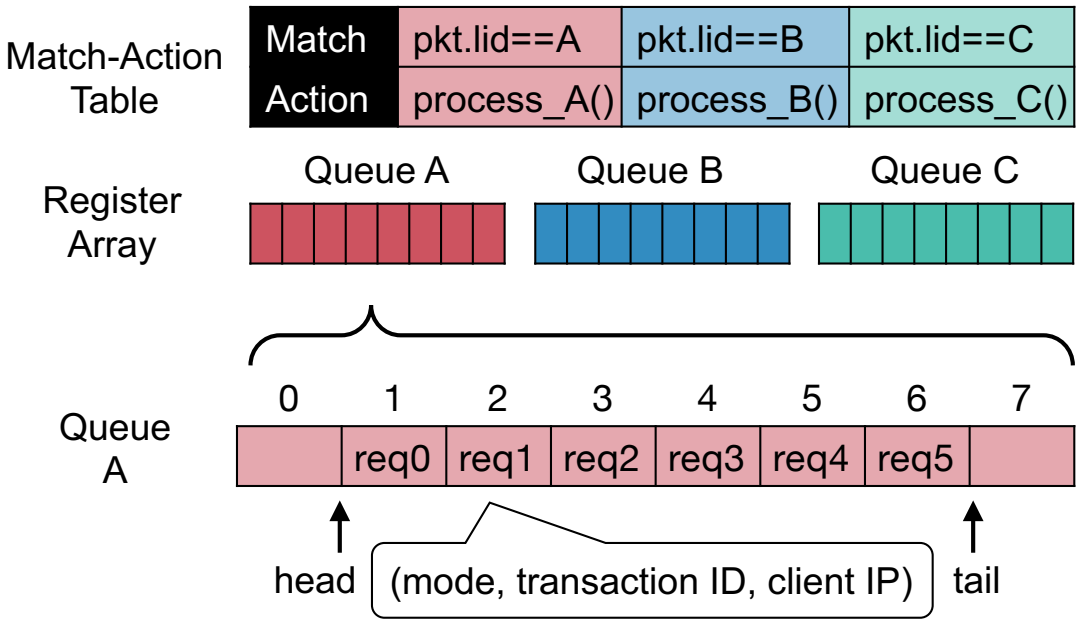


Figure 2-4. Basic data plane design for lock management.

and database servers, and achieve high throughput and low latency. For locks not in the switch, the lock server is combined with the database server as existing solutions to achieve one-RTT transactions. For requests with payloads such as writes, the switch forwards the data if the lock can be granted, and drops the data, otherwise. Some transactions that involve read-modify-write operations cannot fundamentally be done in one RTT because the client has to do some compute and the current design does not push compute to the lock and database servers. In addition to its high performance, NetLock also supports flexible policies that cannot be implemented by existing decentralized solutions.

2.4.2 Switch Data Plane

Programmable switches expose stateful on-chip memory as register arrays to store user-defined data. NetLock leverages register arrays to store and process lock requests in the switch. Figure 2-4 shows a basic data plane design. The design allocates one array for each lock to queue its requests. A special UDP destination port is reserved for NetLock. A lock request contains several fields: action type (acquire/release), lock

ID, lock mode, transaction ID, and client IP. The match-action table maps a lock ID (i.e., *lid*) to its corresponding register array, and the action in the table performs operations on the register array to grant and release locks.

Because register arrays can only be accessed based on a given index, they do not natively support queue operations such as enqueue and dequeue. We implement *circular queues* based on register arrays to support necessary operations for NetLock. Specifically, we allocate extra registers to keep the head and tail pointers. The pointers are looped back to the beginning when they reach the end of the array. For example, queue A in Figure 2-4 has six queued requests, and the head and tail are index 1 and 6, respectively.

Each slot in a queue stores three important pieces of information, i.e., mode, transaction ID, and client IP. Mode indicates whether the request is for a shared or exclusive lock. Transaction ID identifies which transaction the lock is requested for. Client IP stores the IP address from which the lock request is sent. The IP address is used by the switch when it generates a notification to grant the lock to the client. Additional metadata such as timestamp and tenant ID can also be stored together.

Optimize switch memory layout. Because the memory for each register array is pre-allocated and the size is fixed after the data plane program is compiled and loaded into the switch, the basic design cannot flexibly change the queue size at runtime. When the workload changes, the set of locks in the switch and the size of each queue would need to change according to the memory allocation algorithm to maximize the performance. Allocating a large queue to accommodate the maximum possible contentions for each lock is undesirable because it would cause memory fragmentation and result in low memory utilization, especially given that the switch on-chip memory is limited.

To address this problem, we design a *shared queue* to pool multiple register arrays

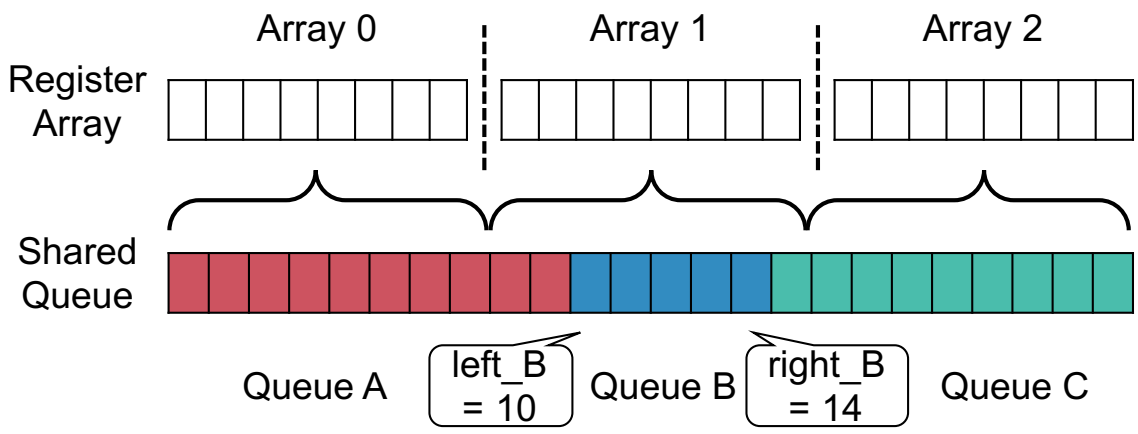


Figure 2-5. Combine multiple register arrays to a shared queue for locks with different queue sizes.

together and enable the queue size to be dynamically adjusted at runtime (Figure 2-5). Instead of statically binding each register array to a lock, we combine these arrays together to build a large queue shared by all the locks. Accessing a slot in the shared queue with an index can be mapped to accessing the register arrays by appropriately setting the index, e.g., accessing slot 10 in the shared queue can be mapped to accessing slot 10-8=2 in array 1. Each lock is allocated with a continuous region in the shared queue to store its requests. We allocate extra registers to store the boundaries of each queue, e.g., 10 and 14 for queue B. Since the boundaries are stored in registers, they can be modified at runtime. Another benefit of this design is that the individual register arrays do not have to be in the same stage, which allows NetLock to pool memory from multiple stages together to build a large queue that exceeds the memory limit of a single stage.

Handle shared and exclusive locks. The shared queue design solves the storage problem of how to store the requests, but it does not solve the computation problem of how to process them. The challenge comes from the limitation that the data plane can only perform one read/write operation to a register array when it processes a packet.

This limitation brings two issues. First, when a lock release notification arrives at the switch, the switch dequeues the corresponding request from the queue, and the lock could be granted to the next request in the queue. This requires two operations: one is to dequeue the head, and the other is to read the new head. Second, when a request to acquire a shared lock is granted, if the following requests in the queue are also for a shared lock, then these requests can also be granted. This requires multiple read operations until an exclusive lock request or the end of the queue. We leverage a feature called *resubmit* available in programmable switches to overcome the limitation. The resubmit feature allows the switch data plane to resubmit the packet to the beginning of the packet processing pipeline, so that the packet can go through and be processed by the pipeline again, obviating the need to send *another* packet to the switch from servers. Note that the use of resubmit here does not cause extra overhead, because the servers in the traditional server-based lock managers also need to send a packet to grant each shared lock to the corresponding client. Figure 2-6 illustrates how to handle the four cases for shared and exclusive locks.

- **Shared → Shared**. When a shared lock is released, the switch dequeues the head, and uses resubmit to check the new head. If the new head is a shared lock request, the processing stops, because the shared lock has already been granted with the old head when it entered the queue.

- **Shared → Exclusive**. This case differs from the first case on that the new head is an exclusive lock request, which has not been granted yet. As such, after the shared lock is released, the lock becomes available, and the switch sends a notification to the client to grant the lock.

- **Exclusive → Shared**. When an exclusive lock is released, the packet is resubmitted to grant the next lock request in the queue. The resubmit action is repeated by multiple times until an exclusive request or the end of the queue.

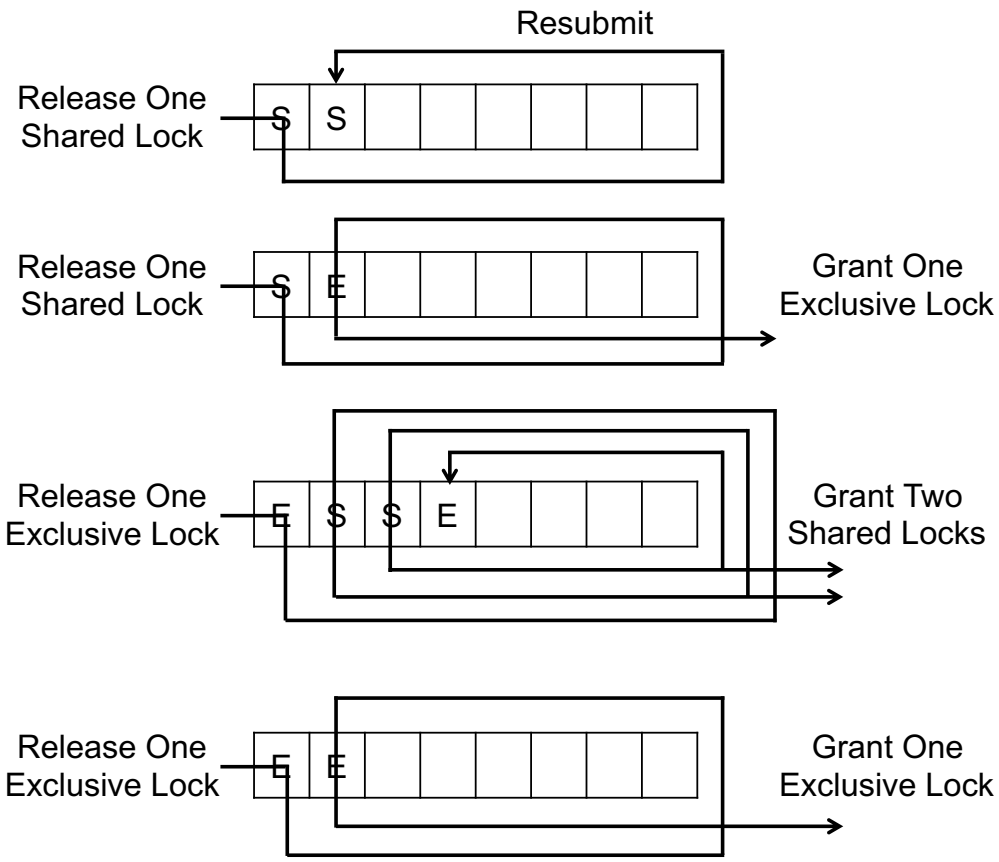


Figure 2-6. Handle shared and exclusive locks.

- **Exclusive → Exclusive**. When an exclusive lock is released and the next request is also exclusive, the next request is granted. Because the lock is exclusive and cannot be shared, the switch does not need to resubmit it again.

Algorithm 2 shows the pseudocode of the switch that covers the above four cases. If the request is to acquire a lock, it is enqueued (line 1-2). The request is directly granted if the queue is empty, or if all requests in the queue are shared and the request is also shared (line 3-5). If the request is to release a lock, the current head in the queue is removed, and the lock is resubmitted to grant the following request (line 7-12). For case "shared → shared", no further processing is needed. For case "shared → exclusive" and "exclusive → exclusive", the new head is granted the lock (line 15-16). For case "exclusive → shared", multiple subsequent shared locks are granted (line 17-27). The nuance in the lock processing is that when there are multiple transactions

Algorithm 2 SwitchDataPlane(pkt)

```
1: if pkt.op == acquire then
2:     queue.enqueue(pkt)
3:     if queue.is_empty() or
4:      (queue.is_shared() and pkt.mode == shared) then
5:         grant_lock(pkt.tid, pkt.cip)
6: else
7:     if meta.flag == 0 then
8:         (mode, tid, cip) ← queue.dequeue()
9:         meta.flag ← 1
10:        meta.mode ← mode
11:        meta.pointer ← queue.head()
12:        resubmit()
13:    else if meta.flag == 1 then
14:        (mode, tid, cip) ← queue[meta.pointer]
15:        if mode == exclusive then
16:            grant_lock(tid, cip)
17:        else if meta.mode == exclusive then
18:            grant_lock(tid, cip)
19:            meta.pointer ← meta.pointer.next()
20:            meta.flag ← 2
21:            resubmit()
22:    else
23:        (mode, tid, cip) ← queue[meta.pointer]
24:        if mode == shared then
25:            grant_lock(tid, cip)
26:            meta.pointer ← meta.pointer.next()
27:            resubmit()
```

holding a shared lock, these transactions may not release their locks in the order that the requests are enqueued. Because the switch can only release locks at the head of the queue, it does not check the transaction ID when releasing locks. This design does not affect the correctness, because only one transaction can hold an exclusive lock, and the operations for releasing shared locks are commutative.

Pipeline layout. A switch may have several pipelines, and the pipelines do not share state. In NetLock, the lock tables and their register arrays are placed in the *egress* pipes that *connect to their corresponding lock servers*. This placement avoids unnecessary recirculation across pipelines. Specifically, when a request arrives, it is sent to the egress pipe that either owns the lock or connects to a lock server that has the lock. If the request is granted, it is mirrored to the upstream port to the client or

the database server to finish the transaction (Section 2.4.1). Otherwise, it is enqueued either at the egress pipe or in a lock server.

2.4.3 Switch-Server Memory Management

Since the switch on-chip memory is limited, NetLock co-designs the switch and servers and stores only the popular locks to the switch memory. The switch control plane is responsible for creating and deleting locks, and assigning memory for locks between the switch and lock servers. The key challenge in memory allocation is that it requires us to consider the contentions from multiple requests to the same lock. When a lock is granted to a client, other requests are queued in the switch and occupy memory space until the lock is released.

Memory allocation mechanism. We first analyze the amount of switch memory required to support a certain throughput. Let the rate of lock requests to object i be r_i. Let the maximum contention for object i be c_i, which means that there are at most c_i concurrent requests for object i. We assume c_i is known based on the knowledge of how many clients may need this lock, and we use a counter to measure r_i. Let the queue size for object i in the switch be s_i. If $s_i \geq c_i$, then the switch can guarantee to process all requests for object i, without queueing requests in the server. The memory allocation is to decide which locks to assign to the switch, and for each assigned lock, how much switch memory to allocate for it. Let the switch memory size be S. We formulate the problem as the following optimization problem.

$$maximize \quad \sum_i r_i s_i / c_i \tag{2.1}$$

$$s.t. \quad \sum_i s_i \leq S \tag{2.2}$$

$$s_i \leq c_i \tag{2.3}$$

The goal is to process as many lock requests in the switch as possible, reducing the

Algorithm 3 MemoryAllocation(locks)

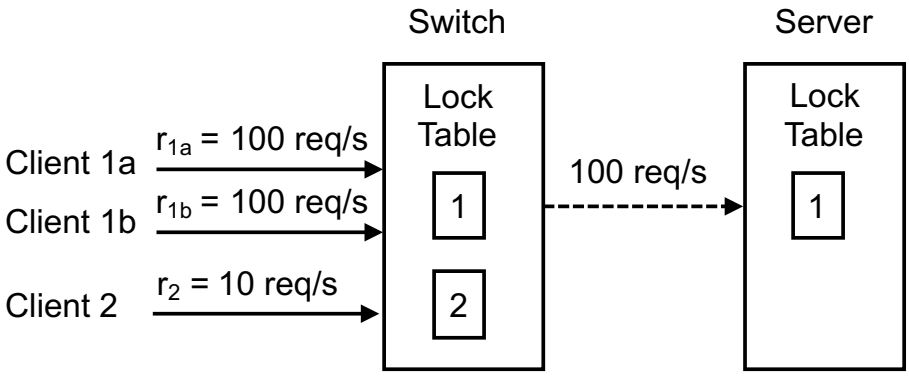
```
1: Sort locks by r_i/c_i in decreasing order
2: for lock i in locks do
3:     s_i ← min(switch.available, c_i)
4:     switch.available ← switch.available − s_i
5:     Allocate s_i for lock i in switch memory
6: Allocate remaining locks to the servers
```

number of servers we need for NetLock. For object i, because in the worse case the lock requests for i always achieve the maximum contention c_i, only a portion (s_i/c_i) of lock requests can be queued at the switch, and the other portion $(1 - s_i/c_i)$ have to be sent to the server. Therefore, the optimization objective, which is the request rate the switch can guarantee to process, is $\sum_i r_i s_i/c_i$. The constraint is that the total memory allocated to the locks cannot exceed the switch memory size S, i.e., $\sum_i s_i \leq S$. The switch does not need to allocate more than c_i memory slots to object i, thus we have $s_i \leq c_i$.

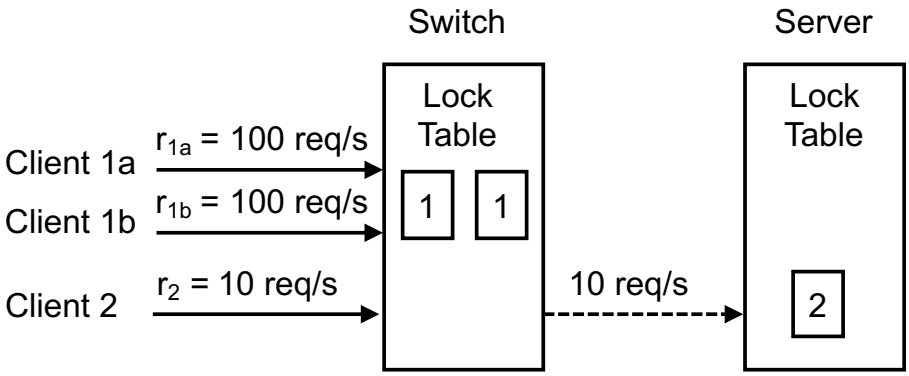
This problem is similar to the fractional knapsack problem, which can be solved with an optimal solution in polynomial time. Algorithm 3 shows the pseudocode. Specifically, the value of allocating one slot to object i in the switch is r_i/c_i. To maximize the objective, the algorithm allocates the switch memory based on the decreasing order of r_i/c_i.

The rate r_i and contention c_i for each lock are obtained by measuring the workload. NetLock maintains two counters to track r_i and c_i for each lock respectively, and updates the memory allocation based on Algorithm 3 when the workload changes. During the update, NetLock first drains the requests of the locks that are to be swapped out from the switch, and then allocates the switch memory to more popular locks. Note that, for inserting a new lock object, the new lock queue is first added to a lock server, and then would be moved to the switch if the lock becomes popular.

Theorem 1. *The memory allocation algorithm (Algorithm 3) is optimal for the optimization problem (1-3).*



(a) Naive memory allocation.



(b) Optimal memory allocation.

Figure 2-7. By allocating two slots in the switch to lock 1, the optimal allocation can process all lock requests to lock 1 in the switch, minimizing the server load.

Proof. We consider the situation where $\sum_i c_i > S$; otherwise, there is enough memory for all the locks. Let there be n locks in total. Without loss of generality, let $\frac{r_1}{c_1} > \frac{r_2}{c_2} > ... > \frac{r_n}{c_n}$. Algorithm 3 allocates as much memory as possible ($min(switch.available, c_i)$) for locks sorted by r_i/c_i. Assume this is not the optimal strategy. Let the optimal strategy be $s_1^*, s_2^*, ..., s_n^*$. Because $\sum_i c_i > S$, there exists at least one lock i such that $s_i^* < c_i$. Let the lock with the smallest ID be j, i.e., for any $i < j, s_i^* = c_i$, and $s_j^* < c_j$. If for any $k > j, s_k^* = 0$, the optimal strategy would be the same as Algorithm 3. Therefore, there exists at least one lock k such that $k > j$ and $s_k^* > 0$. Let $s_j' = s_j^* + 1$ and $s_k' = s_k^* - 1$. Because $\frac{r_j}{c_j} > \frac{r_k}{c_k}$, we have $\sum_i r_i s_i'/c_i > \sum_i r_i s_i^*/c_i$. This contradicts that the allocation $s_1^*, s_2^*, ..., s_n^*$ is optimal. So Algorithm 3 is optimal. □

Example. Figure 2-7 illustrates the key idea of the algorithm. There are two

concurrent clients that acquire exclusive locks for object 1 with a rate of 100 requests per second each. The queue needs two slots to accommodate the contentions from the two clients. There is only one client that acquires exclusive locks for object 2, with a rate of 10 requests per second. The queue only needs one slot for one client. Suppose the switch memory only has two slots. The allocation in Figure 2-7(a) allocates one slot to each lock object. Since the switch cannot queue requests for two clients for object 1, in the worse case where the clients are highly synchronized, half of the requests are sent to the server. On the other hand, the optimal allocation in Figure 2-7(b) allocates two slots to object 1, minimizing the load on the server.

Performance guarantee. Since servers have plenty of memory to queue requests, servers are CPU-bounded, and the bottleneck is on the number of requests that can be processed by a server per second. Let the workload be $W = \{(r_i, c_i)\}$, and the solution to the optimization problem be $S = \{(s_i)\}$. Let r_s and r_e be the request rates that can be supported by a switch and a server, respectively. We assume that the switch is not the bottleneck, i.e., $r_s \geq \sum_i r_i$, so the switch is always able to support the request rate $\sum_i r_i s_i / c_i$. This assumption is reasonable, because if $r_s < \sum_i r_i$, then the ToR switch is congested. In such a case, not all lock requests can even be received by the database rack in the first place, and the workload would not be meaningful. Since the switch can process the request rate $\sum_i r_i s_i / c_i$, it requires $\lceil (\sum_i r_i - \sum_i r_i s_i / c_i) / r_e \rceil$ servers to serve the remaining request rate. In other words, with one switch and $\lceil (\sum_i r_i - \sum_i r_i s_i / c_i) / r_e \rceil$ servers, NetLock guarantees to support the workload $W = \{(r_i, c_i)\}$.

Handling overflowed requests. It is possible that the queues in the switch can be overflowed, because the switch cannot allocate enough memory for the last object it handles or the estimation of maximum contention for an object is inaccurate. For lock i, we denote its switch queue as $q_1[i]$, and its server queue as $q_2[i]$. When $q_1[i]$ is full, the switch forwards the overflowed requests to the server. The overflowed requests are

only buffered in $q_2[i]$ in the server, not processed. Note that this is different from the locks that are not allocated to the switch and only have queues in the servers—the requests of those locks are both buffered and processed by the servers. The switch puts a mark on the packets to distinguish between these two cases.

As both $q_1[i]$ and $q_2[i]$ may contain requests, we need to ensure that the requests are processed as they would in a single queue. To achieve this, the requests are only granted and dequeued by $q_1[i]$ in the switch, and new requests are only enqueued at $q_2[i]$ in the server. When $q_1[i]$ becomes empty, the switch sends a notification to the server, and the server pushes some requests from $q_2[i]$ to $q_1[i]$. The number of requests that can be pushed is no bigger than the number of available slots in $q_1[i]$ to ensure $q_1[i]$ is not overflowed. When $q_2[i]$ becomes empty and $q_1[i]$ is not full, NetLock enters the normal mode, i.e., new requests can directly be enqueued at $q_1[i]$ in the switch. Because $q_2[i]$ is empty, enqueueing at $q_1[i]$ would ensure the same order as a single queue.

Moving locks between the switch and lock servers. When the popularity of a lock changes, the lock will be moved from the switch to a lock server or from its lock server to the switch. When moving a lock, NetLock pauses enqueuing new requests of this lock and waits until the queue is empty to ensure consistency. Memory fragmentation caused by moving locks between the switch and lock servers would reduce the memory that can be actually used to store lock requests. The memory layout on the switch is periodically reorganized to alleviate memory fragmentation.

2.4.4 Policy Support

NetLock is a centralized lock manager that can support and enforce policies. We consider the following three representative policies.

Starvation-freedom. Decentralized lock managers use partial information to grant

locks, which can easily lead to lock starvation. Lock starvation happens when the lock manager allows later lock requests to acquire a lock before earlier lock requests, making some requests wait indefinitely to get the lock. Lock starvation is typically avoided by using a first-come-first-serve (FCFS) policy. The FCFS policy stores lock requests in a first-in-first-out (FIFO) queue, and always grants locks to the head of the queue. This policy is natively supported by the circular queue we design for the switch data plane. With this, NetLock supports request (lock) level starvation-freedom. Note that, there can still be starvation if some transactions do not complete because of deadlock, which is discussed in Section 2.4.5.

Service differentiation with priorities. It is challenging to support priority-based policies in the switch, as a register array can only be accessed once when processing a packet and a priority queue cannot be directly implemented with a register array. We leverage the multi-stage structure of the switch data plane to support priorities. Specifically, we allocate one queue in each stage for one priority. Since the packet is processed stage by stage, the high-priority requests in earlier stages are granted first. The request processing with priorities in the switch data plane follows Algorithm 2 with some tweaks. For a lock request with *i-th* priority, it is directly granted if all queues are empty, or if there is no exclusive lock request holding the lock or queued in the same or higher priority queues and the request itself is also for a shared lock. After the lock is released, NetLock will first grant the lock to the queue with the highest priority. Note that a priority can have a large queue spanning multiple stages to expand its queue size. The limitation of this solution is that the number of priorities is limited to the number of stages, which is usually 10-20 in today's switches. This limitation can be alleviated by approximation, e.g., grouping multiple fine-grained priorities into a single coarse-grained priority. Moreover, only high-priority requests need to be processed in the switch. Low-priority requests do not need fast processing, and can always be offloaded to the lock servers.

Performance isolation with per-tenant quota. Cloud databases often have multiple tenants and need to enforce fairness between them. Without a centralized lock manager, a tenant can generate requests and acquire locks at a faster rate than another tenant, and thus occupies most of the resources. While an FCFS policy can avoid starvation of the slower tenant, it cannot enforce the tenants to stay within their shares. It requires the lock manager to use rate limiters to enforce per-tenant quota. Rate limiters can be implemented in the switch data plane with either meters that can automatically throttle a tenant, or counters that count the tenants' requests and compare with their quotas.

2.4.5 Practical Issues

Switch memory size. We examine whether the switch memory is sufficient for a lock manager from two aspects.

Think time. The think time affects the maximum *turnover* rate of a memory slot. Let T be the duration of a request occupying a slot, which includes the round trip time of sending the grant and release messages and that of executing the transaction (i.e., think time). A slot can be reused by $1/T$ times per second (i.e., the *turnover* rate), providing a throughput of $1/T$ RPS. With S slots, the switch can achieve S/T RPS. Given fast networks and low-latency transactions, T can be a few tens of microseconds. As a switch has tens of MB memory, 100K slots with 20B slot size only consume 2 MB memory, which is a small portion of the total memory. Assuming $T = 20\ \mu s$ and $S = 100K$, the switch can support $S/T = 5$ BRPS, which is sufficient for the database servers the same rack. On the other hand, if $T = 1\ ms$, the switch needs 1M slots to achieve $S/T = 1$ BRPS.

Memory allocation. The memory allocation mechanism affects the utilization of the switch memory. It determines whether the switch can achieve the maximum rate S/T. If the switch memory is allocated to unpopular locks, the switch would only

process a small portion of the total locks. Even when a memory slot is available, it may not be used to process a new request for its lock as there are no pending requests for this unpopular lock. If the memory slots are empty for half of the time, then the switch needs to double its memory slots in order to achieve the maximum rate. NetLock uses an optimal knapsack algorithm to efficiently allocate switch memory to popular locks to maximize the memory utilization. This handles skewed workload distributions. For uniform workload distributions, we combine multiple locks into one *coarse-grained* lock to increase the memory utilization.

In summary, the think time determines the maximum *turnover* rate of a memory slot and thus the maximum throughput the switch can support with a given amount of memory, and the memory allocation mechanism determines whether the system can achieve the maximum *turnover* rate. Experimental results in Section 2.6.4 illustrate the relationship.

Scalability. We focus on rack-scale database systems in this chapter. Based on the above analysis on switch memory size and the experimental results in Section 2.6.4, the memory of one switch is sufficient for most rack-scale workloads, and the ToR switch can be naturally used as the lock switch. In the cases where more memory is needed, additional lock switches can be attached to the rack as specialized accelerators for lock processing. For large-scale database systems that span multiple racks, each rack runs an instance of NetLock to handle the lock requests of its own rack.

Failure handling. We describe how to handle different types of failures in NetLock.

- *Transaction failure.* Transaction failures can be caused by network loss, application crashes, and client failures. When a transaction fails without releasing its acquired locks, other transactions that request for the same locks cannot proceed. NetLock uses a common mechanism, leasing [28], to handle transaction failures. It stores a timestamp together with each lock, and a transaction expires after its lease. The

switch control plane periodically polls the data plane to clear expired transactions.

- *Deadlock.* Deadlocks are caused by multiple transactions waiting for locks held by others, and no transaction can make progress. It is resolved in the same way as for transaction failures. Clients retry when the leases expire until they succeed. In addition, deadlocks can be avoided if priority-based policies are employed.

- *NetLock failure.* When a lock server fails, the locks allocated to this server is assigned to another lock server. Clients resubmit their requests to the new server, and the server waits for the leases to expire before granting the locks. A switch failure is handled in the same way by assigning the locks to a backup switch. After the original switch restarts, the lock requests are queued into the original switch. When releasing a lock, we only grant locks from the backup switch until the queue in the backup switch gets empty. After all the queues in the backup switch get empty, the backup switch is no longer useful. When the switch restarts, it also synchronizes its states with the lock servers and waits for the overflowed requests that are buffered at the lock servers to drain before the switch starts processing new requests on the corresponding locks. The unpopular locks stored in lock servers are not affected by switch failures.

2.5 Implementation

We have implemented a prototype of NetLock, including the lock switch, the lock server, and the client.

The lock switch is implemented with 1704 lines of code in P4, and is compiled to Intel Tofino ASIC [19]. The lock table has a shared queue with a total of 100K slots. With 20B slot size, it only consumes 2MB, which is a small portion of tens of MB on-chip memory. The switch control plane is implemented with 750 lines of code in Python, which allocates the memory in the shared queue to different locks.

The lock server is implemented with 2807 lines of code in C. It handles lock requests that cannot be directly processed by the lock switch. To maximize the efficiency of multi-core processing and improve the performance, it uses Intel DPDK [29], and leverages Receive Side Scaling (RSS) to partition the lock requests between cores and dispatch the lock requests to the appropriate RX queues by the NIC for each core. With these optimizations, a lock server can achieve up to 18 MRPS with a 40G NIC in our testbed.

The client is implemented with 3176 lines of code in C. It is used to generate lock requests to measure the performance in the experiments. It also uses Intel DPDK and RSS to optimize performance, and one client server can generate up to 18 MRPS with a 40G NIC in our testbed.

2.6 Evaluation

2.6.1 Methodology

Testbed. The experiments of NetLock are conducted on our testbed consisting of one 6.5 Tbps Intel Tofino switch and 12 servers. Each server has an 8-core CPU (Intel Xeon E5-2620 @ 2.1GHz) and one 40G NIC (Intel XL710).

Comparison. We compare NetLock with the state-of-the-art lock manager DSLR [8] and DrTM [7]. Since DSLR and DrTM require RDMA, the experiments on DSLR and DrTM are conducted in the Apt cluster of CloudLab [30]. The configuration is comparable to our own testbed. Each server is equipped with an 8-core CPU (Intel Xeon E5-2450 @ 2.1GHz) and a 56G RDMA NIC (Mellanox ConnectX-3). We also compare NetLock with a recently proposed switch-based solution NetChain [17]. NetChain is not a fully functional lock manager, as it only supports exclusive locks. Therefore, requests for shared locks are treated as exclusive locks. NetChain handles concurrent requests with client-side retry. Since NetChain only stores items in the

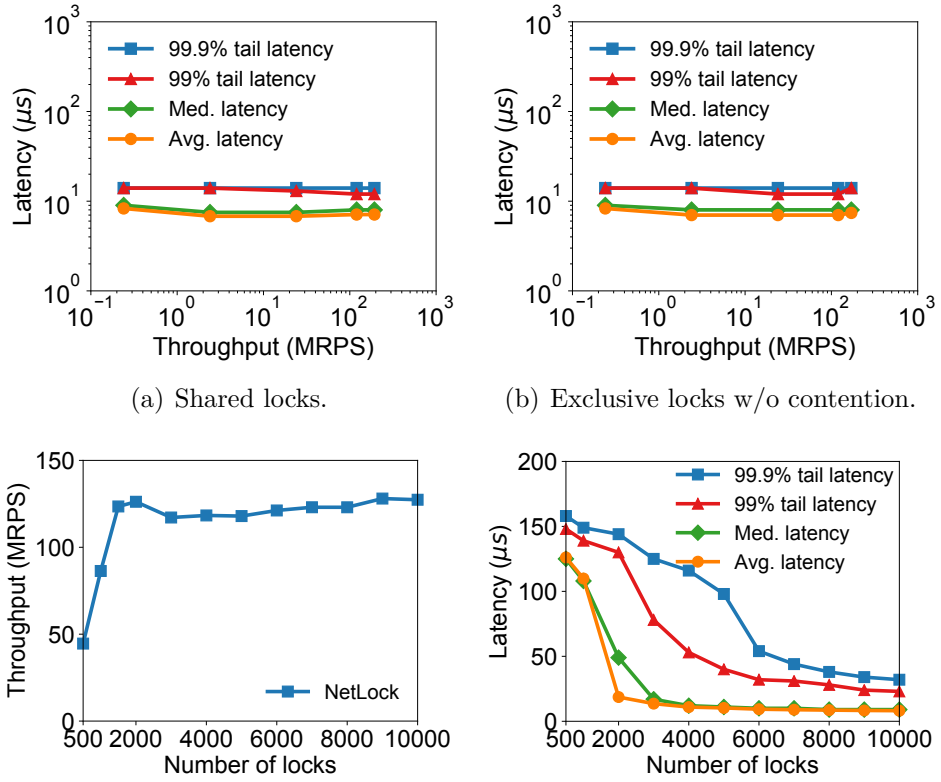
switch, we adapt the lock granularity based on the switch memory size and the number of locks, so that NetChain can handle all the requests in the switch. We emphasize that DSLR, DrTM and NetChain do not support flexible policies.

Workloads. We use two workloads. The first workload is a microbenchmark, which simply generates lock requests to a set of locks. It is useful to measure the basic performance of lock processing. The second workload is TPC-C [31]. It generates transactions based on TPC-C, and each transaction contains a set of lock requests. It is useful to measure the application-level performance. We use two settings for TPC-C, which is the same as DSLR: a low-contention setting with ten warehouses per node, and a high-contention setting with one warehouse per node. We use throughput, in terms of lock requests granted per second (RPS) and transactions per second (TPS), and latency as the evaluation metrics.

2.6.2 Microbenchmark

We use microbenchmark experiments to measure the basic throughput and latency of the lock switch to process lock requests. We cover both shared and exclusive locks.

Shared locks. We first evaluate the performance for shared locks. We use all 12 servers in the testbed to generate requests to the lock switch. Since the requests are for shared locks, there are no contentions and the locks can be directly granted. Figure 2.8(a) shows the relationship between latency and throughput. The median (average) latency is 8 μs (7.1 μs), and the 99% (99.9%) latency is 12 μs (14 μs). We emphasize that the latency is dominated by the processing latency at the client software and NIC; the processing latency at the switch is under 1 μs. The latency is not affected by the throughput, because even we use all 12 servers to generate requests, they can still not saturate the switch. The switch can handle the lock request at line rate, and the Intel Tofino switch used in the experiment is able to process more than 4 billion packets per second.



(a) Shared locks.

(b) Exclusive locks w/o contention.

(c) Exclusive locks w/ contention.

(d) Exclusive locks w/ contention.

Figure 2-8. Microbenchmark results of switch performance on handling lock requests.

Exclusive locks. We then evaluate the performance for exclusive locks. Similar to the previous experiment, we use 12 servers to generate requests for exclusive locks. To measure the baseline performance, the requests are sent to different locks and there are no contentions. Figure 2.8(b) shows the results, which are similar to those for shared locks. This is because in both cases, the requests are directly granted by the switch and processed at line rate.

To show the impact of contention on exclusive locks, we let the servers send lock requests to the same set of locks, and vary the number of locks in the set. The level of contention decreases as the number of locks increases. Figure 2.8(c) shows the impact of contention on the throughput. Under high contention (i.e., when the number of locks is small), the throughput is very limited. This is because the requests for the same lock have to be processed one by one, even though the switch still has spare

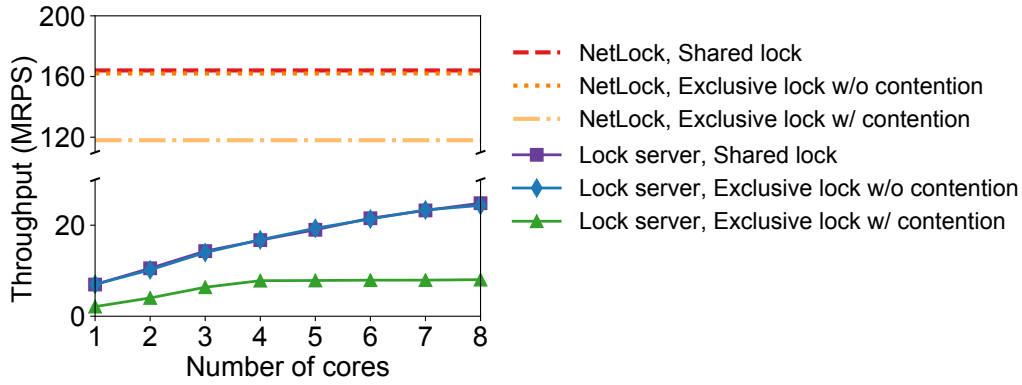
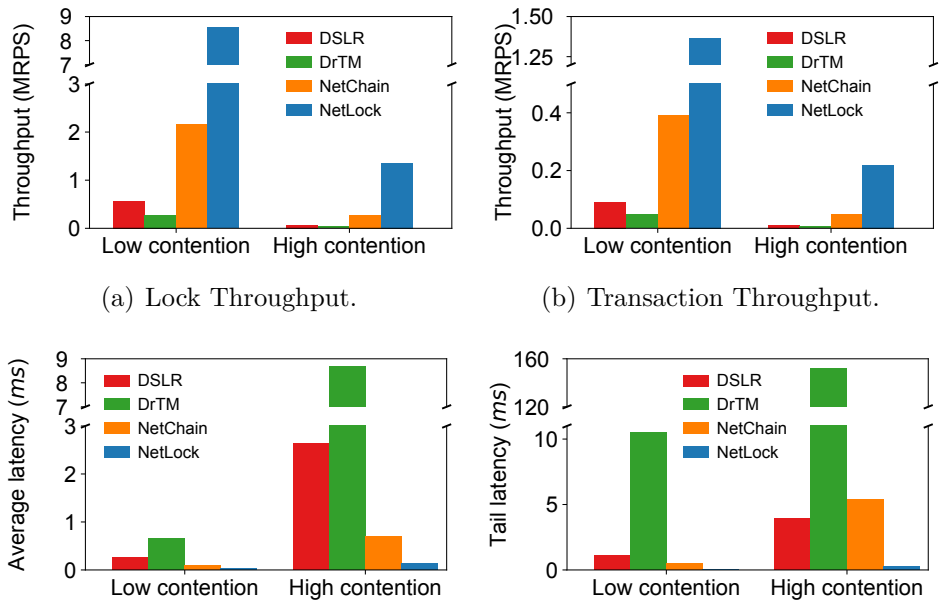


Figure 2-9. Comparison between a lock switch and a lock server with various number of cores. Ten servers are used to generate requests. The lock switch is not saturated. The lock switch can support a few billion requests per second.

capacity. The throughput increases as the contention decreases. Under low contention, the throughput is maximized by the speed of the 12 servers to generate lock requests. Figure 2.8(d) shows the latency results. The latency is more than 100 μs under high contention, and decreases to a few μs under low contention.

Comparison with lock server. We also compare the performance of a lock switch with a lock server. We use 10 servers to generate requests, and the workloads are similar to the previous experiments: shared locks, exclusive locks without contention, and exclusive locks with contention (5000 locks). The lock server is implemented with the same functionality and is configured with a different number of cores (1$\sim$8) in this experiment. Figure 2-9 shows the throughput of a lock switch and a lock server. The lock switch outperforms the lock server by 7$\times$ as the lock server easily gets saturated by a large number of requests. We emphasize that the lock switch is not saturated by the ten clients in this experiment. The performance gap would be even larger if there are more clients sending requests: the switch can process a few billion requests per second and can potentially replace hundreds of servers for the same functionality.



(a) Lock Throughput.

(b) Transaction Throughput.

(c) Average latency.

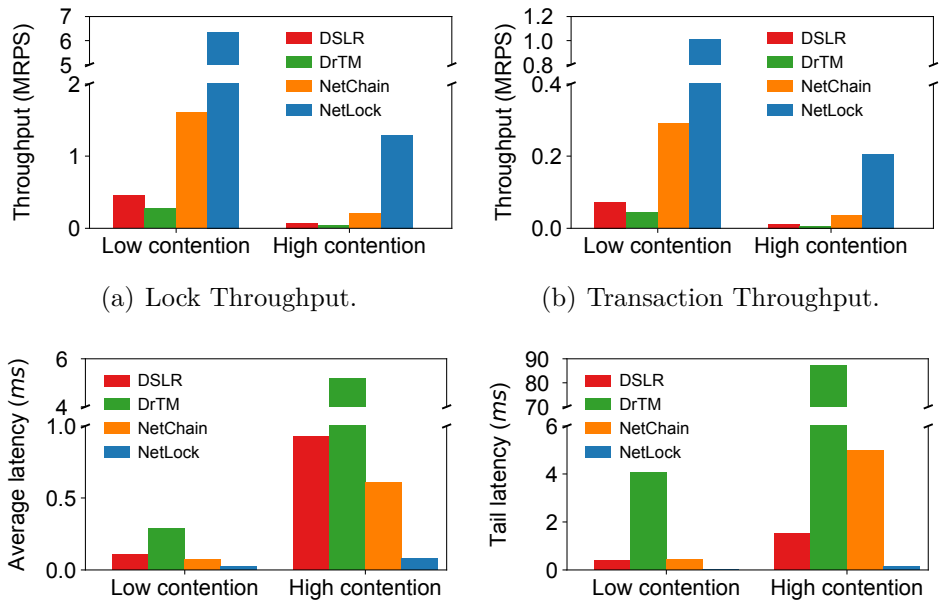
(d) Tail latency.

Figure 2-10. System comparison under TPC-C with ten clients and two lock servers.

2.6.3 Benefits of NetLock

We show the benefits of NetLock on its performance improvement and flexible policy support. The experiments use the TPC-C workload to show application-level performance.

Performance improvement over DSLR, DrTM and NetChain. We show the performance improvement of NetLock over the state-of-the-art solutions DSLR, DrTM and NetChain. We show two scenarios, and each is conducted under two TPC-C workload settings (high-contention and low-contention). Figure 2-10 shows the throughput and latency of the first scenario, where we use ten machines as clients to generate requests, and two machines as lock servers that run NetLock, DSLR or DrTM; NetChain only uses the switch, and does not use any servers for lock processing. Because NetChain treats both shared and exclusive locks as exclusive locks, it has many fail-and-retry operations which degrade its performance. With the co-design of the switch and lock servers, NetLock avoids a large number of fail-and-retry operations



(a) Lock Throughput.

(b) Transaction Throughput.

(c) Average latency.

(d) Tail latency.

Figure 2-11. System comparison under TPC-C with six clients and six lock servers.

caused by contentions compared to NetChain. The clients only need to retry when there is a packet loss or deadlock. By offloading using a fast switch to process most requests and avoiding most of retries, NetLock improves the transaction throughput by 14.9× (28.6×, 3.5×) and 18.4× (33.5×, 4.4×) in low and high contention settings respectively compared with DSLR (DrTM, NetChain). Besides throughput, NetLock also reduces both the average and tail latencies, by up to 20.3× (66.8×, 5.4×) and 18.4× (653.9×, 23.1×) respectively compared with DSLR (DrTM, NetChain). Figure 2-11 shows the results of the second scenario, where we use six machines as clients and six machines as lock servers for NetLock, DrTM and DSLR, and NetChain only uses the switch for lock processing. While in this scenario the lock servers are less loaded than they are in the previous scenario, NetLock still achieves significant improvement. Compared to DSLR (DrTM, NetChain), it improves the transaction throughput by up to 17.5× (33.1×, 5.5×), and reduces the average and tail latency by up to 11.8× (65.6×, 7.7×) and 10.5× (602.8×, 34.4×) respectively.

Policy support. Besides performance, another benefit of NetLock is its flexible policy

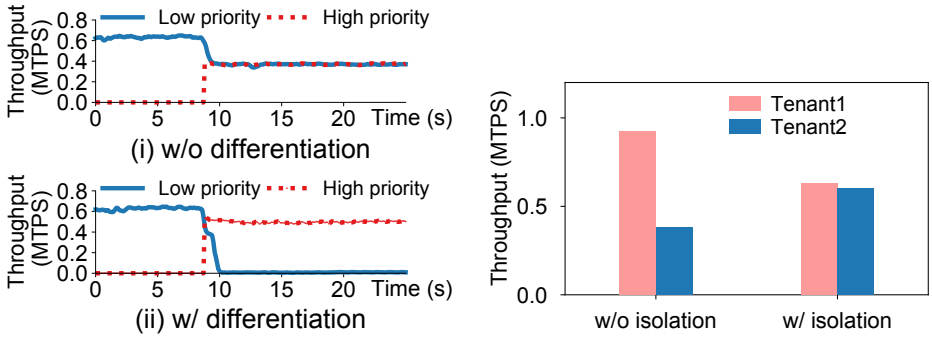
support. The default policy is starvation-freedom which helps reduce tail latency and is shown in the previous experiment. Here we show the other two representative policies mentioned in Section 2.4.4. Figure 2.12(a) shows how NetLock provides service differentiation with priorities. There are two tenants with five clients each. Without service differentiation, both tenants have similar performance when the high-priority tenant begins to send requests. With service differentiation, the high-priority tenant is prioritized over the low-priority tenant.

Figure 2.12(b) shows how NetLock enforces performance isolation. Different from the service differentiation experiment, we assign seven clients to tenant 1 and three clients to tenant 2. Because tenant 1 has more clients to generate requests at a faster rate than tenant 2, when there is no performance isolation, tenant 1 starves tenant 2 and achieves higher throughput. With performance isolation, each tenant can only obtain the tenant's own share, which is half of the resources here, and two tenants achieve similar performance.

2.6.4 Memory Management

We evaluate the efficiency of the memory allocation algorithm and the impact of the switch memory size on system performance. The experiments are conducted with ten clients and two lock servers under TPC-C workload (ten warehouses per node).

Memory allocation. NetLock uses an optimal knapsack algorithm to efficiently pack popular locks into limited switch memory to maximize system performance. We compare it with a strawman algorithm that randomly divides locks between the switch and the servers. Figure 2.13(a) shows the lock request throughput and its breakdown on the lock switch and the servers. Because the random approach does not allocate the switch memory to the popular locks, the switch only processes a small number of lock requests. On the other hand, NetLock efficiently utilizes the limited switch memory to process as many requests as possible, and improves the total



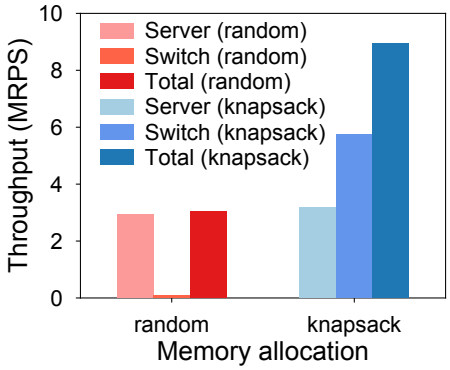
(a) Service differentiation. (b) Performance isolation.

Figure 2-12. Policy support of NetLock.

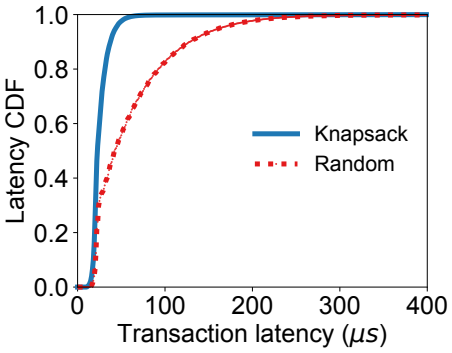
throughput by 2.95×. Figure 2.13(b) shows the latency CDF of the two algorithms. Because the random approach processes most lock requests in the lock servers, it incurs high latency, especially at the tail. In comparison, because of the efficient memory allocation, NetLock processes many requests directly in the switch and significantly reduces the transaction latency.

Switch memory size. As discussed in Section 2.4.5, the impact of switch memory size on the system performance depends on the think time and the memory allocation mechanism. Figure 2.14(a) shows the impact of memory size on throughput under different think times. The think time determines the maximum *turnover* rate of a memory slot, which limits the maximum throughput the switch can support with a given amount of memory. From the figure, we can see that when the think time is zero, the throughput quickly grows up with more memory slots and achieves 8.64 MRPS at the maximum. As the think time increases, the throughput is smaller and also grows more slowly. When the think time is 100 μs, the system can only achieve 0.60 MRPS because the memory in the switch is not efficiently utilized. Thus, NetLock is more suitable for low-latency transactions.

Figure 2.14(b) shows the impact of memory size on throughput under different memory allocation mechanisms. Because the knapsack algorithm used by NetLock can efficiently utilize switch memory, the throughput increases quickly with more memory

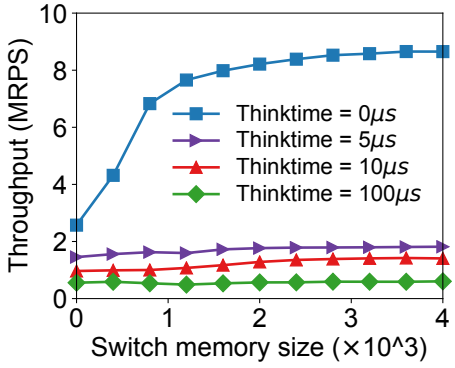


(a) Throughput.

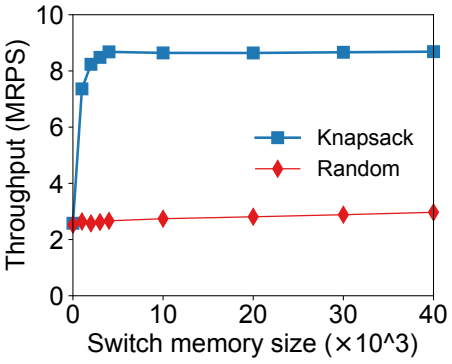


(b) Latency.

Figure 2-13. Impact of memory allocation mechanisms.



(a) Think time.



(b) Memory allocation.

Figure 2-14. Impact of memory size under different memory allocation mechanisms and think times.

slots, and reaches the maximum throughput of 8.61 MRPS with 3000 slots. We emphasize that the maximum throughput is bottlenecked by the speed of generating requests from the clients and the intrinsic contentions between the transactions, not the switch. On the other hand, because the random algorithm allocates the switch memory to a random set of locks, it utilizes the switch memory poorly. As a result, more memory slots does not help improve the transaction throughput of the system under the inefficient memory allocation algorithm. Under this workload, NetLock can achieve significant improvement with 5×10^3 memory slots (160KB), which is only a small fraction of the switch memory (tens of MB).

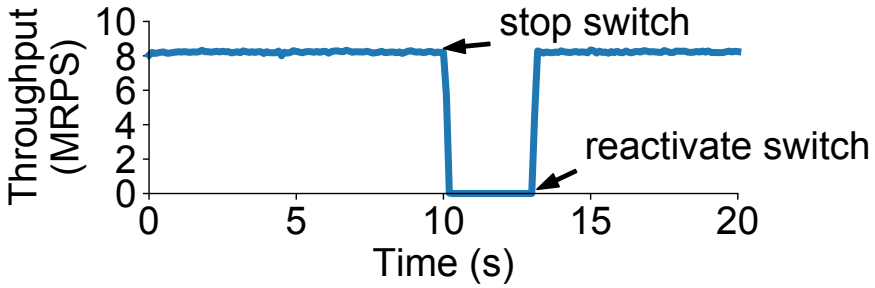


Figure 2-15. Failure handling result.

2.6.5 Failure Handling

We finally evaluate how NetLock handles failures. We manually stop the switch to inject a switch failure, and then reactivate the switch. Figure 2-15 shows the throughput time series. At time 10 s, we let the NetLock switch stop processing any packets. The system throughput drops to zero immediately upon the switch "failure". Then we reactivate the switch to process lock requests. The switch retains none of its former state or register values. During the switch failure, the client keeps retrying and requesting locks for their transactions. Upon reactivation, some lock requests of a transaction can be processed by the new (reactivated) switch while others may be lost. NetLock uses leasing to handle this situation. After reactivation, the system throughput returns to the pre-failure level instantly. NetChain can be applied to chain several NetLock switches to further reduce the temporary downtime.

2.7 Related Work

Lock management. Today's centralized lock managers are implemented on servers [9–13]. While they are flexible to support various policies, they suffer from limited performance. Recent work has exploited decentralized lock managers for high performance [7, 8, 14, 15]. These decentralized solutions achieve high performance at the cost of limited policy support. Compared to them, NetLock is a centralized lock manager that provides both high performance and the flexibility to support rich policies.

Fast distributed transactions. There is a long line of research on fast distributed transaction systems [23, 24, 27, 32–39]. These systems use a variety of techniques to improve performance, from designing new transaction algorithms and protocols, to exploiting new hardware capabilities like RDMA and hardware transactional memory. NetLock can be used as a fast lock manager to improve general transactions without any modifications to transaction protocols.

In-network processing. Recently there have been many efforts exploiting programmable switches for distributed systems, such as key-value stores [40–43], coordination and consensus [17, 44–48], network telemetry [49, 50], machine learning [51, 52], and query processing [53]. Kim *et al.* [16] proposes to extend switch memory with server memory using RDMA. NetLock provides a new solution for lock management, does not rely on RDMA, and includes an optimal memory allocation algorithm to integrate switch and server memory for the lock manager.

2.8 Conclusion

We present NetLock, a new centralized lock management architecture that co-designs programmable switches and servers to simultaneously achieve high performance and rich policy support. NetLock provides orders-of-magnitude higher throughput than existing systems with microsecond-level latency, and supports many commonly-used policies on performance and isolation. With the end of Moore's law, we believe NetLock exemplifies a new generation of systems that leverage network programmability to extend the boundary of networking to IO-intensive workloads.

Chapter 3

HCSFQ: Hierarchical Core-Stateless Fair Queueing.

3.1 Introduction

Fair queueing is a canonical mechanism to provide fair bandwidth allocation to network traffic by ensuring that each flow gets its fair share irrespective of the other flows. This way, fair queueing enforces isolation between competing flows, which ensures that normal flows are protected from ill-behaving flows. There is a long history of research on fair queueing [54–65]. Many of the proposed solutions require to maintain per-flow state in the switch, and rely on complex data structures and scheduling algorithms to realize fair queueing.

Core-Stateless Fair Queueing (CSFQ) [66] is a scalable algorithm to realize fair queueing. Compared to the alternatives, CSFQ has the unique property that it does not maintain per-flow state in the network. With CSFQ, the sources or switches at the edge classify traffic into flows and estimate per-flow rate. In turn, the switches in the network estimate the fair rate, and use probabilistic dropping to regulate each flow to its fair rate without maintaining per-flow state.

While CSFQ was proposed more than twenty years ago, it has not taken off. This is primarily due to two reasons. First, it requires changes to the IP protocol (i.e., adding a field to the IP header) and coordination across all switches (routers) in the

network. Second, CSFQ requires switches to estimate the fair rate, compute a drop probability, and update the header of each packet. To perform these operations at line rate we need hardware support. These challenges are exacerbated by the fact that routers belong to different, often competing, Internet Service Provides (ISPs), which would all need to cooperate to upgrade their infrastructures to support CSFQ.

However, two emerging technologies are making CSFQ relevant again: (*i*) the advent of cloud computing and (*ii*) the increased popularity of programmable switches. Cloud providers own large datacenters consisting of many thousands of servers. Since a datacenter is typically owned by a single administrative entity (cloud provider) that controls both the software and hardware, it is relatively easy for a cloud provider to upgrade all its switches and servers to support CSFQ. FairCloud [67] proposes to apply CSFQ for network isolation in datacenters, but it does not have a hardware implementation for CSFQ. The emergence of programmable switches makes it possible to implement sophisticated packet processing at line rate. In particular, as we will show in this chapter, existing programmable switches are powerful enough to support CSFQ at line rate.

While datacenter deployment removes the adoption barriers for CSFQ, it also raises new challenges. In particular, while CSFQ has been designed for a flat hierarchy, the traffic in today's datacenters is naturally structured in a multi-level hierarchy. For example, at the top level we typically have tenants and at the bottom level we have the flows of those tenants. The mechanism of choice to manage such traffic is hierarchical fair queueing [62, 63, 68], where each non-leaf node distributes its excess bandwidth (i.e., the bandwidths unused by some of its children) across its children. This allocation policy is consistent with a per-tenant payment granularity, i.e., network resources are divided between tenants in proportion to their payments [67]. In this case, if a flow of a tenant stops sending data, that tenant would want to re-allocate the flow's bandwidth to its other flows, and not to the flows of other tenants in the

datacenter.

However, implementing hierarchical fair queueing is challenging. Existing solutions require per-flow state, and more importantly, require complex queue management and packet transfers in a hierarchy of queues [62, 63, 68]. Because of the implementation complexity, hierarchical fair queueing is not supported by today's high-speed hardware switches.

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

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

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

In summary, we make the following contributions.

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

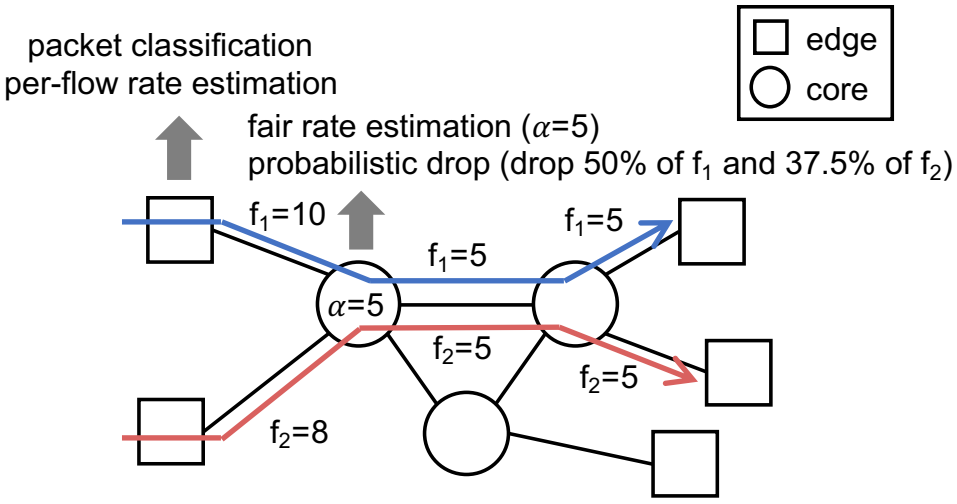


Figure 3-1. Core-Stateless Fair Queueing.

3.2 Background and Motivation

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

3.2.1 Core-Stateless Fair Queueing

Fair queueing provides max-min fairness for competing flows. A max-min fair bandwidth allocation is one that any increase of the allocation to some flows would necessarily decrease the allocation of some other flows. The basic way to realize fair queueing in a switch is to keep one queue for each flow and use a scheduling algorithm to pick which queue to dequeue a packet each time. There has been decades of research on fair queueing [54–65]. While we leave the extensive discussion to related work (§3.7), we emphasize that most solutions are not scalable because of the need to maintain *per-flow* state to classify flows and shape their rates with per-flow queues and complex queue management. As a result, commodity switches only support 10–20 queues.

CSFQ is a *scalable* algorithm to achieve fair queueing with a unique property that

it does *not* maintain per-flow state in the network. Figure 3-1 shows the architecture of CSFQ. CSFQ divides the network into edge and core. The switches or hosts at the edge, which do maintain per-flow state, use per-flow state to classify packets into flows and estimate per-flow arrival rate. Then the arrival rate of each flow is *carried* in a custom packet *header*. The switches in the core only estimate the *total* arrival rate of all flows, and then use it to estimate the *fair share rate* with an iterative algorithm. The switches compare the per-flow arrival rate in the packet header with the fair share rate to compute a drop probability, and drop packets to shape the rate of each flow to the fair share rate.

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

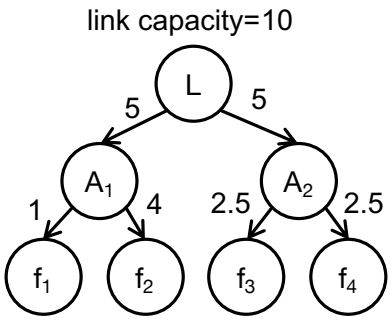
3.2.2 Opportunities

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

The first opportunity is from cloud computing. Cloud computing has become the fundamental infrastructure of today's Internet. Datacenters power large-scale Internet services we use everyday such as search, social networking and e-commerce, and enterprises are increasingly moving their workloads to the cloud. Fair bandwidth allocation and network isolation for datacenter networks is an important problem [67, 69–81]. While there has been many fair queueing algorithms proposed in the past [54–65], they are rarely deployed in practice because they need to maintain per-flow

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

(a) Fair queueing.



(b) Hierarchical fair queueing.

Figure 3-2. Fair queueing and hierarchical fair queueing.

state in switches but switches can only support 10–20 queues. CSFQ provides a scalable solution to address this problem. Datacenter operators control the entire infrastructure, including both software and hardware. Adopting CSFQ to enforce isolation for datacenter networks naturally eliminates the need of cooperation between different operators or ISPs, as a datacenter network is under a single administrative domain.

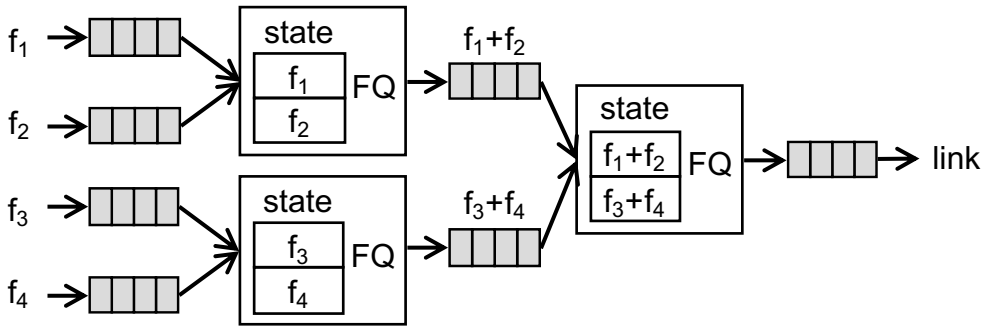
The second opportunity is from programmable switching ASICs. Traditional switching ASICs are fixed-function, and adding a new feature like CSFQ requires switch vendors to design a new ASIC. Emerging programmable switching ASICs, such as Intel Tofino [19], Broadcom Trident 4 [20] and Cavium XPliant [21], allow users to program the data plane and develop new features. Specifically, to implement CSFQ on a programmable switch, we can program the parser to parse the custom header of CSFQ (to carry per-flow rate), program the match-action tables to implement the CSFQ algorithm, and program the on-chip memory to store the aggregate state. Because a datacenter network is under a single administrative domain, it is easy for the operator to adopt the protocol and hardware changes with programmable switching ASICs.

3.3 Hierarchical Fair Queueing

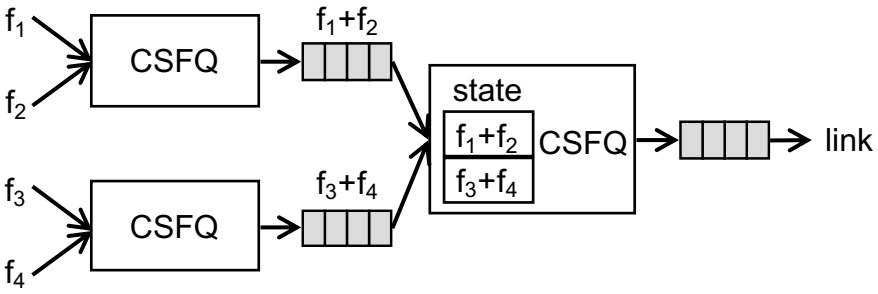
A multi-tenant cloud has a natural two-layer hierarchy, with the tenants at the first layer and the flows of each tenant at the second layer. Network isolation for multi-tenant datacenters naturally requires hierarchical fair queueing. CSFQ only supports fair queueing, but not hierarchical fair queueing.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

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

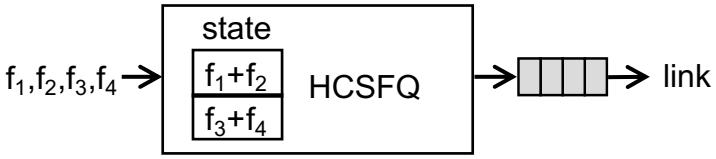
Example. We use an example in Figure 3-2 to contrast hierarchical fair queueing with fair queueing. There are four flows, i.e., f_1, f_2, f_3, and f_4. The arrival rates of the four flows are 1, 4, 5, and 5, respectively. The link capacity is 10. With only fair queueing, the unused share of f_1 is evenly allocated to all other three flows. As shown in Figure 3-2(a), the bandwidth allocation to the four flows is (1, 3, 3, 3).



(a) Traditional design of hierarchical fair queueing.



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



(c) HCSFQ design.

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

Suppose that f_1 and f_2 are in one flow aggregate (A_1), and f_3 and f_4 are in the other (A_2). With hierarchical fair queueing, the unused fair share of f_1 is only allocated to f_2, instead of also being shared by f_3 and f_4. Figure 3-2(b) shows the bandwidth allocation with two-layer hierarchical fair queueing, where the flows receive 1, 4, 2.5, and 2.5, respectively.

Challenge. Hierarchical fair queueing is known to be challenging to realize in switches at high speed. A traditional design to support hierarchical fair queueing is to leverage a hierarchy of queues, and each node in the hierarchy implements fair queueing for the queues of its child nodes. Figure 3-3(a) shows an example of such a design to support

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

3.4 HCSFQ Design

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

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

The major distinction between HCSFQ and CSFQ is that HCSFQ needs to maintain the state of interior nodes. This is necessary because HCSFQ aims to provide hierarchical fair bandwidth allocation for a flow hierarchy. Note that the naive design of extending CSFQ in Figure 3-3(b) also requires maintaining the state of interior

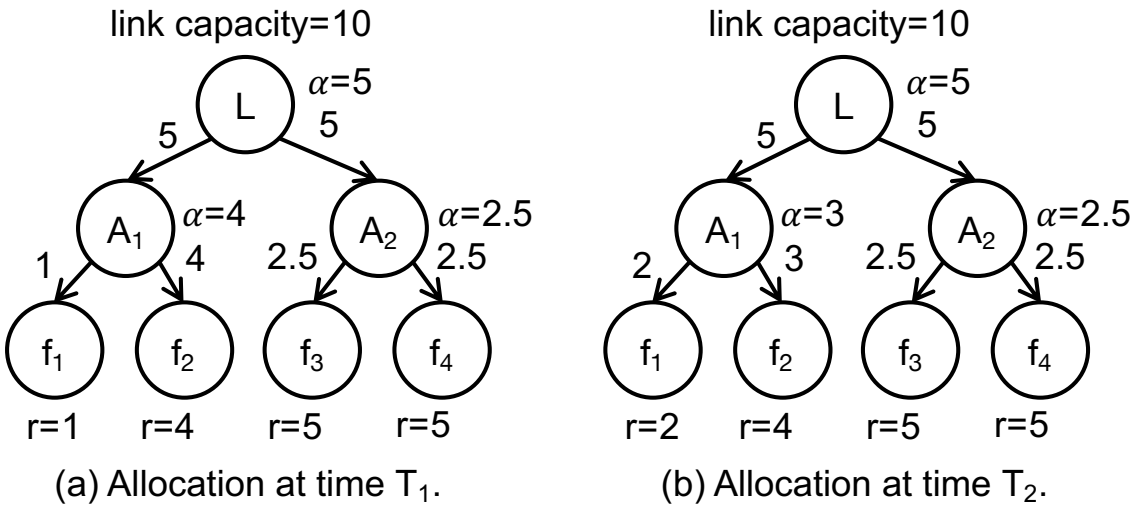


Figure 3-4. The flow arrival rates change from T_1 to T_2. It is necessary for the switch to keep the state for the interior nodes of the hierarchy in order to realize hierarchical fair queueing.

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

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

3.4.1 Fluid Model

We first use a fluid model to formalize hierarchical fair queueing. The fluid model considers a switch with output link capacity C, and the flows are modeled as a continuous stream of bits. The flow hierarchy is represented as a directed graph $G(V, E)$, where V is the set of nodes and E is the set of edges. A node $v \in V$ represents a flow aggregate (i.e., a set of flows), where $r(v)$ is the arrival rate of the flow aggregate and $c(v)$ is the capacity allocated to v. A directed edge $e(v, u) \in E$ represents that u is a child of v.

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

$$c(v) = \sum_{e(v,u)\in E} \min(\alpha(v), r(u)). \tag{3.1}$$

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

$$\alpha(v) = \max_{e(v,u)\in E} r(u). \tag{3.2}$$

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

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

$$\max(0, 1 - \frac{\alpha(v.parent)}{r(v)}). \tag{3.3}$$

3.4.2 HCSFQ Algorithm

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

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

$$r(v)_{new} = (1 - e^{T_i/K})\frac{l_i}{T_i} + e^{T_i/K} r(v)_{old}, \tag{3.4}$$

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

Fair share rate estimation. The fair share rate estimation is used to estimate the fair share rate that a node allocates to its children. The capacity of node v is $c(v)$.

Eq.(3.4) gives the arrival rate of the node $r(v)$. If $r(v) \leq c(v)$, then $\alpha(v)$ is calculated using Eq.(3.2). Otherwise, $\alpha(v)$ should be the unique solution to Eq.(3.1). We apply the iterative algorithm in CSFQ to approximately solve the equation. Specifically, for each node v, we maintain the accepted rate estimation $f(v)$, which is updated with Eq.(3.4) if the packet is not dropped. Then, $\alpha(v)$ is approximately computed with the following formula,

$$\alpha(v)_{new} = \alpha(v)_{old} \frac{c(v)}{f(v)}. \tag{3.5}$$

Note that the computation of $\alpha(v)$ is iterative. It converges to the solution of Eq.(3.1) after several iterations, i.e., processing several packets. Similar to CSFQ, HCSFQ also uses a window of size K_c to account for inaccuracies introduced by exponential averaging in rate estimation. That is, $\alpha(v)$ is updated only if the node is congested ($r(v) > c(v)$) or uncongested ($r(v) \leq c(v)$) for an interval of length K_c.

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

- *pkt.r* is the arrival rate estimate of the flow the packet belongs to.
- *pkt.nodes* is a list of node IDs that indicate the flow aggregates the packet belongs to in the flow hierarchy, excluding the leaf. For example, in Figure 3-2, if a packet *pkt* belongs to f_1 or f_2, then $pkt.nodes = [L, A_1]$.

CSFQ only carries *pkt.r* in the packet header as there is no flow hierarchy. HCSFQ additionally carries *pkt.nodes* to track the set of flow aggregates the packet belongs to in the hierarchy. Similar to CSFQ, both *pkt.r* and *pkt.nodes* are inserted at the edge. An edge switch (e.g., a software switch, a NIC or a ToR switch in datacenter networks) performs packet classification to get *pkt.nodes*, and uses Eq.(3.4) to estimate the flow rate *pkt.r*. Both *pkt.r* and *pkt.nodes* are transparent to end hosts and are removed

Algorithm 4 HCSFQ(pkt)

```
1: cur_α ← 0
2: for v ∈ pkt.nodes do
       // estimate arrival rate
3:     r[v] ← estimate_rate(pkt)
4:     cur_α ← α[v]
   // calculate drop probability
5: prob ← max(0, 1 − cur_α/pkt.r)
6: if prob > rand(0, 1) then
7:     drop_flag ← TRUE
8: for v ∈ pkt.nodes do
       // estimate accepted rate
9:     if drop_flag is False then
10:        f[v] ← estimate_rate(pkt)
       // allocate bandwidth
11:    if v is root then
12:        c[v] ← link capacity
13:    else
14:        c[v] ← min(α[v.parent], r[v])
       // update fair share rate
15:    if r[v] > c[v] then
16:        if congest_flag[v] is FALSE then
17:            congest_flag[v] ← TRUE
18:            start_time ← current_time
19:        else if current_time − start_time > K_c then
20:            α[v] ← α[v] · c[v]/f[v]
21:            start_time ← current_time
22:    else
23:        if congest_flag[v] is TRUE then
24:            congest_flag[v] ← FALSE
25:            start_time ← current_time
26:            tmp_α[v] ← 0
27:        else if current_time − start_time ≤ K_c then
28:            child_r ← v.next = NULL ? pkt.r : r[v.next]
29:            tmp_α[v] ← max(tmp_α[v], child_r)
30:        else
31:            α[v] ← tmp_α[v]
32:            start_time ← current_time
33:            tmp_α[v] ← 0
34:    cur_α ← α[v]
   // drop or enqueue pkt
35: if drop_flag then
36:    drop(pkt)
37: else
38:    enqueue(pkt)
   // update the packet rate
39: pkt.r ← min(cur_α, pkt.r)
```

by the switch at the last hop.

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

Algorithm 4 shows the pseudo code of the HCSFQ algorithm. When a packet *pkt* arrives at the switch, the switch updates the arrival rate estimate for each flow aggregate the packet belongs to using Eq.(3.4), and gets the fair share rate of the flow (line 1-4). Then the switch computes the dropping probability based on Eq.(3.3) and decides whether to drop the packet (line 5-7). After this, the switch recursively updates the fair share rate of each flow aggregate in the hierarchy (line 8-34). Based on whether the packet is dropped, the switch updates the accepted rate estimate for each flow aggregate (line 9-10). If node v is the root, then all flows are under this node, and its allocated capacity is the link capacity (line 11-12); otherwise, its allocated capacity is the max of the fair share rate allocated by its parent and its arrival rate (line 13-14). If the arrival rate of v is bigger than its allocated capacity, then the node is congested, and the fair share rate is updated based on Eq.(3.5) (15-21); otherwise, the fair share rate is the max arrival rate of its children, i.e., based on Eq.(3.2) (line 22-33). Note that we use a window of length K_c for fair share update to account for inaccuracies in rate estimation. Based on the dropping decision, the switch drops or

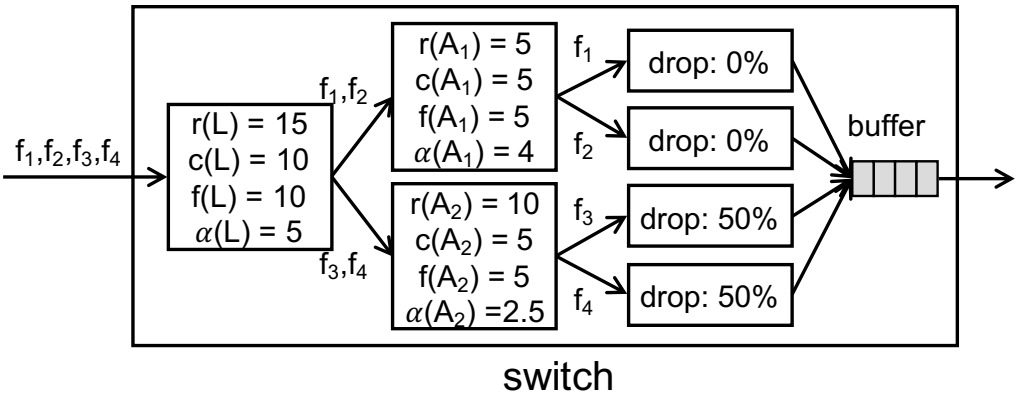


Figure 3-5. Example of the HCSFQ algorithm to provide hierarchical fair queueing for the scenario in Figure 3-2(b).

enqueues the packet (line 35-38). Finally, the arrival rate $pkt.r$ is updated and will be used by the next-hop switch (line 39). Note that the loops (line 2-4 and line 8-34) are done in one pass and the fair share rate is updated based on $c[v.parent]$ from the last round.

Figure 3-5 illustrates how the algorithm works to realize hierarchical fair queueing for the example in Figure 3-2. At the root, the total arrival rate of all flows $r(L)$ is 15, and the capacity $c(L)$ is the link capacity 10, which is below the arrival rate. The root fairly allocates the capacity to the two flow aggregates, A_1 and A_2. The figure shows the stable state when the accepted rates and fair share rates of all the nodes have converged. After convergence, the accepted rate $f(L)$ is 10, and the fair share rate $\alpha(L)$ is 5. At node A_1, the arrival rate $r(A_1)$, which is the sum of $r(f_1)$ and $r(f_2)$, is 5, and the allocated capacity $c(A_1)$ is 5. The fair share rate is set as 4, and there is no need to drop packets for f_1 and f_2. At node A_2, the arrival rate $r(A_2)$, which is 10, is bigger than the allocated capacity, which is 5. A_2 allocates its capacity to f_3 and f_4 fairly. Each receives a fair share rate of 2.5. So the switch drops 50% of the packets for both f_3 and f_4.

Weighted HCSFQ. The HCSFQ algorithm can be extended to support flows and flow aggregates with weights. For node v, we use $w(v)$ to represent the weight of the

flow or flow aggregate of v. Under max-min fair bandwidth allocation, competing flows or flow aggregates at the bottlenecked link have the same fair share rate $r(v)/w(v)$. There are two changes to the algorithm in order to incorporate the weight. The first change is on the equation to compute the fair rate $\alpha(v)$ when $r(v) > c(v)$. Eq.(3.1) is changed to

$$c(v) = \sum_{e(v,u)\in E} w(u) \cdot min(\alpha(v), \frac{r(u)}{w(u)}). \tag{3.6}$$

The second change is on the equation to compute the drop probability. Eq.(3.3) is changed to

$$\max(0, 1 - \alpha(v.parent) \cdot \frac{w(v)}{r(v)}). \tag{3.7}$$

3.4.3 Theoretical Guarantee

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

Theorem 2. *Consider a link with a hierarchical fair queueing policy and a flow in the hierarchy. Let w_1, w_2, ..., w_n be the weights of the nodes from the root to the flow. Let α_1, α_2, ..., α_n be the constant normalized fair rate of the nodes from the root to the flow. Let $r_{\alpha_i} = \alpha_i w_i$. If probabilistic dropping is applied at the last layer, then the excess service received by the flow that sends at a rate at no larger than R, is bounded above by*

$$r_{\alpha_n} K(1 + ln\frac{R}{r_{\alpha_n}}) + l_{max} \tag{3.8}$$

where l_{max} is the maximum packet length.

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

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

The the parent node does not drop packets.

Proof. The first conclusion is directly derived from the guarantee of CSFQ [66].

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

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

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

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

$$r'_i \leq r_{\alpha'}, \quad \forall i. \tag{3.9}$$

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

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

And by definition,

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

We now recursively prove Eq. (3.9).

(i) First let $i = 1$.

We will use the following inequality to prove Eq. (3.9):

$$(1 - e^{-T_1/K}) \frac{h_1^{(j)}}{T_1} \leq r_\alpha^{(j)}, \quad \forall j. \tag{3.10}$$

On the one hand, if Eq. (3.10) is true, we have

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

which implies Eq. (3.9) for $i = 1$.

On the other hand, recall $r_1^{(j)} = (1 - e^{-T_1/K})\frac{\ell_1^{(j)}}{T_1}$, we then prove Eq. (3.10) as following:

1. If $r_1^{(j)} < r_\alpha^{(j)}$, then $h_1^{(j)} = \ell_1^{(j)}$, thus
$$(1 - e^{-T_1/K})\frac{h_1^{(j)}}{T} = (1 - e^{-T_1/K})\frac{\ell_1^{(j)}}{T} = r_1^{(j)} \le r_\alpha^{(j)}.$$
2. If $r_1^{(j)} \ge r_\alpha$, then $h_1^{(j)} = \ell_1^{(j)}\frac{r_\alpha^{(j)}}{r_1^{(j)}}$, thus
$$(1 - e^{-T_1/K})\frac{h_1^{(j)}}{T_1} = (1 - e^{-T_1/K})\frac{\ell_1^{(j)}}{T_1}\frac{r_\alpha^{(j)}}{r_1^{(j)}} = r_\alpha^{(j)}.$$

Thus Eq. (3.10) holds.

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

We will use the following inequality to prove our claim:
$$(1 - e^{-T_i/K})\frac{h_i^{(j)}}{T_i} \le r_\alpha^{(j)}, \quad \forall j. \tag{3.11}$$

On the one hand, if Eq. (3.11) is true, we have
$$\begin{aligned} r'_i =& (1 - e^{-T_i/K})\frac{\sum_{j=1}^k h_i^{(j)}}{T_i} + e^{-T_i/K} r'_{i-1} \\ \le& \sum_{i=1}^k r_\alpha + e^{-a/K} r'_\alpha \le r'_\alpha, \end{aligned}$$
which implies Eq. (3.9) for i.

On the other hand, recall
$$r_i^{(j)} = (1 - e^{-T_i/K})\frac{\ell_i^{(j)}}{T_i} + e^{-T_i/K} r_{i-1}^{(j)},$$
we then prove Eq. (3.11) as following:

1. If $r_i^{(j)} < r_\alpha^{(j)}$, then $h_i^{(j)} = \ell_i^{(j)}$, thus
$$\begin{aligned} (1 - e^{-T_i/K})\frac{h_i^{(j)}}{T_i} =& (1 - e^{-T_i/K})\frac{\ell_i^{(j)}}{T_i} \\ =& r_i^{(j)} - e^{-T_i/K} r_{i-1}^{(j)} \\ \le& r_i^{(j)} \le r_\alpha^{(j)}. \end{aligned}$$

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

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

Thus Eq. (3.10) holds. By (i) and (ii) and mathematical induction our proof is finished. □

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

3.5 Data Plane Design and Implementation

In this section, we describe a data plane design to implement CSFQ and HCSFQ on new-generation programmable switches. Programmable switches enable users to program the multi-stage match-action pipeline in the switch data plane to implement custom features. Users can also access the on-chip memory and implement stateful operations using the register arrays provided by programmable switches. Programmable switches also support a set of primitive actions (e.g., recirculate, bit shift, add and subtract) which make HCSFQ possible. Based on the constructs of programmable switches, we show how to design and implement the rate estimation, the fair rate computation and the flow shaping logic (i.e., Algorithm 4) on programmable switches. Our HCSFQ implementation contains 1952 lines of code in P4 and is compiled to Intel Tofino ASIC [19]. The code is open-source and available at https://github.com/netx-repo/HCSFQ.

3.5.1 Single Layer

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

Rate estimation. The switch needs to estimate two rates: the total arrival rate r, and the accepted rate f. Both rates are estimated with Eq.(3.4). Because switches have strict timing and resource requirements, an action in a match-action table can only contain a small number of operations in a limited operation set. The equation cannot be directly implemented in the switch data plane due to two reasons. First, the equation involves several multiplication, division and exponentiation operations on floating points. These operations are quite complex and require multiple clock cycles to compute. As such, they are not typically supported by the switch data plane. Second, a rate (r or f) is stored in a register of the on-chip memory. To update the rate, the switch needs to read the rate from the register, uses the equation to calculate the new rate, and then updates the register. A register can only be accessed by its own stage, but the equation includes multiple arithmetic operations, which requires multiple stages to compute.

We leverage the high-precision timestamps available in the data plane, and use a window-based mechanism to estimate the rates. Programmable switches are able to provide high-precision timestamps at the granularity of one nanosecond. To estimate a rate, the switch maintains a pair of registers ($reg.byte$ and $reg.start$). One register ($reg.byte$) stores the total bytes of packets the switch has received in the current window. The other register ($reg.start$) stores the start timestamp of the current window. For each incoming packet, the switch first checks the current timestamp and compares it with $reg.start$ to see if the packet belongs to the current window. If so, the

switch adds the size of the packet to $reg.byte$; otherwise, the switch clears $reg.byte$ and sets $reg.start$ to the current timestamp. The switch keeps another register $reg.rate$ to store the current rate estimate. When a window is passed, the switch uses $reg.byte$ to update $reg.rate$, which can be done with either a direct assignment, or a moving average. Our experiments indicate that using a moving average (implemented with several bit shift and addition operations) works better and avoids oscillation with the control loop that updates the fair share rate and drops packets.

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

Probabilistic drop. Probabilistic drop is used to regulate the flows to the fair share rate. The switch uses the fair share rate α and the flow arrival rate r to compute the probability to drop packets of the flow (Eq.(3.3) and line 5 in Algorithm 4). Then the switch checks the condition $max(0, 1 - \alpha/r) > rand(0, 1)$ to decide whether to drop an incoming packet or not. Similar to rate estimation, the challenge is that switches do not support the division operation to compute the probability. One way to solve the problem is to use a similar window-based mechanism as rate estimation, i.e., divide time into windows with window size δ, and keep counters to allow up to $r\delta$ packets to pass in each window and drop all remaining packets. The drawback of this approach is that it introduces bursty packet drops, which do not work well with congestion control. We want to mimic the behavior of CSFQ to have random packet drops that are uniformly distributed in the packet stream.

We discretize the probability computation to approximate the drop probability with bounded error. We leverage the random number generator provided by the data plane and use multiple stages to realize the discretized computation. Specifically, to check

the condition $max(0, 1 - \alpha/r) > rand(0, 1)$, it is sufficient to check $rand(0, 1) > \alpha/r$. We multiply r to both sides of the inequality, and transform the condition to

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

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

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

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

Fair rate update. When the link is congested, the fair share rate is the unique solution to Eq.(3.1). Because HCSFQ does not maintain per-flow state, it uses $\alpha_{new} = \alpha_{old}c/f$ (Eq.(3.5)) to approximately compute the fair share rate, where c is the capacity and f is the accepted rate. Like rate estimation and probabilistic

drop, Eq.(3.5) cannot be supported because it contains multiplication and division. What is more challenging is that the fair rate update introduces the following circular dependency to the packet processing.

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

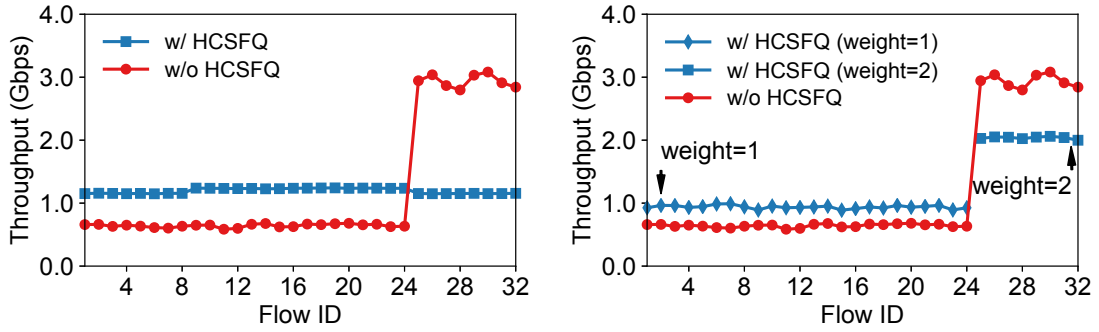
Specifically, the switch needs to read α to compute the drop probability. Then based on whether to enqueue or drop a packet, the switch updates the accepted rate f, which is then used to update α. Because a register can only be accessed by its own stage, the new value of α cannot be used to update the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

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

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

3.5.2 Multiple Layers

The single-layer design is used as a building block to support multiple layers. As shown in Algorithm 4 and Figure 3-5, the processing of HCSFQ on a packet is performed layer by layer, from the root to the leaf node. This well matches the multi-stage packet processing pipeline of programmable switches. The layers in HCSFQ can be mapped to the stages in the pipeline, which naturally processes packets sequentially stage by stage. The major difference between HCSFQ and CSFQ is that HCSFQ needs to store more states as it has multiple layers. CSFQ is a single-layer HCSFQ and only maintains the state for three variables, which are the total arrival rate r, the accepted rate f, and the fair share rate α. Each variable use multiple registers as described in §3.5.1. HCSFQ maintains the state for all interior nodes, each of which includes the three variables. Commodity switches have 10-100 MB on-chip memory [82], which is able to support a large number of interior nodes. For a two-layer HCSFQ for tenant-level and flow-level isolation in multi-tenant datacenters, a switch needs to maintain per-tenant state, but not per-flow state. With 10-100 MB memory, the switch can support millions of tenants. In terms of the number of layers, our prototype supports up to four layers on Intel Tofino. There is no theoretical limit on the number of layers given the scalable algorithm design. The constraints for practical implementations mainly come from the restricted hardware primitives to implement the algorithm as we describe in §3.5.1. These constraints are not fundamental. Newer programmable switches (e.g., Intel Tofino 2) have more stages and provide more hardware primitives to support more layers. Despite this, we expect HCSFQ with 2–4 layers should be sufficient to provide hierarchical isolation for many datacenter scenarios (e.g., multi-tenancy).



(a) Equal weights.

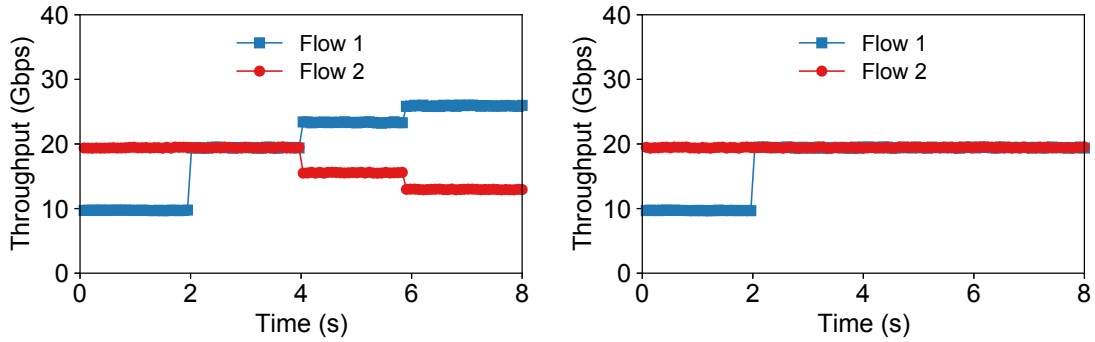
(b) Different weights.

Figure 3-6. Testbed experiments of fair queueing for UDP. Flow 1–24 send at 2Gbps and Flow 25-32 send at 8Gbps.

3.6 Evaluation

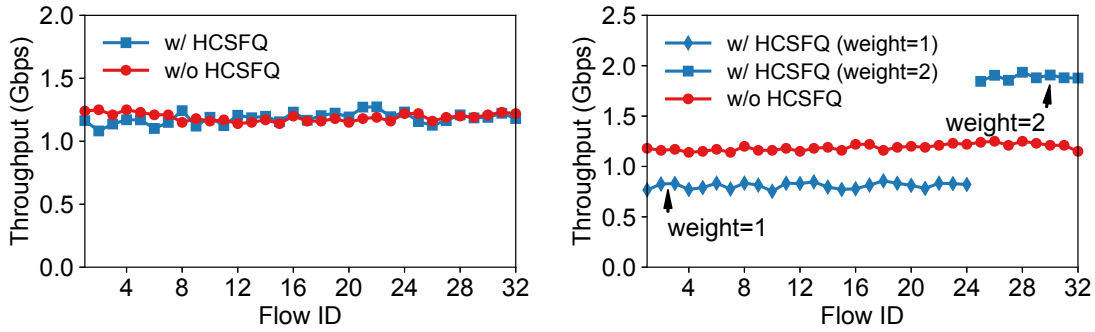
In this section, we provide experimental results to demonstrate the performance of HCSFQ. We first evaluate the performance of single-layer HCSFQ (i.e., CSFQ), and show that it can provide fair queueing (§3.6.1). We then evaluate the performance of two-layer HCSFQ, and show that it can provide hierarchical fair queueing to enforce tenant-level and flow-level isolation for multi-tenant datacenters (§3.6.2). Finally, we use simulations to evaluate HCSFQ in a large-scale datacenter environment and compare it with several alternatives (§3.6.3).

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



(a) w/o HCSFQ. (b) w/ HCSFQ.

Figure 3-7. Testbed experiments of fair queueing for UDP. Flow 1 is sending at a different rate every 2 seconds. Flow 2 is sending at 20Gbps.



(a) Equal weights. (b) Different weights.

Figure 3-8. Testbed experiments of fair queueing for TCP.

3.6.1 Fair Queueing Experiments

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

UDP. If all UDP flows have the same sending rate, they would get similar bandwidth under the tail-drop FIFO queue in the switch. To make the experiment more interesting, we assign different sending rates to the UDP flows. We let 24 flows (Flow 1–24) send

at 2Gbps and 8 flows (Flow 25–32) send at 8Gbps. As shown in Figure 3.6(a), without HCSFQ, Flow 25–32 obtain higher bandwidth than Flow 1–24 because Flow 25–32 have larger sending rates. In comparison, HCSFQ is able to fairly allocate bandwidth to the flows.

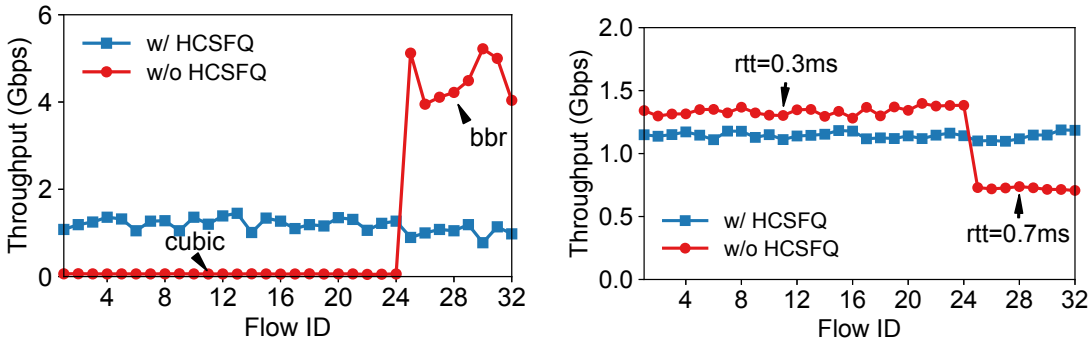
HCSFQ supports weighted fair queueing. We assign weight 1 to Flow 1–24 to and weight 2 to Flow 25–32. As shown in Figure 3.6(b), without HCSFQ, the result is the same as that with equal weights in Figure 3.6(a). On the other hand, HCSFQ is able to allocate the bandwidth based on the weights. Flow 25–32 achieve higher throughput than Flow 1–24.

We also evaluate HCSFQ when the UDP flows dynamically change their rates. As shown in Figure 3-7, we let Flow 1 send at a different rate every 2 seconds (10Gbps, 20Gbps, 30Gbps and 40Gbps, respectively) and let Flow 2 keep sending at 20Gbps. Without HCSFQ, when the link is congested (from 4s to 8s), each flow achieves a throughput in proportional to its sending rate. With HCSFQ, two flows get the fair share (20Gbps) when the link is congested.

TCP. Figure 3.8(a) shows the throughput of the flows with and without HCSFQ. Because TCP congestion control provides fair bandwidth allocation, the flows have similar throughput even without HCSFQ. Adding HCSFQ to the switch does not change the bandwidth allocation and thus has a similar result.

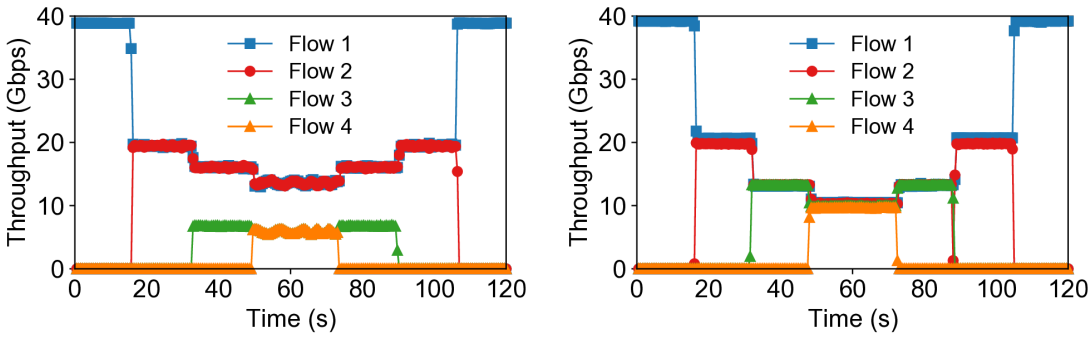
However, TCP cannot support weighted fair queueing. To show the benefits of HCSFQ, we let Flow 1–24 have weight 1 and Flow 25–32 have weight 2. Without HCSFQ, the result in Figure 3.8(b) is similar to that in Figure 3.8(a). With HCSFQ, the flows get bandwidth in proportional to their weights. The flows with higher weights (Flow 25–32) receive more bandwidth than those with lower weights (Flow 1–24).

Different TCP algorithms. There are many TCP congestion control algorithms. Without in-network enforcement, the flows using aggressive congestion control algo-



(a) Different TCP algorithms. (b) Different RTTs.

Figure 3-9. Testbed experiments of fair queueing for TCP under different configurations.

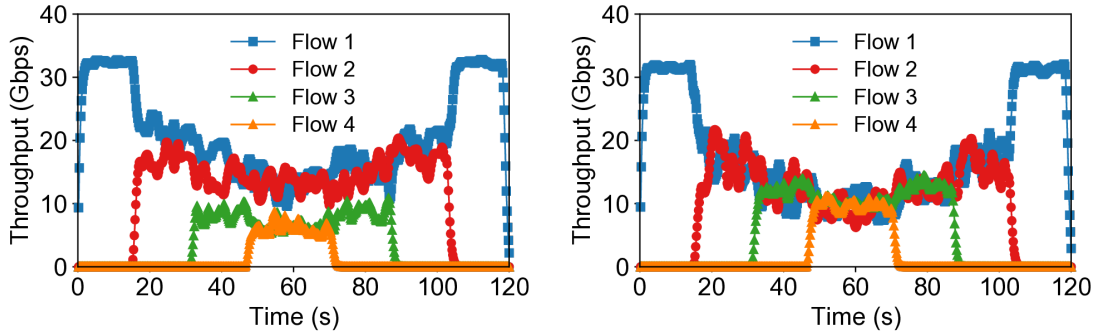


(a) Without HCSFQ. (b) With HCSFQ.

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

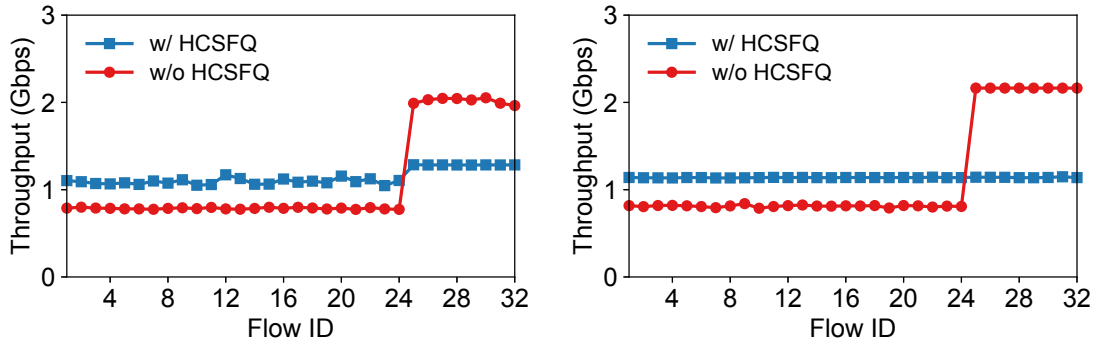
rithms would get more bandwidth. In this experiment, we let Flow 1–24 use TCP Cubic (provided by default in Linux) and Flow 25–32 use TCP BBR. As shown in Figure 3.9(a), without HCSFQ, because TCP BBR is more aggressive than TCP Cubic, the flows with TCP BBR get almost all the bandwidth. On the other hand, HCSFQ is able to provide fair queueing, regardless of the TCP algorithms they use. We have also tried TCP Reno, which performs similar to TCP Cubic.

Different RTTs. In this experiment, we increase the RTT of Flow 25–32 by 0.4 ms using Linux Traffic Control (Linux tc). The default RTT measured by ping in the testbed, i.e., the RTT of Flow 1–24, is 0.3 ms (mostly host overhead). The TCP throughput is inverse proportional to RTT [83]. In our case, the flows with 0.3 ms RTT (Flow 1–24) should have $0.7/0.3 \approx 2\times$ higher bandwidth than the flows with



(a) Without HCSFQ. (b) With HCSFQ.

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



(a) Testbed experiment. (b) Packet-level simulation.

Figure 3-12. Evaluation result of mixed TCP and UDP traffic.

0.7 ms RTT (Flow 25–32), which is close to what we see in Figure 3.9(b). On the other hand, HCSFQ is able to provide fair queueing even when the flows have different RTTs.

Convergence. We let four flows from different clients join and leave a link every 16 seconds to evaluate convergence. Figure 3-10 shows the UDP result. Flow 1 and 2 send at 40Gbps (using DPDK [29]), and Flow 3 and 4 send 20Gbps. When HCSFQ is enabled, the four flows quickly converge to a similar rate, even though they have different sending rate. Figure 3-11 shows the TCP result. We set the RTTs of Flow 3 and 4 to 0.7ms using Linux tc, and the RTTs of of Flow 1 and 2 are around 0.3ms by default. With HCSFQ, the four flows quickly converge to a similar rate, regardless of

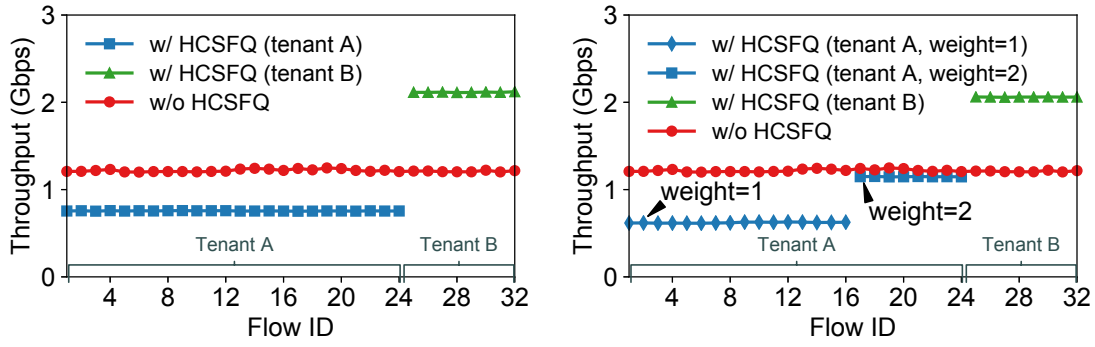
different RTTs.

Mixed UDP and TCP traffic. We evaluate HCSFQ under a mixed workload with both UDP and TCP traffic, and consider the impact of ill-behaved UDP flows on TCP flows. In the experiment, Flow 1–24 are TCP flows, and Flow 25–32 are UDP flows that send at 3.2Gbps. As shown in Figure 3.12(a), without HCSFQ, because UDP flows are not affected by TCP congestion control, Flow 25–32 get 84% higher throughput than their fair share. In comparison, HCSFQ is able to allocate bandwidth fairly between all flows.

Gap between prototype implementation and theoretical algorithm. Although the above experiment demonstrates the effectiveness of HCSFQ on protecting TCP flows from aggressive UDP flows, there is still a small gap from the theoretical upper bound. Figure 3.12(b) shows the simulation result on the same setup using a packet-level simulator Netbench [84]. In the simulation, the TCP and UDP flows get almost identical throughput with HCSFQ. The reason for the gap between Figure 3.12(a) and Figure 3.12(b) is that to realize HCSFQ on a real switch, we make several approximations described in §3.5. These approximations cause extra jitters for TCP flows, and UDP flows occupy the spare bandwidth caused by the jitters and obtain higher throughput. We believe as programmable switches get more capable, these approximations can be removed to enable more accurate implementation of HCSFQ in the future.

3.6.2 Hierarchical Fair Queueing Experiments

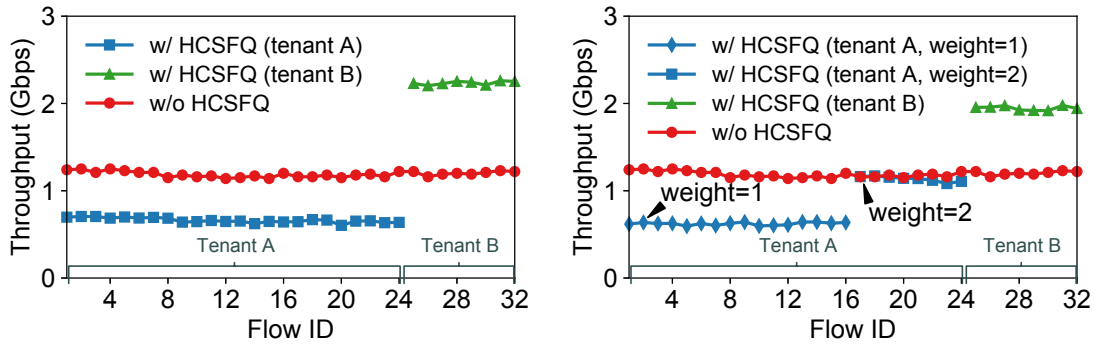
We now evaluate the capability of HCSFQ to provide hierarchical fair queueing. We show that two-layer HCSFQ can provide tenant-level and flow-level isolation for multi-tenant datacenters. Similar to the previous experiments, we use 4 servers to send a total of 32 flows to a receiver. To evaluate hierarchical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



(a) Equal weights.

(b) Different weights.

Figure 3-13. Testbed experiments of hierarchical fair queueing for UDP. Two tenants should have the same *total* throughput.



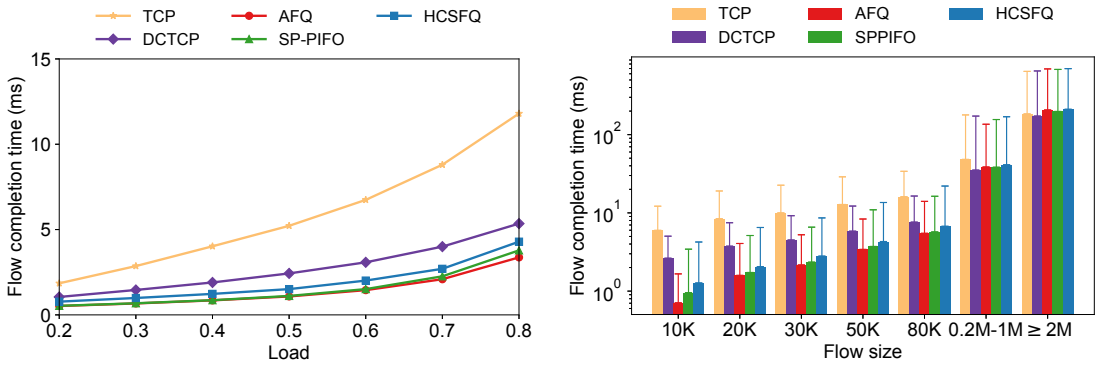
(a) Equal weights.

(b) Different weights.

Figure 3-14. Testbed experiments of hierarchical fair queueing for TCP. Two tenants should have the same *total* throughput.

UDP. We set the sending rates of all 32 UDP flows to 8 Gbps. As shown in Figure 3.13(a), without HCSFQ, the flows have similar throughput. Because tenant A has three times as many flows as tenant B, the total throughput of A is three times as that of B. With HCSFQ, two tenants get the same total throughput. Because A has more flows, each flow in A has lower throughput than that in B.

To evaluate weighted hierarchical fair queueing, we assign different weights to tenant A's flows. We let Flow 1–8 have weight 2 and Flow 9–24 have weight 1. We assign the same weight to tenant A and B. As shown in Figure 3.13(b), the result without HCSFQ is the same as it in Figure 3.13(a). All flows receive the same bandwidth, regardless of tenants and weights. With HCSFQ, because the two tenants



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

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

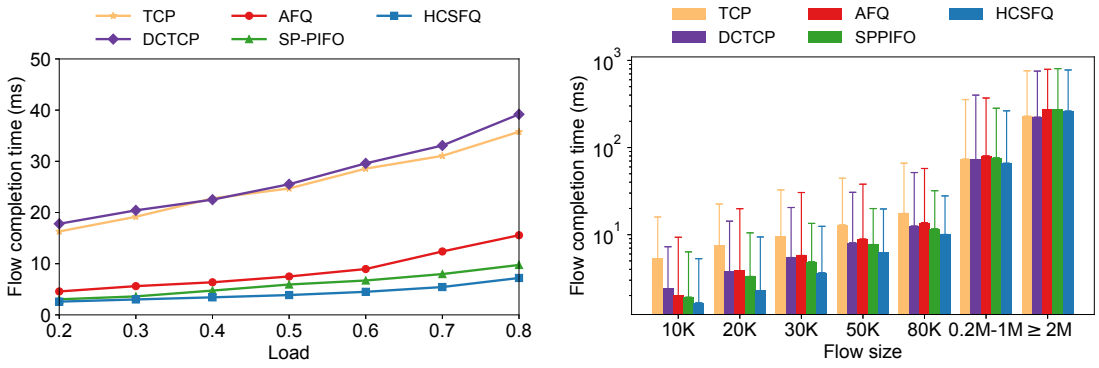
Figure 3-15. Simulation result under the web search workload.

have the same weight, the bandwidth allocation to the two tenants stays the same. In tenant A, a flow with weight 2 has double throughput as a flow with weight 1. In tenant B, all flows have the same weight, and thus they have the same throughput.

TCP. TCP congestion control does not recognize tenants. Figure 3.14(a) shows the throughput of 32 TCP flows. Similar to the UDP experiment, without HCSFQ, every flow receives the same amount of bandwidth, and tenant A has higher total throughput. With HCSFQ, the bandwidth is allocated equally to the two tenants, and each flow in A has lower throughput than each flow in B. We also assign weights to the TCP flows as the UDP experiment, and the result is in Figure 3.14(b). Similarly, with HCSFQ, Flow 1–8 in tenant A have lower throughput than Flow 9–24, because Flow 1–8 lower higher weight. The flows in tenant B have the same throughput because we do not change their weights.

3.6.3 Large-Scale Simulation

We use simulations to evaluate HCSFQ in a large-scale datacenter environment. The simulations are conducted with a packet-level simulator Netbench [84]. Following the setting in SP-PIFO [65], we use a leaf-spine topology with 144 servers, 9 leaf switches and 4 spine switches, and set the access and leaf-spine links to 1Gbps and



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

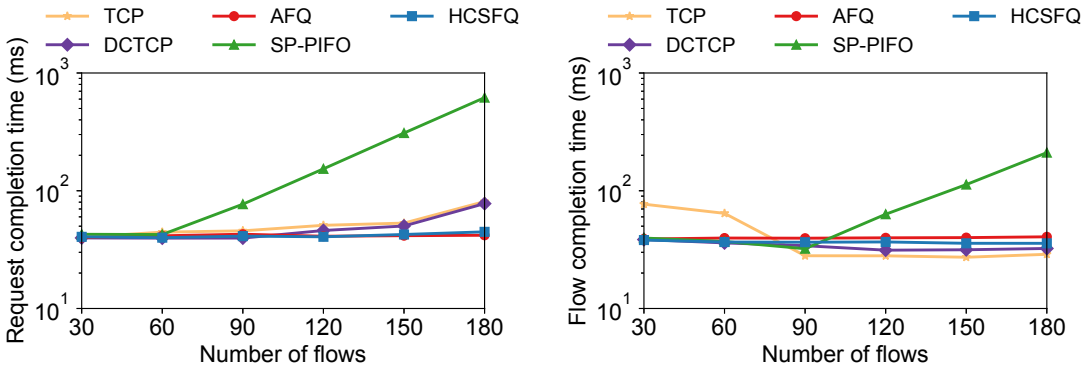
(b) Flow completion time (avg. and 99th) breakdown for 70% load.

Figure 3-16. Simulation result under the web search workload with injected UDP traffic.

4Gbps, respectively. We compare HCSFQ with TCP, DCTCP, and two state-of-the-art approaches AFQ (32 queues) [64] and SP-PIFO (32 queues) [65]. As in [64, 65], we enable ECN marking and use DCTCP as the transport layer for HCSFQ, AFQ and SP-PIFO.

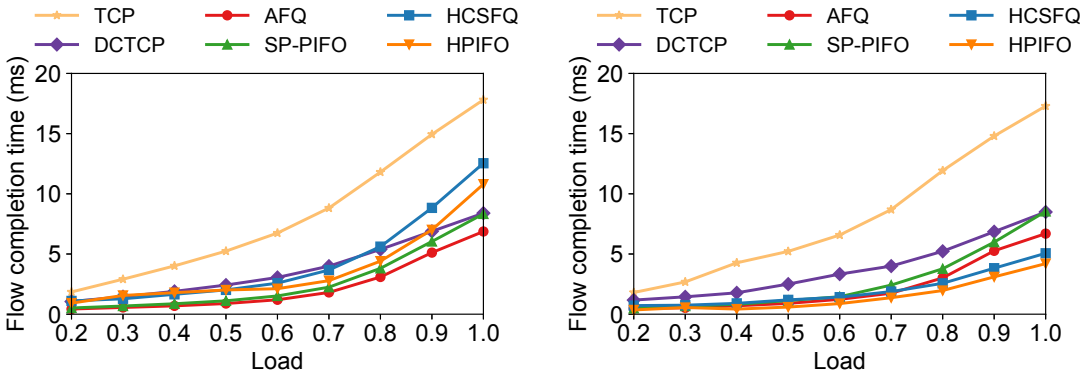
Web search workload. We generate traffic based on the web search workload [85]. Figure 3.15(a) shows the flow completion time (FCT) for small flows less than 100KB, and Figure 3.15(b) shows the flow completion time breakdown when the network is at 70% utilization. HCSFQ achieves up to 60% lower FCT than vanilla TCP and DCTCP. AFQ and SP-PIFO are 15% better than HCSFQ on FCT because HCSFQ enforces fairness by packet dropping and cannot provide guarantee for sensitive packets which can be a drawback for datacenter workload. However, the gap is small and does not grow as the traffic load gets larger. The result demonstrates that HCSFQ is compatible with DCTCP, and can provide significant improvement under a representative datacenter topology and workload as the smaller flows can finish faster with a fair share rate.

Web search workload with injected UDP traffic. To evaluate performance isolation, we inject additional ill-behaved UDP flows to the web server workload. The UDP flows are evenly distributed in the topology and occupy about half of the



(a) Total request. (b) Individual flows.

Figure 3-17. Simulation result under the incast scenario.



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

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

Figure 3-18. Simulation result under the web search workload with two tenants. Tenant 1 sends five times as many flows as tenant 2, and should have higher FCT than tenant 2.

total bandwidth of the network. Figure 3-16 shows that TCP and DCTCP perform significantly worse than others, because they do not have performance isolation between TCP and UDP flows. HCSFQ performs better than AFQ and SP-PIFO, because AFQ and SP-PIFO map different flows to a small number of queues and aggressive UDP traffic overloads the queues shared by multiple TCP and UDP flows, while HCSFQ drops excessive UDP packets before they enter the queues.

Incast. This experiment evaluates HCSFQ in an incase scenario where a receiver requests for a 4.5MB file distributed over N (=30–180) sender nodes. We follow the common practice to use a small RTO_{min} (200μs) for all schemes [86]. As shown in

Figure 3.17(a), when the number of flows grows, HCSFQ achieves a lower request completion time compared with SP-PIFO, TCP and DCTCP, and is close to AFQ. SP-PIFO does not handle the incast traffic pattern well, because there are many packets arriving at the same time with similar ranks saturating some queues and getting dropped. Figure 3.17(b) shows that HCSFQ achieves low average completion times for individual flows as the number of flows changes.

Web search workload with two tenants. This experiment evaluates hierarchical fair queueing with two tenants. Tenant 1 sends five times as many flows as tenant 2, and the flow size and arriving time follow the web search workload [85]. As shown in Figure 3-18, HCSFQ can provide tenant-level fairness, so that since tenant 1 has more flows, the average flow completion time of tenant 1 is higher than that of tenant 2. We also implement a hierarchical version of PIFO (HPIFO) as an upper bound for comparison. Note that although HPIFO delivers the best result, it needs to maintain three queues (one in the first layer and two in the second layer) for two tenants. It cannot be implemented on today's switches and it is hard to support many tenants due to the need of hierarchical queues. Other approaches do not distinguish between tenants, and the average flow completion times of the two tenants are similar.

Scalability with many tenants. We show the scalability of HCSFQ on supporting many tenants and flows. When there are many tenants and flows, the share of each tenant/flow is small and the bias from rate estimation and rate update in each step will accumulate. In this experiment, we examine 50 tenants. Half of the tenants (tenant 1-25) have one VM in each server, and the other half (tenant 26-50) have two VMs in each server. Each VM has a long-lasting TCP flow with another VM of the same tenant in another rack. We set the bandwidth of access links and leaf-spine links to 10Gbps and 40Gbps respectively in order to accommodate more tenants and flows than previous experiments. Figure 3-19 shows that TCP, DCTCP, AFQ and SP-PIFO do not provide tenant-level fairness, and the tenants with more flows have

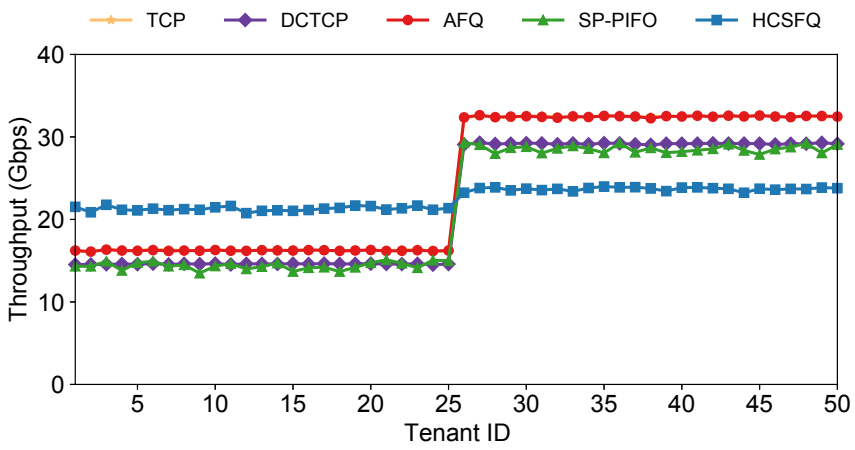


Figure 3-19. Throughput of different tenants. Each tenant of Tenants 1-25 has one VM in each server, while each tenant of Tenants 26-50 has two VM in each server. Each tenant is sending pairwise TCP traffic between its VMs.

higher total throughput. In comparison, HCSFQ provides fair bandwidth allocation between tenants, regardless of the number of flows each tenant has.

3.7 Related Work

Fair queueing. There is a long history of work on fair queueing. The original proposal from Nagle [54] introduces the idea of using separate FIFO queues for flows to achieve fair bandwidth allocation. The bit-by-bit round robin (BR) algorithm [55, 56] computes a bid number to estimate the departure time for each packet, and transmits the packet with the lowest bid number with a priority queue. To avoid expensive priority queues, several algorithms, such as SFQ [57] and DRR [58], propose to map flows to a small number of FIFO queues, which do not work well when the number of flows are far larger than the number of queues. Another approach is probabilistic packet dropping, which maintains per-flow state to estimate drop probability, such as FRED [59], RED-PD [60] and AFD [61]. CSFQ [66] is distinct from these algorithms in that it does not require per-flow state, per-flow queues or an expensive priority queue. Hierarchical fair queueing adds a hierarchy to fair queueing, which require not only per-flow state, but also a hierarchy of queues [62, 63, 68]. HCSFQ eliminates both requirements, making

hierarchical fair queueing feasible to be implemented in high-speed hardware switches.

Network isolation in multi-tenant cloud. Prior work has proposed techniques to provide performance guarantees and share bandwidth between multiple tenants [67, 69–81, 87, 88]. However, existing works either can only enforce hierarchical fairness at end hosts, or can not be efficiently implemented in today's hardware. For example, BwE [88] is a WAN bandwidth allocation mechanism which enforces hierarchical fair allocation at end hosts. FairCloud [67] proposes to apply CSFQ for network isolation in datacenters, but it does not have a hardware implementation for CSFQ and does not support hierarchical fair queueing. HCSFQ is to the best of our knowledge, the *first* solution to provide hierarchical fair queueing on commodity switches with small switch memory footprint and a single FIFO queue.

Programmable switches. Programmable switches have triggered many innovations in recent years [17, 39, 41–53, 82, 89–94]. Programmable packet scheduling is the most relevant to HCSFQ. UPS [95] shows that Least Slack Time First (LSTF) provides a good approximation for many scheduling algorithms in practice. PIFO [63] provides a hardware design to realize the abstraction of a push-in first-out (PIFO) queue. It relies on a tree of PIFO queues to implement hierarchical fair queueing. AFQ [64] approximates fair queueing by using a few queues to emulate many queues. It stores per-flow counters in a count-min sketch, and does not support hierarchical fair queueing. SP-PIFO [65] uses several strict priority queues to emulate a PIFO queue, which can support fair queueing, not hierarchical fair queueing. Compared to them, we show how to leverage programmable switches to support fair queueing without per-flow state based on CSFQ, and present a new algorithm HCSFQ to support hierarchical fair queueing.

3.8 Conclusion

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

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

Chapter 4

AIFO: Programmable packet scheduling with a single queue.

4.1 Introduction

Packet scheduling is a central research topic in computer networking. Over the past several decades, a great many packet scheduling algorithms have been designed to provide different properties and optimize diverse objectives [55, 85, 96–98]. Unfortunately, most of these algorithms, despite many novel ideas among them, never have found their way to impact the real world. This is largely due to the high cost to design and deploy switch ASICs to implement them, since packet scheduling algorithms must run in the data plane at line rate in order to process every single packet.

Programmable packet scheduling is a holy grail for packet scheduling as it enables scheduling algorithms to be programmed into a switch without changing the hardware design. With programmable packet scheduling, one is able to develop or simply download a packet scheduling algorithm that best matches the operational goals of the network. This enables network operators to highly customize packet scheduling algorithms based on their needs. Particularly, it simplifies the testing and deployment of new scheduling algorithms, and it enables algorithms that are targeted at small niche markets and thus cannot justify the high cost of developing new switch ASICs to be used and deployed.

A Push-In First-Out (PIFO) queue is a popular abstraction for programmable packet scheduling [63, 65]. PIFO associates a rank with each packet and maintains a sorted queue to buffer packets. Newly arrived packets are inserted into the queue based on their ranks, and packets are dequeued from the head. Different packet scheduling algorithms can be implemented on top of PIFO by changing the rank computation function. Prior works have shown that PIFO can support a wide range of popular scheduling algorithms, such as Shortest Remaining Processing Time (SRPT) [96] for minimizing flow completion times (FCTs) and Start-Time Fair Queueing (STFQ) [99] for weighted fairness.

PIFO, while elegant in theory, is challenging to implement in practice. A recent work [63] proposes a hardware design to support PIFO at a clock frequency of 1 GHz on shared-memory switches. The major design complexity lies in supporting a sorted queue at 1 GHz. Yet, there is a gap from the design to a real switch ASIC implementation, and the design has scalability limitations—it can only support a few thousand flows. SP-PIFO [65] is an approximation of PIFO that can run on existing hardware. The basic idea is to map the possibly large number of ranks into a small set of priorities, and then simply schedule the small number of queues based on their priorities. This solution, however, requires multiple precious strict-priority queues.

In this chapter, we present Admission-In First-Out (AIFO) queues, a new solution for programmable packet scheduling that uses only a single first-in first-out (FIFO) queue. FIFO (drop-tail) queues are one of the simplest queues that can run at line rate and are available in almost all switches. Thus, AIFO is amenable to be implemented in high-speed switches with line rate, and we show not only a concrete design, but also a real implementation of AIFO on existing hardware (Intel Tofino), with minimal requirements on hardware primitives—a single FIFO queue, as opposed to multiple strict-priority queues.

AIFO is motivated by the confluence of two recent trends in datacenter networking:

shallow buffers in the switches [100] and fast-converging congestion control protocols implemented in end hosts [101]. Together, they significantly reduce the queueing latency inside the network, which is especially important for datacenter environments where low latency is critical for real-time online services with strict Service Level Objectives (SLOs) [100]. Given these trends, we observe that the decisive factor in modern datacenters is often *which* packets are enqueued or dropped by the switch, not the *ordering* in which they leave the switch. For example, dropping packets of an elephant flow when it competes with two mice flows is more important to the flow completion times of the mice flows than the ordering that their packets are dequeued, especially when the queue length is kept small such that only *a few* packets occupy it at any moment.

Based on this insight, the major technical challenge we tackle in this chapter is *finding the right set of packets to admit into the queue.* Ideally, AIFO should admit the same set of packets as PIFO to closely approximate it. AIFO addresses this challenge by maintaining a sliding window to track the ranks of recent packets in the window and computing the relative rank of an arriving packet in order to decide whether to admit or proactively drop it even when the queue may still have room! Unlike traditional active queue management (AQM) solutions, AIFO drops packets based on their relative ranks instead of using threshold comparisons against average queue length [102–104] or delay estimations [105]. Theoretically, we prove that AIFO provides performance close to that of PIFO. We complement it with a concrete data plane design and implementation to show how to efficiently realize AIFO on Intel Tofino.

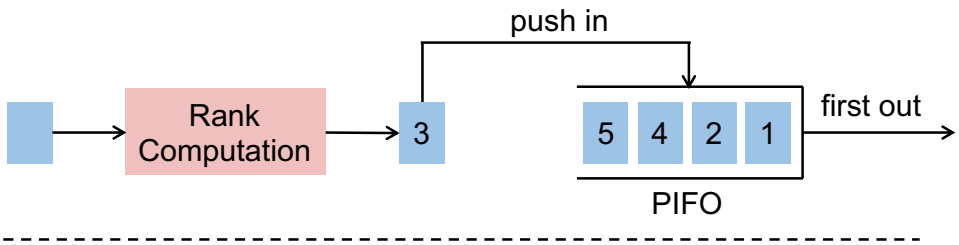
AIFO explores an interesting design question: what are the minimal hardware requirements for programmable packet scheduling? AIFO is an extreme point in the design space—it only requires a single FIFO queue. This is not only theoretically interesting, but also has important practical implications. Our conversations with

industry collaborators, including a large-scale search engine and a large-scale e-commerce service, indicate that physical queues are critical resources, and are reserved to ensure strong physical isolation and differentiation between applications of multiple tenants; modern datacenters are already short of physical queues available in switches. Unlike SP-PIFO which requires multiple physical queues for packet scheduling, AIFO enables operators to continue using physical queues for strong physical isolation and differentiation between tenants, and additionally use AIFO to program the packet scheduling algorithm for intra-tenant traffic (e.g., SRPT to minimize the flow completion time).

As an unexpected positive byproduct, AIFO naturally supports *starvation prevention* by design, a necessary feature of pFabric [85] that schedules the packets of the same flow in FIFO to prevent packet reordering. While PIFO supports a wide variety of scheduling algorithms, it cannot support starvation prevention needed by pFabric because the latter packets of a flow would be scheduled first when PIFO is programmed to use SRPT. With the strong demand on minimizing FCTs for low-latency online services, pFabric is arguably the killer application of programmable packet scheduling, as pFabric is considered to be one of the best solutions for minimizing FCTs. AIFO enables us to implement and deploy pFabric on existing hardware.

In summary, we make the following contributions.

- We propose AIFO, a new approach to programmable packet scheduling that uses only a single queue.
- We design an algorithm based on sliding windows and efficient relative rank computation to realize AIFO in the switch data plane. Theoretically, we prove that AIFO provides bounded performance to PIFO.
- We implement a AIFO prototype on a Intel Tofino switch. We use a combination of simulations and testbed experiments to evaluate AIFO under a range of real



Rank Computation for Shortest Remaining Processing Time (SRPT)

```
p.rank= p.remaining_processing_time
```

Rank Computation for Start-Time Fair Queueing (STFQ)

```
if p belongs to a new flow:
  p.start = virtual_time
else:
  p.start = max(virtual_time, finish_time[p.flow_id])
finish_time[p.flow_id] = p.start + p.length/weight[p.flow_id]
p.rank = p.start
```

Figure 4-1. Background on programmable packet scheduling with PIFO.

workloads and scheduling algorithms, demonstrating AIFO closely approximates PIFO.

The code of AIFO is open-source and is publicly available at https://github.com/netx-repo/AIFO.

4.2 Background and Motivation

In this section, we first provide background information on programmable packet scheduling, and then use an example to motivate the key ideas of AIFO.

4.2.1 Programmable Packet Scheduling

Programmable packet scheduling enables the packet scheduling algorithm in a switch to be changed without the need to change the switch ASIC. PIFO [63] is a proposal for programmable packet scheduling. It contains two components: a PIFO queue and

a rank computation component. Each packet is associated with a *rank*. The PIFO queue is a priority queue that sorts packets based on their ranks. Packets are inserted into the queue based on their ranks, and are dequeued from the head (i.e., the smallest rank).

Programmability lies in the rank computation component. Programming a packet scheduling algorithm in the context of PIFO refers to programming how a rank for each packet is computed. One simple example is to program SRPT [96] for minimizing FCTs, as shown in Figure 4-1. In this example, the rank of a packet is simply the remaining processing time of the flow (or simply the remaining bytes of the flow). Note that SRPT requires end hosts to put the remaining processing time in an appropriate field in the packet header [85, 96], which is orthogonal to packet scheduling in the switch. Given such rank computation, the PIFO queue would schedule the packet with the shortest remaining processing time first, i.e., realizing SRPT.

A more complicated example is to program STFQ [99] for weighted fairness, which is also shown in Figure 4-1. In this example, the rank of a packet is the virtual start time of the packet in STFQ. The virtual start time is computed as the maximum of the virtual time and the virtual finish time of the previous packet of the same flow. The virtual time maintains the virtual start time of the last dequeued packet across all flows. The virtual finish time of a packet is the virtual start time of the packet plus the length of the packet divided by the flow weight. Given such rank computation, the PIFO queue would schedule the packet with the smallest virtual start time first, i.e., realizing STFQ.

Beyond these two example, it has been shown that PIFO can support a wide range of packet scheduling algorithms, such as Least-Slack Time-First [97], Service-Curve Earliest Deadline First (SC-EDF) [98], etc.

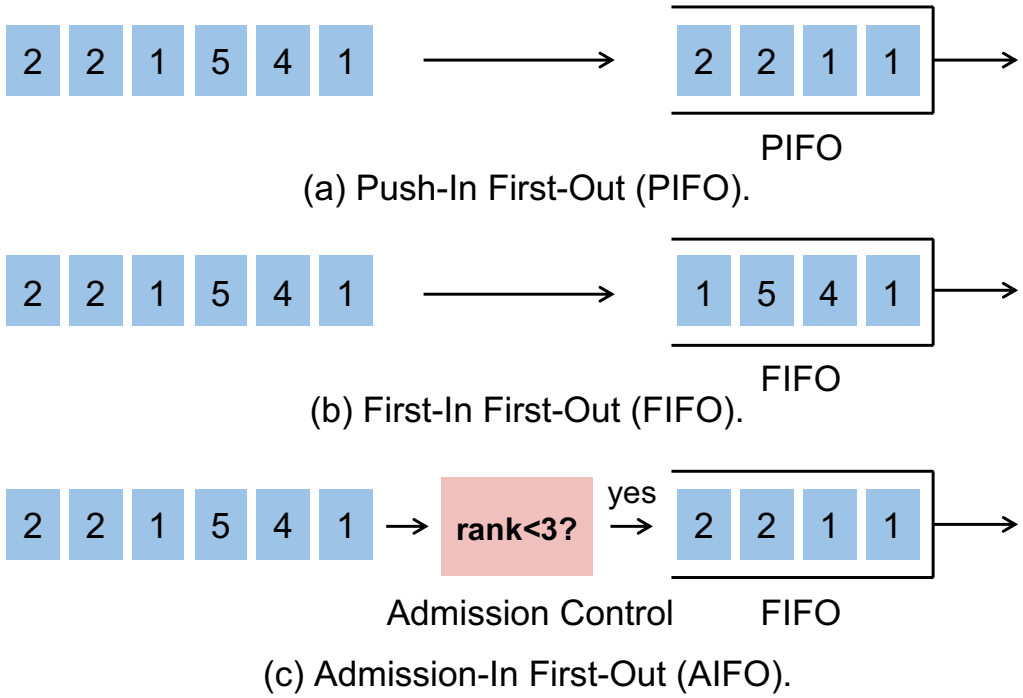


Figure 4-2. Motivating example of AIFO.

4.2.2 Motivating Example

While PIFO is an appealing solution for programmable packet scheduling, it is challenging to implement in hardware, especially in switch ASICs. The rank computation component is relatively easy. It can be implemented as a packet transaction [106] in the data plane of existing programmable switches. The major challenge is to implement the PIFO queue. Existing switches do not support a sorted queue in the data plane. There is a proposal on how to support a sorted queue in the data plane at 1 GHz [63]. But the proposal only provides a design, not a real implementation, and the design is not scalable as it can only support a few thousand flows. SP-PIFO [65] provides an approximation of a PIFO queue using multiple strict-priority queues. But strict-priority queues are precious hardware resources as commodity switches have a limited number of strict-priority queues and the operators would like to use them to ensure strong physical isolation between multiple tenants. In this chapter, we aim to design a solution that has the minimal hardware requirements for programmable

packet scheduling.

Example. To find such a solution, let us get down to the fundamentals to analyze the problem. We consider the arrival traffic and departure traffic of a queue. When the packet arrival rate is no higher than the link speed (i.e., the upper bound of the departure rate), the entire traffic is admissible and there is no persistent queue buildup. It does not matter whether the queue is PIFO, FIFO, or anything else. The distinction happens when the arrival rate is higher than the link speed, which can be either due to a microburst or a longer-term congestion. In this case, some of the packets are not admissible and the queuing discipline matters.

We examine the examples in Figure 4-2. The example is simplified to provide the intuition of our approach. In the example, there is a burst of six packets arriving at the switch. The queue has four slots and is empty in the beginning. For the first four packets, PIFO would enqueue them one by one and the sorted queue becomes [1, 1, 4, 5]. Then when the fifth packet with rank 2 arrives, PIFO would insert the packet into the queue and the last packet in the queue is dropped due to overflow. The queue becomes [1, 1, 2, 4]. Finally, when the sixth packet arrives, the last packet in the queue is dropped again and the queue is [1, 1, 2 ,2] in the end.

In terms of FIFO, it enqueues the packets one by one. After four packets, the queue is full, and the fifth and sixth packets cannot be enqueued. The queue is [1, 4, 5, 1] in the end. PIFO and FIFO behave very differently.

However, if there is an oracle that knows the precise arrival pattern of the packets in advance, then the switch can perform admission control before the packets are enqueued. Specifically for the example in Figure 4-2, the admission control can use a threshold of 3. If the rank of a packet is no bigger than 3, then the packet can be enqueued; otherwise the packet is dropped. With this, the second and third packets would be dropped and the queue is [1, 1, 2, 2] in the end. FIFO with such admission

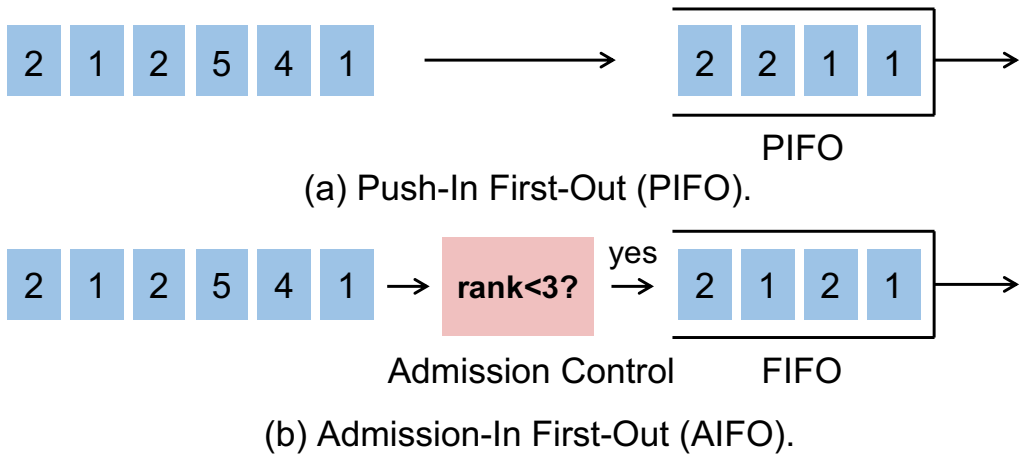


Figure 4-3. An example that PIFO and AIFO dequeue the same set of packets ({1, 1, 2, 2}), but the dequeueing orderings are different ([1, 1, 2, 2] vs. [1, 2, 1, 2]).

control behaves the *same* as PIFO in this example.

4.3 Design Goal

Based on the insights from the motivating example, we can transform the packet scheduling problem into an admission control problem, and PIFO can be approximated by FIFO with admission control.

We term this approach AIFO. Our goal is to minimize the gap between the ideal case (i.e., PIFO) and the approximation (i.e., AIFO). The gap can be measured quantitively with the following metric: the difference between the packets dequeued by PIFO and those dequeued by AIFO. Formally, let the set of packets dequeued (up to time t) by PIFO and AIFO to be $\mathbb{P}(t)$ and $\mathbb{A}(t)$, respectively. Then we use

$$\Delta(t) = \frac{|\mathbb{P}(t) \setminus \mathbb{A}(t)| + |\mathbb{A}(t) \setminus \mathbb{P}(t)|}{|\mathbb{P}(t)| + |\mathbb{A}(t)|} \tag{4.1}$$

to measure the gap between PIFO and AIFO. Here, $|\mathbb{P}(t) \setminus \mathbb{A}(t)|$ is the cardinality of the set difference between $\mathbb{P}(t)$ and $\mathbb{A}(t)$, and $|\mathbb{A}(t) \setminus \mathbb{P}(t)|$ is that between $\mathbb{A}(t)$ and $\mathbb{P}(t)$. $|\mathbb{P}(t)|$ and $|\mathbb{A}(t)|$ are the cardinalities of sets $\mathbb{P}(t)$ and $\mathbb{A}(t)$, respectively.

We have $\Delta(t) \in [0, 1]$, and a large value of Δ indicates a large gap between AIFO

and PIFO. When AIFO and PIFO dequeue the same set of packets (i.e., no gap), $\Delta = 0$; when AIFO and PIFO dequeue completely different packets, $\Delta = 1$. We theoretically prove that the difference between AIFO and PIFO is negligible when the system is stationary (§4.4), and empirically demonstrate that AIFO provides close performance as PIFO with a range of real workloads (§4.5).

Packet ordering. Another possible metric would be to not only count the number of different packets that are dequeued, but also account for the difference in the dequeuing ordering. Figure 4-3 provides an example to illustrate this metric. The example is similar to the one in Figure 4-2, and the only difference is that the third and the fourth arrival packets are swapped in Figure 4-3. With this arrival sequence of packets, AIFO still admits the same set of packets as PIFO, which are {1, 1, 2, 2}. However, the orderings that the packets are dequeued are different. AIFO uses the ordering [1, 2, 1, 2], which PIFO uses the ordering [1, 1, 2, 2].

We argue that this metric is less important than the first metric, and sometimes is even undesirable to optimize for. First, there are two important trends for data-center networking: (*i*) the trend towards shallow buffers for low latency in modern datacenters [100]; and (*ii*) the trend towards tight control loops at end hosts [101]. The confluence of these two trends ensures that the switch queues would not buffer many packets, making the difference on the dequeueing ordering between PIFO and AIFO minimal. As we are essentially emulating PIFO with a FIFO queue, we want to keep the buffer shallow so that the packets can have a short waiting time in the queue. Empirically, we show in Section 4.5 that such a difference would not impact the flow-level metrics like flow completion time (FCT) much and AIFO behaves almost the same as PIFO.

Second, strictly following PIFO causes packet reorderings, which is undesirable. SRPT achieves near optimality on minimizing FCTs [96]; it schedules flows based on the remaining flow size, so that small flows are scheduled before big flows to minimize

FCTs. Packet reorderings happen when PIFO is programmed to implement SRPT by using the remaining flow size as the rank (like Figure 4-1). This is because for the *same* flow, a latter packet would have a *smaller* remaining flow size than its previous packet, and thus is scheduled first by the switch if both packets are enqueued by the switch.

This is a known issue, and pFabric [85] addresses this issue by adding an extra feature called *starvation prevention* to SRPT. Starvation prevention dequeues the packets of the same flow in the order they arrive, so that the first packet of a flow would be dequeued first if the flow is scheduled and the first packet would not be *starved*. Between flows, pFabric uses SRPT to select which flow to schedule first. Given the strong demand on low-latency for datacenter networks, pFabric is arguably the killer application of programmable packet scheduling. Yet, it cannot be supported by PIFO [63]. Unexpectedly, a positive byproduct of AIFO is that it naturally supports starvation prevention and eliminates packet reordering by design.

Summary. To summarize, our goal is to design an algorithm that has the minimal hardware requirements (i.e., a single queue) and admits the right set of packets to minimize Δ and maintain shallow buffers. The algorithm should be able to be implemented in the data plane of existing hardware and run at line rate. We want to ensure that the algorithm provides bounded performance to PIFO with respect to Δ.

4.4 AIFO Design

In this section, we first introduce the key ideas of AIFO. Then we describe the AIFO algorithm, and theoretically prove that AIFO closely approximates PIFO. Finally, we describe the switch data plane design to implement AIFO on a programmable switch.

4.4.1 Key Ideas

AIFO only uses a single FIFO queue, instead of a PIFO queue or multiple strict-priority FIFO queues. It adds admission control in front of the FIFO queue to decide whether to admit or drop an arriving packet. Admitted packets are buffered and sent by the queue in FIFO order, and no extra scheduling is needed. The admission control is designed to minimize Δ described in Section 4.3, in order to minimize the gap between AIFO and PIFO.

AIFO achieves a close approximation of PIFO with *two-dimensional* admission control by simultaneously considering both *time* and *space* dimensions. Its *temporal* component considers the time dimension by changing the threshold over time based on the fluctuation of the arrival rate; its *spatial* component considers the space dimension by deciding the threshold based on the ranks of the packets at each time. These two together ensure AIFO admits a similar set of packets as PIFO. At a high level, the two components work as follows.

- **Temporal component.** The threshold of admission control is dynamic, instead of fixed. It is updated based on the real-time discrepancy between arrival rate and departure rate. When the arrival rate significantly exceeds the departure rate, the threshold becomes more aggressive. It ensures the rate of admitted packets roughly matches the departure rate.

- **Spatial component.** The admission control treats packets differently based on their ranks, instead of using a naive rank-agnostic criteria (e.g., randomly dropping 10% packets). It prefers to drop high-rank packets over low-rank packets, as low-rank packets are expected to be scheduled first. The threshold is decided based on the arrival rate distribution of different ranks. It ensures the admitted packets have similar ranks as those admitted by PIFO.

We note that the basic idea of dynamic, proportional adaption is widely used, and in particular for networking, it has been instantiated in various forms in congestion control [101, 107, 108]. For examples, delay-based congestion control algorithms like TIMELY and Swift [101, 107] adapt the TCP window size dynamically based on the end-to-end delay, and ECN-based algorithms like DCTCP [108] adapt the window size in proportional to the number of packets with the ECN flag. These instantiations all put the control in the end hosts. In comparison, AIFO places the dynamic, proportional adaption in the network, and it serves a different purpose, i.e., programmable packet scheduling. This context brings stringent requirements to the algorithm design: the algorithm should not only achieve optimality, but also be carefully designed to be implemented at line rate.

For readers familiar with the packet scheduling literature, AIFO can be considered as an AQM solution. Traditional AQM solutions consider a specific objective, and drop packets using threshold comparisons against average queue length [102–104] or delay estimations [105]. In comparison, AIFO is designed to be a general solution that can be programmed to support different objectives, and it drops packets with a combination of threshold comparisons (i.e., the temporal component) and relative packet rank estimations (i.e., the spatial component).

4.4.2 Algorithm

We design AIFO based on these key ideas. Algorithm 5 shows the pseudocode. At the ingress (line 1-8), AIFO uses admission control (line 2-5) to decide whether to enqueue (line 6) or drop a packet (line 8). The threshold is dynamically determined by queue length (c) and queue size (C), and we use quantile estimation ($W.quantile(pkt)$) to estimate the relative rank of current packet. The queue is a FIFO queue which enqueues the packet to the end of the queue. At the egress (line 9-12), when the queue is not empty, AIFO dequeues a packet from the head of the queue, and sends the

Algorithm 5 AIFO

```
1: function INGRESS(pkt)
       // Admission Control
2:     Update sliding window W with pkt
3:     c ← Queue.length
4:     C ← Queue.size
5:     if c ≤ k · C ∥ W.quantile(pkt) ≤ 1/(1−k) (C−c)/C then
           // Admit packet
6:         Queue.enqueue(pkt)
7:     else
           // Drop packet
8:         Drop pkt
9: function EGRESS
10:    if Queue is not empty then
11:        pkt ← Queue.deque()
12:        Send pkt
```

packet out.

Next, we explain the admission control part in detail. For the temporal component, it uses the difference between the current queue length (denoted by c) and the target queue size (denoted by C) to capture the discrepancy between arrival rate and departure rate. The threshold of admission control is more aggressive when the current queue length approaches the target queue size, i.e., when $\frac{C-c}{C}$ is small. We allocate a headroom to tolerate small bursts with a parameter k. When the queue length is within the headroom (i.e., $c \leq k \cdot C$), all packets are admitted. Accordingly, the difference between the queue length and the queue size is also scaled by $\frac{1}{1-k}$ to account for the headroom. We separate $c \leq k \cdot C$ and $W.quantile(pkt) \leq \frac{1}{1-k}\frac{C-c}{C}$ into two conditions at line 5 for clarity. Mathematically, the first condition $c \leq k \cdot C$ is redundant. This is because when $c \leq k \cdot C$, then $\frac{1}{1-k}\frac{C-c}{C} \geq 1 \geq W.quantile(pkt)$ where $W.quantile(pkt)$ estimates the quantile of pkt, and the packet is always admitted.

It is important to note that C is not necessarily the physical size of the FIFO queue. The physical size of a queue in a commodity switch varies in a large range from tens of packets to hundreds or even thousands of packets, depending on the switch

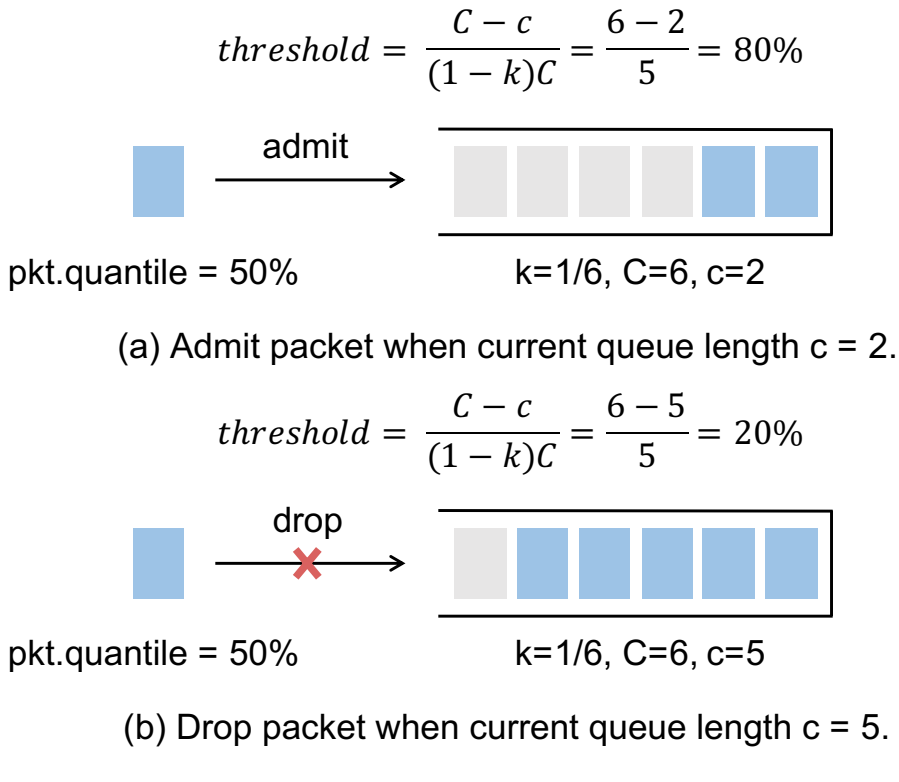


Figure 4-4. Examples of admission control in AIFO.

ASIC. Despite this capability, production networks tend to use shallow buffers and limit the queue size in deployment for low latency. As such, C can be configured to a smaller number than the physical queue size, and thus we term it as the target (not physical) queue size in the algorithm description.

For the spatial component, AIFO maintains a sliding window of recently received packets and uses the quantile of the rank of the arrival packet ($W.quantile(pkt)$) as the criteria. When the quantile is no bigger than $\frac{1}{1-k}\frac{C-c}{C}$, the packet is admitted; otherwise, the packet is dropped. The intuition is that after accounting for the headroom with $\frac{1}{1-k}$, $\frac{1}{1-k}\frac{C-c}{C}$ captures the amount of *remaining* queue space, in terms of the *percentage* of the target queue length. Only a subset of the following packets that can fit the remaining queue space can be admitted. We find a rank r^* of which the quantile is equal to the percentage representation of the remaining queue space. We only admit the packets with ranks no bigger than r^* to ensure that the admitted

subset of packets are the low-rank packets that should be admitted and can just fit the remaining queue space. We maintain a sliding window to estimate the quantile of an arrival packet based on the past packets.

The benefit of the two-dimensional approach is that the inaccuracy of one component can be compensated by the other component. If the quantile estimation of the sliding window (the spatial component) is a bit off, i.e., admitting extra packets, then the queue length c would increase, making the quantile threshold $\frac{1}{1-k}\frac{C-c}{C}$ (the temporal component) more strict. And this corrects the spatial component to use a smaller rank threshold.

We provide two examples to illustrate different cases in the admission control. The examples are shown in Figure 4-4. The target queue length C is 6 and the headroom parameter k is 1/6 (i.e., a headroom of $6 \times 1/6 = 1$ packet). Suppose the quantile of the arriving packet's rank is 50%.

- **Case 1: admit packet below quantile threshold.** When the current queue length c is 2, the quantile threshold is $\frac{1}{1-k}\frac{C-c}{C} = 80\%$. This means a packet can be admitted if the quantile of the packet's rank is no bigger than 80%. Since the quantile of the arriving packet's rank is 50%, which is smaller than 80%, the packet is admitted.

- **Case 2: drop packet above quantile threshold.** When the current queue length c is 5, the quantile threshold is $\frac{1}{1-k}\frac{C-c}{C} = 20\%$. This means a packet can be admitted if the quantile of the packet's rank is no bigger than 20%. Since the quantile of the arriving packet's rank is 50%, which is bigger than 20%, the packet is dropped.

4.4.3 Theoretical Guarantee

We provide the theoretical guarantees for AIFO as follows. The proofs of the theorems are in Appendix.

Packet departure rate and queue length. We consider n packet ranks, denoted by $r_1 < r_2 < \cdots < r_n$ (smaller rank value means higher priority). Let λ_i be the arrival rate of packets with rank i. Let $\gamma > 0$ be the queue draining rate. We can prove properties for the departure rate of each rank and the queue length.

Theorem 3. *Assume $\sum_{i=1}^{n} \lambda_i > \gamma$. Let $n^* := \min_i\{\lambda_1 + \cdots + \lambda_i \geq \gamma\}$. When the algorithm reaches the stationary state, it has the following properties on the packets departure rates:*

1. *AIFO and PIFO has the same departure rate for each rank:*
   - *for rank $i < n^*$, its departure rate is λ_i;*
   - *for rank $i = n^*$, its averaged departure rate is $\gamma - \sum_{i<n_*} \lambda_i$;*
   - *for rank $i > n^*$, its departure rate is zero.*
2. *FIFO does not perform as the same as PIFO:*
   - *for rank $i \in \{1, \ldots, n\}$, its departure rate is $\frac{\lambda_i}{\sum_{i=1}^{n} \lambda_i}\gamma$.*

When the algorithm reaches the stationary state, it has the following properties on the queue length:

1. *The queue length for PIFO and FIFO is C.*
2. *The queue length for AIFO is*
   - *$\left(1 - \frac{n^*}{n}(1-k)\right) C$, if $\sum_{j\leq n^*} \lambda_j > \gamma$;*
   - *or bounded between $\left(1 - \frac{n^*+1}{n}(1-k)\right) C$ and $\left(1 - \frac{n^*}{n}(1-k)\right) C$, if $\sum_{j\leq n^*} \lambda_j = \gamma$.*

Proof. We ignore the constraint $c \leq KC$ as it is covered by constraint $W.quantile(pkt) \leq \frac{1}{1-k}\frac{C-c}{C}$, because $\frac{1}{1-k}\frac{C-c}{C} > 1$ when $c \leq KC$.

For FIFO, every packets are treated equally based on first arrival first admission principle, thus their incoming rate is proportional to their sending rate. Let us assume the incoming rates of each type of packet to be $k\lambda_1, k\lambda_2, \ldots, k\lambda_n$. On the other hand, at the stationary state, the total incoming rate of the queue equals to its total outcoming rate γ, i.e., $k\lambda_1 + k\lambda_2 + \cdots + k\lambda_n = \gamma$, which implies $k = \frac{\gamma}{\sum_{j=1}^{n}\lambda_j}$. Thus for packet i, its incoming and outcoming rate is $k\lambda_i = \frac{\gamma\lambda_i}{\sum_{j=1}^{n}\lambda_j}$.

For PIFO, note that the packets are admitted according to its priority, i.e., high priority packets are always admitted ahead of low priority packets. Recall that $\sum_{i=1}^{n}\lambda_i > \gamma$, i.e., the total sending rate is greater than the allowed outcoming rate. Thus when the system reaches its stationary state, there exists a threshold $n^* := \min_i\{\lambda_1 + \cdots + \lambda_i \geq \gamma\}$, such that for $i < n_*$, packet i will always be admitted, i.e., its incoming and outcoming rate is λ_i; for $i = n_*$, packet i will be admitted partly to fill the remaining outcoming ability apart the portion taken by packets $1, 2, \ldots, n_* - 1$, i.e., its incoming and outcoming rate is $\gamma - \sum_{i<n_*}\lambda_*$; for $i > n_*$, packet i can no longer be admitted since the system is already stationary and its priority is below the admitted ones, thus the incoming/outcoming rate is zero.

For AIFO, given a queue length c, the algorithm decides an admission priority threshold

$$n(c) = \frac{1}{1-k}\frac{C-c}{C} \cdot n,$$

where packets $i > n(c)$ cannot be admitted, and packets $i \leq n(c)$ will be admitted. Note that $n(c)$ is decreasing with respect to c. Consider the following two queue length thresholds:

$$c^- = \left(1 - \frac{n^*+1}{n}(1-k)\right)C, \qquad c^* = \left(1 - \frac{n^*}{n}(1-k)\right)C.$$

Clearly $0 < c^- < c^* < C$, and $n(c^-) = n^* + 1$, $n(c^+) = n^*$. Therefore,

- if $c \leq c^-$, all packets $i \leq n^* + 1$ will be admitted. By the choice of n^* we have $\sum_{j \leq n^*+1} \lambda_j > \gamma$, i.e., the total incoming rate is strictly greater than the total outcoming rate, thus the queue length increases;

- if $c > c^*$, all packets $i \geq n^*$ cannot be admitted. By the choice of n^* we have $\sum_{j \leq n^*-1} \lambda_j < \gamma$, i.e., the total incoming rate is strictly less than the total outcoming rate, thus the queue length decreases;

- if $c^- < c \leq c^*$, all packets $i \leq n^*$ will be admitted, and all packets $i > n^*$ will not be admitted. Note that $\sum_{j \leq n^*} \lambda_j \geq \gamma$. We discuss two cases:

  - if $\sum_{j \leq n^*} \lambda_j = \gamma$, then the system reaches its stationary state at the first time when the queue length satisfies $c^- < c \leq c^*$;

  - if $\sum_{j \leq n^*} \lambda_j > \gamma$, then the queue length keeps increases until it becomes larger than c^*, and falls into the previous category hence the length decreases then. In sum the system reaches its stationary state with queue length being c^*. Moreover, due to the negative feedback principle, in the stationary state, the incoming rate/outcoming rate of packet n^* would be $\gamma - \sum_{j<n^*} \lambda_j$.

In sum, at the stationary state, we have the following: for the packets $i < n_*$, the incoming/outcoming rate is λ_i; for packet n_*, the incoming/outcoming rate is $\gamma - \sum_{i<n_*} \lambda_i$; for the packets $i > n_*$, the incoming/outcoming rate is 0. Moreover, we can also compute the queue length at the stationary state: if $\sum_{j \leq n^*} \lambda_j = \gamma$, the queue length at the stationary state satisfies $c^- < c \leq c^*$; if $\sum_{j \leq n^*} \lambda_j > \gamma$, the queue length at the stationary state is $c = c^*$; □

Remark 1. *As an extension to the above setting, we can consider a setting with T time interval, where within each time interval t, the packets are sent with constant sending rate $\lambda_1(t), \ldots, \lambda_n(t)$, but across different time interval, the sending rate of each packet can vary, i.e., $\lambda_i(t) \neq \lambda_i(t')$ for $t \neq t'$. Suppose each time interval is sufficiently*

long such that the system can reach its stationary state, then we can apply the above theorem within each time interval to characterize the behavior of the algorithms at the stationary state.

Remark 2. *We briefly discuss the behavior of each algorithms in their stationary state.*

For PIFO, the queue is filled with packets $n_* + 1$, but they cannot be popped. For packet $i \leq n_*$, once it is received, it gets output. For packet $i = n_* + 1$, it can be admitted but cannot be output. For packet $i > n_* + 1$, it cannot be admitted.

For AIFO, the queue is filled with packets $i \leq n_*$. The AIFO outputs packets in the queue in a random sequence (since their arrival time is random). The AIFO admits packets according to the rule specified before: packets $i \leq n_*$ will be admitted, and packets $i > n_*$ cannot be admitted.

For FIFO, the queue is filled with all kinds of packets, and the number of each type of packets are proportional to their sending rate. And the packets are admitted and output at random.

Remark 3. *So long as we assume the system stays in its stationary state for sufficiently long time, its behavior would be nearly decided by that in the stationary state.*

This theorem means that at the stationary state, the departure rate of each packet rank with AIFO is the same as that with PIFO. The behavior of FIFO is very different from those of AIFO and PIFO. As FIFO does not do any scheduling, the departure rate of a rank is proportional to the arrival rate of the rank. The theorem also shows that with admission control, the queue length with AIFO is slightly smaller than that of PIFO.

Admitted packet set. Consider a time interval from 0 to T and n packet ranks. Let $a_i(t)$ be the departure rate of rank i with AIFO, and $p_i(t)$ be the departure rate

of rank i with PIFO. We can prove properties for the difference between the departure packets with AIFO and those with PIFO.

Theorem 4. *Adopt the assumption in Theorem 3. Suppose the systems are initialized at time 0, and after time t_0 both PIFO and AIFO reach and stay at their stationary states. Then for the gap measure defined in Eq. (4.1), we have*

$$\lim_{T\to\infty} \Delta(T) = 0.$$

Proof. Let t_0 be the maximum of the time for the AIFO and PIFO reaching its stationary state. Suppose the sending rate of a packet is at most M. Recall that from t_0 to T, $a_i(t) = p_i(t)$ as shown in the theorem. Then we have the following estimation:

$$\begin{aligned}
\Delta(T) &= \frac{\sum_{i=1}^{n} |\int_{t=0}^{T} p_i(t)dt - \int_{t=0}^{T} a_i(t)dt|}{\sum_{i=1}^{n} \left(\int_{t=0}^{T} p_i(t)dt + \int_{t=0}^{T} a_i(t)dt\right)} \\
&\le \frac{\sum_{i=1}^{n} \left(|\int_{t=0}^{t_0} (p_i(t) - a_i(t))dt| + |\int_{t=t_0}^{T} (p_i(t) - a_i(t))dt|\right)}{\sum_{i=1}^{n} \left(\int_{t=0}^{T} p_i(t)dt + \int_{t=0}^{T} a_i(t)dt\right)} \\
&= \frac{\sum_{i=1}^{n} |\int_{t=0}^{t_0} (p_i(t) - a_i(t))dt|}{\sum_{i=1}^{n} \left(\int_{t=0}^{T} p_i(t)dt + \int_{t=0}^{T} a_i(t)dt\right)} \\
&\le \frac{nM \cdot t_0}{\sum_{i=1}^{n} \left(\int_{t=0}^{T} p_i(t)dt + \int_{t=0}^{T} a_i(t)dt\right)} \\
&\le \frac{nM \cdot t_0}{\sum_{i=1}^{n} \left(\int_{t=t_0}^{T} p_i(t)dt + \int_{t=t_0}^{T} a_i(t)dt\right)}.
\end{aligned}$$

Note that for both the systems, at the stationary state ($t > t_0$), the total incoming/outcoming rate is constant γ, i.e., $\sum_{i=1}^{n} p_i(t) = \sum_{i=1}^{n} a_i(t) = \gamma$. Then we have

$$\sum_{i=1}^{n} \left(\int_{t=t_0}^{T} p_i(t)dt + \int_{t=t_0}^{T} a_i(t)dt\right) = 2\gamma(T - t_0),$$

which implies

$$\begin{aligned}
\Delta(T) &\le \frac{nM \cdot t_0}{\sum_{i=1}^{n} \left(\int_{t=t_0}^{T} p_i(t)dt + \int_{t=t_0}^{T} a_i(t)dt\right)} \\
&\le \frac{nM \cdot t_0}{2\gamma(T - t_0)}.
\end{aligned}$$

Note that (1) the nominator is a constant that is independent of T; and (2) the denominator keeps cumulating with constant non-zero rate at its stationary state.

Therefore for any small tolerance $\epsilon > 0$, let the running time T be

$$T > t_0 + \frac{nMt_0}{2\gamma\epsilon},$$

we have

$$\Delta(T) < \epsilon.$$

To sum up, if the system run for sufficiently long time, the difference between PIFO and AIFO tends to be negligible. □

Theorem 3 already provides a strong guarantee on the departure rate of each rank. Theorem 3 goes further to show the gap on the difference of the dequeued packets. It proves that Δ defined in Section 4.3 is close to 0, meaning that AIFO and PIFO dequeue the same set of packets.

4.4.4 Data Plane Design and Implementation

We describe the data plane design to implement AIFO on a programmable switch. We emphasize that the algorithm of AIFO itself is independent of the hardware architecture, and can be implemented on programmable switch ASICs, FPGAs or network processors. The purpose here is to provide a concrete data plane design and implementation to demonstrate the viability of AIFO. We implement AIFO with 827 lines of code in P4. The implementation can run on Intel Tofino at line rate. We describe the major challenges and our solutions in our design and implementation.

Queue length estimation for the temporal component. The main challenge for the temporal component is to maintain the dynamic threshold $\frac{1}{1-k}\frac{C-c}{C}$ based on the queue length. The queue length information is managed by a module called traffic manager which sits between the ingress pipe and the egress pipe. The difficulty is that for commodity switches including Intel Tofino, the queue length information can only be obtained when a packet goes through the traffic manager, and thus can only

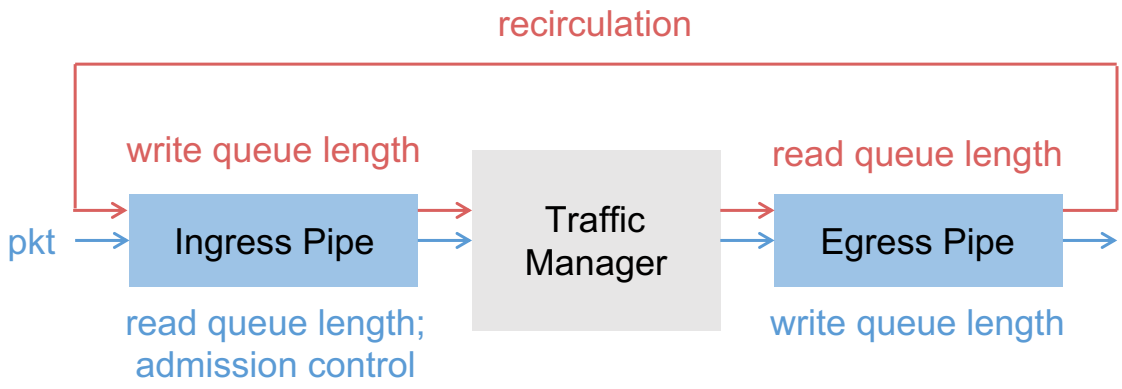


Figure 4-5. Worker packets carry queue length information from egress pipe to ingress pipe via recirculation. Normal packets read queue lengths and make admission control decisions at ingress pipe. Normal packets also write queue lengths at egress pipe.

be read at the egress pipe. However, AIFO requires the queue length to compute the threshold at the ingress pipe in order to make admission control decisions.

To address this challenge, we design a recirculation-based solution to bring the queue length information from the egress pipe to the ingress pipe. Specifically, we use a register array to store the queue length for each egress port at the egress pipe, denoted by q_len_egress. Packets can write the queue length value into q_len_egress after passing through the traffic manager. At the same time, we have a copy of the register array at the ingress pipe, denoted by $q\_len\_ingress$. We use a set of *worker* packets to read the queue lengths from q_len_egress at the egress pipe. The worker packets are recirculated to enter the ingress pipe again when they leave the egress pipe, and they update the queue lengths in $q\_len\_ingress$ using the values they read.

As the worker packets make the queue lengths ready in the ingress pipe, a normal arriving packet can then access the queue length information in the ingress pipe. After the routing decision is made for the packet (i.e., the egress port is known), it can read the queue length of its egress port from $q\_len\_ingress$. Then the threshold $\frac{1}{1-k}\frac{C-c}{C}$ can be calculated with the queue length to decide whether to admit or drop the packet. If the packet is admitted, it also writes the current queue length to the

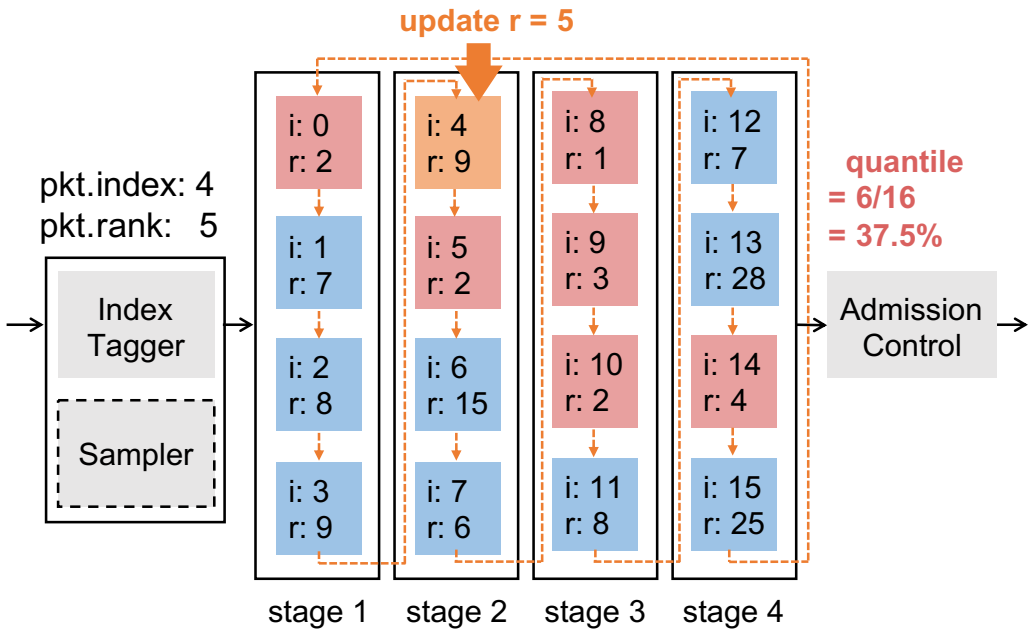
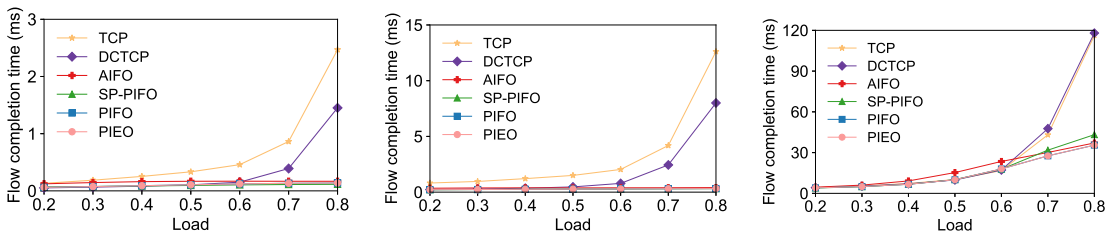


Figure 4-6. Compute quantile with a sliding window.



(a) Average FCT for small flows. (b) 99th FCT for small flows. (c) Avearage FCT for large flows.

Figure 4-7. Simulation results of web search workload to minimize FCT.

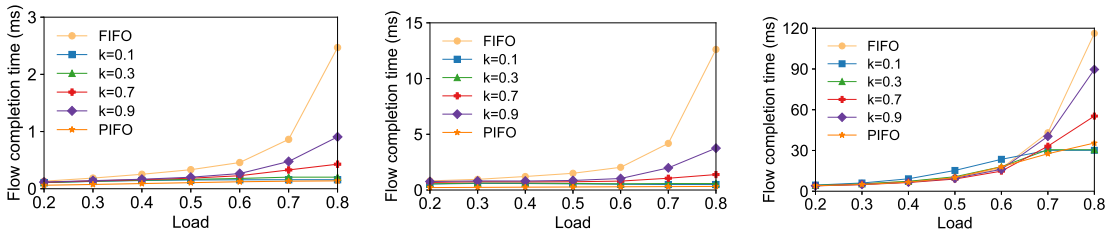
egress pipe. Figure 4-5 illustrates how the solution works.

Since the worker packets keep being recirculated all the time, they only go through a designated recirculation port, and thus would not contribute to the queue lengths of the egress ports. Assuming it takes 200 ns for a worker packet to go through the pipeline and be recirculated, for a port with $10Mpps$ rate, it would only cause a bias of 2 packets, which is negligible. Also note that for switches that support reading queue length directly in the ingress pipe (e.g., Intel Tofino 2), recirculation is not needed.

Quantile estimation for the spatial component. The spatial component estimates the quantile of the rank of each arriving packet. We use a set of stages to implement a sliding window to store recent packets and estimate quantiles. Programmable switches normally support accessing several registers per stage, e.g., $m = 4$ registers per stage. In order to support a sliding window with n slots, we need n/m stages. We use m registers per stage over n/m stages, and use n registers in total. The index of each register is from 0 to $n-1$, and it indicates the position of the packet in the sliding window. The value of register i stores the rank of the packet at position i in the sliding window. Figure 4-6 shows an example with $n = 16$ and $m = 4$. We use 4 stages and use 4 registers per stage, with a total of 16 registers. Each register stores the rank for a packet in a sliding window of 16 recent packets.

We use an index tagger module to track the sliding window. The index tagger module keeps a circular counter from 0 to $n-1$. It assigns its counter the index of an arriving packet ($pkt.index$), and then increments its counter by one. The counter is reset to 0 when it reaches n. The packet index indicates which register stores the rank of the oldest packet in the sliding window, and thus should be updated with the rank of the arriving packet. In Figure 4-6, $pkt.index$ is 4, and thus the value of the first register at stage 2 (i.e., the register with $i = 4$) is updated with the rank of the arriving packet. The index $pkt.index$ will be set as 5 for the next packet and point to the second register at stage 2 (following the dotted arrow).

At the same time, when a packet goes through each stage, the switch also compares the rank of the packet with the value in each register with an ALU. Each ALU outputs a result indicating whether the packet rank is smaller than the register value: if the packet rank is smaller, $output = 1$; otherwise, $output = 0$. By summing up the $output$s of all ALUs together, we get the relative ranking of the arriving packet in the sliding window: $q = \sum_i output_i$. The quantile of the arriving packet can be computed by dividing q by the length of the window: $W.quantile(pkt) = q/n$.



(a) Average FCT for small flows. (b) 99th FCT for small flows. (c) Average FCT for large flows.

Figure 4-8. The effect of parameter k.

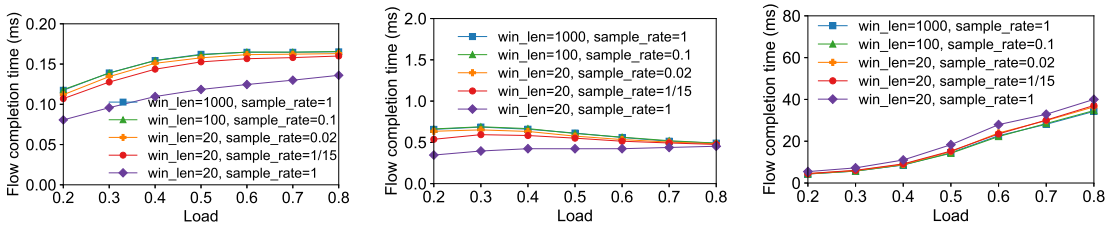
In Figure 4-6, the rank of the arriving packet is 5. The rank is smaller than the values of 6 registers, which are marked with red in the figure. As the size of the sliding window n is 16, the quantile is $6/16 = 37.5\%$.

While our evaluation results show that a small sliding window size (e.g., 20) is sufficient for many common scenarios, a large sliding window is sometimes needed for certain workloads. However, commodity switches normally provide only a few stages and a small amount of memory. To efficiently use precious switch resources, we use a sampling method to virtually scale up the sliding window size by adding a sampler aside with the index tagger. For example, instead of using a window with the size of 1000, we can use a smaller window with the size of 20, and set the sampling rate as 0.02.

As both the queue length (c) and the quantile ($\frac{q}{n}$) are available, we can make the admission control decision based on the condition $\frac{q}{n} \leq \frac{1}{1-k}\frac{C-c}{C}$. This condition can be transformed to $\frac{C\cdot(1-k)}{n} \cdot q + c \leq C$. Since C, k, and n are constants, $\frac{C\cdot(1-k)}{n} \cdot q$ can be easily calculated in one stage with a math unit available in programmable switches.

4.5 Evaluation

In this section, we provide experimental results to demonstrate the performance of AIFO. We first evaluate AIFO using simulations to show that AIFO can achieve high performance in a large-scale datacenter environment. In the simulations, we benchmark



(a) Average FCT for small flows. (b) 99th FCT for small flows. (c) Average FCT for large flows.

Figure 4-9. The effect of window length and sampling rate.

AIFO with state-of-the-art solutions to demonstrate its end-to-end performance. Besides, we also evaluate the effect of different parameters, and the admitted packet set of AIFO. At last, we evaluate our prototype for AIFO on a Intel Tofino switch in a hardware testbed.

4.5.1 Packet-Level Simulations

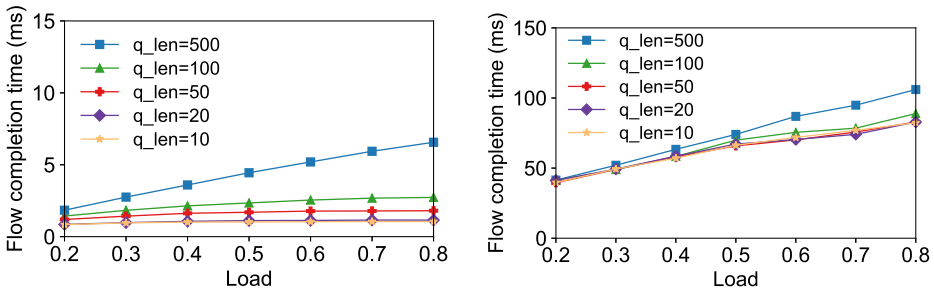
We use packet-level simulations to evaluate AIFO in a large-scale datacenter environment. We use a similar setting as recent works on packet scheduling [65, 85]: a leaf-spine topology which contains 9 leaf switches, 4 spine switches and 144 servers, and the bandwidth of the access and leaf-spine links is set at 10Gbps and 40Gbps, respectively. The simulations are conducted with Netbench [84], a packet-level simulator.

We evaluate two use cases of programmable packet scheduling: minimizing FCT and providing fairness. (*i*) We use AIFO to implement pFabric, and compare it with TCP, DCTCP as well as state-of-the-art approaches PIFO [63], SP-PIFO [65], and PIEO [109] under a realistic traffic workload: web search workload [85]. We also conduct a sensitivity analysis to evaluate and analyze the effect of different parameters (i.e., queue length, scaling parameter k, window length and sampling rate) on AIFO and the admitted packet set of AIFO. (*ii*) We implement Start-Time Fair Queueing (STFQ) on top of AIFO and compare it with other state-of-the-art solutions. For

AIFO, we set the target queue length as 20, $k = 0.1$, window length as 20, and sampling rate as $\frac{1}{15}$ by default.

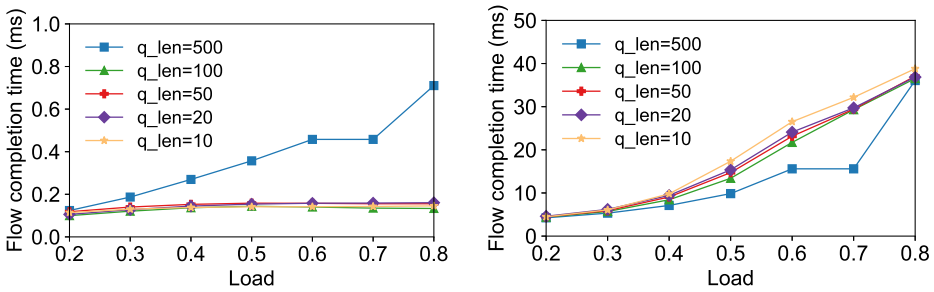
Minimizing FCT with AIFO. We first show the performance of AIFO when implementing SRPT for pFabric [85] to minimize FCT under the web search workload. The traffic starts according to a Poisson distribution. For comparison, we also implement SRPT with SP-PIFO and PIFO, and compare them with TCP and DCTCP. In addition, we consider PIEO, which is a more scalable design for programmable packet scheduling compared with PIFO. We use pFabric as the transport layer for AIFO, PIFO and SP-PIFO at the hosts. Figure 4-7 shows the average FCT for small flows (Figure 4.7(a)), the $99th$ percentile FCT for small flows (Figure 4.7(b)), and the average FCT for large flows (Figure 4.7(c)). AIFO, PIFO, PIEO, and SP-PIFO can achieve much lower FCT compared with TCP and DCTCP, especially when the load is high. Among all these approaches, PIFO and PIEO achieve the best performance as it enforces strict priority with a PIFO queue. The performance of SP-PIFO is close to PIFO. While PIFO requires a PIFO queue which is hard to implement and SP-PIFO requires multiple FIFO queues (eight queues in the simulations), AIFO achieves a good performance that is close to PIFO and SP-PIFO with a single FIFO queue. Besides, AIFO can deal with different sizes of traffic well as the admission control threshold can adapt the current workload traffic dynamically. Figure 4-7 shows that as the traffic load grows, the FCT for AIFO does not go up as TCP or DCTCP does, and the gap between AIFO and PIFO/SP-PIFO gets smaller.

The effect of parameter k. The headroom parameter k controls how aggressively AIFO drops high-rank packets. We set k among $0.1 \sim 0.9$ and compare the results with FIFO and PIFO. Figure 4-8 shows the results. AIFO with smaller k always delivers better performance than larger k for small flows, and it also delivers better performance for large flows when the traffic load is big (e.g., 0.7, 0.8). The reason is that with a small k, AIFO drops packets aggresively and keeps the buffer shallow so



(a) Average FCT for small flows. (b) Average FCT for large flows.

Figure 4-10. The effect of queue length on 1G/4G network.

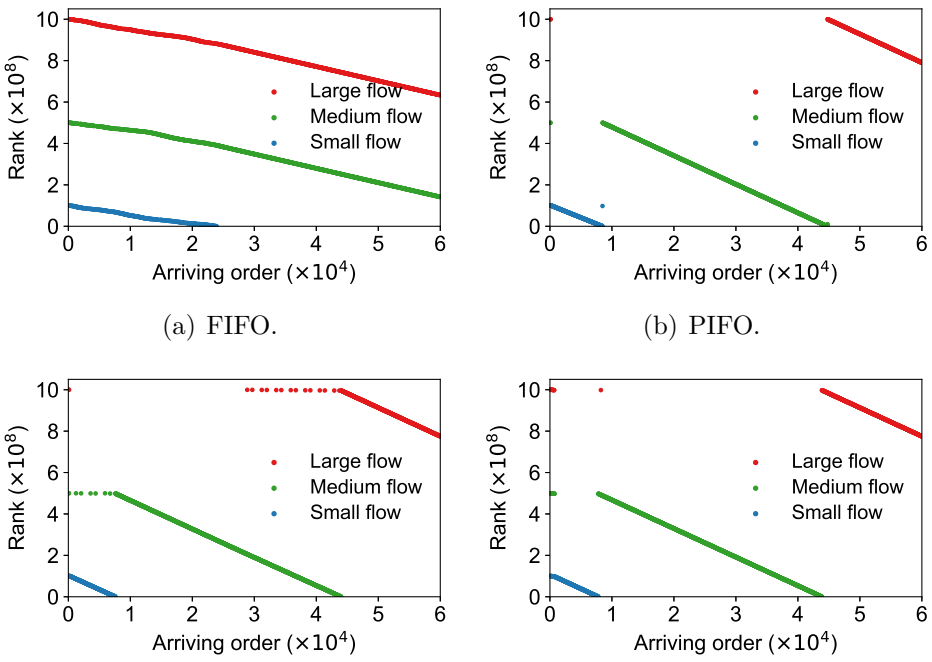


(a) Average FCT for small flows. (b) Average FCT for large flows.

Figure 4-11. The effect of queue length on 10G/40G network.

that the admitted packets get low latency. When k is small, AIFO delivers a close performance compared with PIFO. As we increase k, AIFO admits more packets and it becomes closer to FIFO. When the traffic load grows, the queue buffer accumulates quickly, and it leads to a large delay. While dropping packets aggressively harms large flows especially when the traffic load is not big and the network capacity is underutilized, we show that the harm is slight compared with the benefit it brings to small flows. When $k = 0.1$, the average and $99th$ percentile FCT for small flows is about $9\times$ lower than that of $k = 0.9$, and the FCT for large flows is only slightly higher than the lowest.

The effect of window length and sampling rate. We also evaluate how the sliding window length affects the performance of AIFO and how well a small sliding window approximates a large sliding window with sampling for AIFO. As shown in



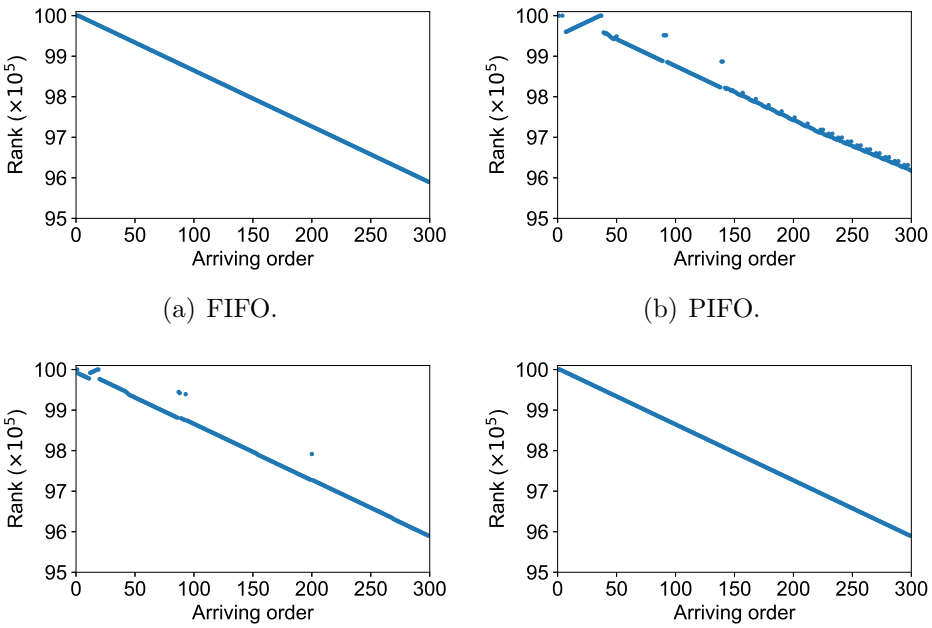
(a) FIFO. (b) PIFO.

(c) SP-PIFO. (d) AIFO.

Figure 4-12. Packet distribution logged at the receiver. Three senders send one flow each to a receiver at the same time. The size of the three flows are 100MB (large), 50MB (medium) and 10MB (small), respectively. The link between the switch and the receiver is the bottleneck.

Figure 4-9, when the window length is 20, the performance is better than that when the window length is 1000 for small flows, but worse for large flows. It is because that a large window records packets for a longer time, and the possibility for packets from large flows (high-rank packets) to be admitted is more stable. As a result, there are more high-rank packets admitted into the queue in the long run, which makes the FCT for small flows higher and FCT for large flows lower. It is interesting to see in Figure 4.9(b) that the $99th$ percentile FCT is decreasing as the traffic load grows when $win\_len = 1000$. The reason is that when the window is large and the traffic load is low, the quantile is less accurate. The flows that experience deep buffer and inaccurate quantile estimation would have larger FCTs, and these flows would normally contribute to the $99th$ FCT.

By comparing lines <win_len=20, $sample\_rate$=0.02>, <win_len



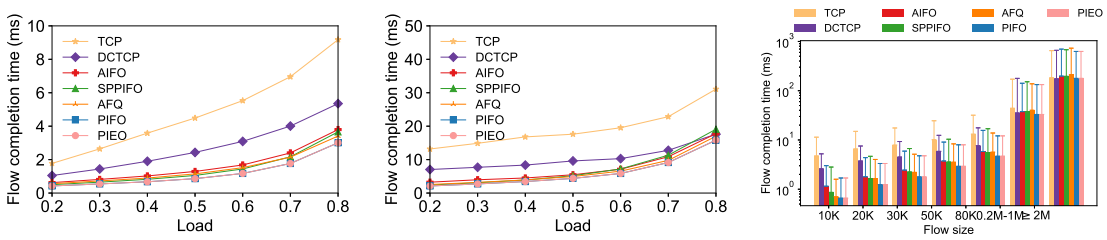
(a) FIFO. (b) PIFO.

(c) SP-PIFO. (d) AIFO.

Figure 4-13. The first 300 packets of the small flow logged at the receiver. The setting is the same as Figure 4-12: Three senders send one flow each to a receiver at the same time. The size of the three flows are 100MB (large), 50MB (medium) and 10MB (small), respectively.

=100, *sample_rate*=0.1> and <*win_len*=1000, *sample_rate*=1>, we can see that AIFO does not require a very precise quantile and a small window can approximate a large window with sampling. This is important to the practicability of AIFO as a window with 20 slots can be implemented in programmable switches with tiny resource consumption.

The effect of queue length. To evaluate the impact of queue length on the performance of AIFO, we use different queue lengths and run simulations on both a 1G/4G network (access link: 1Gbps, leaf-spine link: 4Gbps) and a 10G/40G network. As shown in Figure 4-10 and Figure 4-11, AIFO is more sensitive to the change of queue length when the bandwidth is low or when the traffic load is high. Figure 4-11 shows that FCT achieved by AIFO when $q\_len = 20$ is close to $q\_len = 100$ on the 10G/40G network. However, FCT achieved by AIFO when $q\_len = 20$ is much



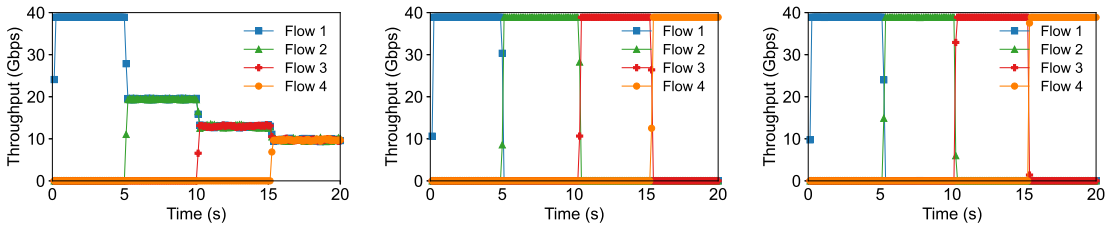
(a) Average FCT for small flows. (b) 99th FCT for small flows. (c) FCT breakdown for 70% load.

Figure 4-14. Simulation results of web search workload with fair queueing.

smaller than $q\_len = 100$ in Figure 4-10 on the 1G/4G network. This is because it takes a while for a long queue to drain when the bandwidth is low, which leads to a considerable queueing delay. A relatively small queue benefits the FCT.

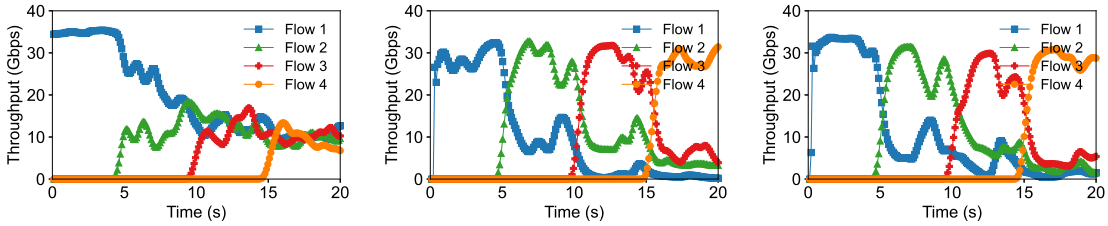
Admitted packet sets. Recall that we indicate in Theorem 4 that the sets of packets dequeued by AIFO and PIFO are similar. This experiment examines the gap between AIFO and PIFO in terms of the difference between the packets dequeued by AIFO and those dequeued by PIFO. Here we use four servers and the servers are connected with a Top-of-Rack switch. The bandwidth of the links between the servers and the switch are set as 1Gbps. We let three servers serve as senders, and the other server serves as a receiver. Each sender sends one flow to the receiver at the same time, and the sizes of the flows are 100MB (large), 50MB(medium), and 10MB(small), respectively. The servers run pFabric as the transport and the flows are tagged with the remaining flow size as their rank. The switch is programmed to support SRPT with AIFO, PIFO or SP-PIFO.

We log the ranks of the first 60000 packets received by the receiver, and plot the log in Figure 4-12. The x-axis is the arriving order of the packets, and the y-axis is the rank of the packets. As shown in Figure 4-12, when the network is running FIFO without admission control and packet scheduling, the three flows share the bandwidth and the small flow (blue) finishes late. For the other three solutions (AIFO, PIFO, SP-PIFO), the small flow finishes at about the same time, and it finishes much



(a) FIFO. (b) SP-PIFO. (c) AIFO.

Figure 4-15. Testbed experiments for UDP. Four flows start one by one every five seconds. Flows have different ranks: R(Flow 1) $>$ R(Flow 2) $>$ R(Flow 3) $>$ R(Flow 4).



(a) FIFO. (b) SP-PIFO. (c) AIFO.

Figure 4-16. Testbed experiments for TCP. Four flows start one by one every five seconds. Flows have different ranks: R(Flow 1) $>$ R(Flow 2) $>$ R(Flow 3) $>$ R(Flow 4).

earlier than it does with FIFO. Besides, it is shown that AIFO is closer to PIFO than SP-PIFO in terms of the admitted packets set: there are larger overlaps on the arriving order (x-axis) between the small flow (blue) and the medium flow (green), as well as between the medium flow and the large flow (red) in Figure 4.12(c), than those in Figure 4.12(d) and Figure 4.12(b).

Packet reordering. Besides the admitted set, another interesting metric is the dequeued order of the packets. PIFO always dequeues the packet with the lowest rank in the queue, which may cause out-of-order and harm the end-to-end performance. However, as AIFO only enforces admission control on a FIFO queue, it does not cause out-of-order.

We run the same setting as in Figure 4-12 and we only log the first 300 packets of the small flow in order to show the packet out-of-order clearly. As shown in Figure 4.13(a) and Figure 4.13(d), when we enable AIFO, the packet order of one

flow is the same as that with FIFO and there is no packet out-of-order. It is because that AIFO only uses a FIFO queue and does not do packet scheduling inside the queue. However, both PIFO and SP-PIFO get some out-of-order packets, as shown in Figure 4.13(b) and Figure 4.13(c). The reason is that PIFO and SP-PIFO always dequeue the packet with the lowest rank, while the packets with higher rank will be left in the queue and be scheduled later. With pFabric, the rank is based on the remaining flow size, so a later packet has a lower rank compared with an earlier packet. As a result, these two methods lead to a number of out-of-order packets. As SP-PIFO approximates PIFO with a set of FIFO queues, it causes fewer out-of-orders compared with PIFO.

Fair queueing with AIFO. Programmable packet schedulers like PIFO can be used to implement different kinds of packet scheduling algorithms by changing the rank computation function. Besides implementing SRPT to minimize FCTs, here we show how AIFO performs when we implement Start-Time Fair Queueing (STFQ) [99] on top of it for fair queueing. We also implement STFQ on top of PIFO, SP-PIFO and PIEO to compare them with AIFO. Besides, we include TCP, DCTCP, and the state-of-the-art fair queueing solution AFQ for comparison. We run the web search workload and show the average FCT for small flows (Figure 4.14(a)), the $99th$ percentile FCT for small flows (Figure 4.14(b)), and the breakdown of FCT for different flow sizes (Figure 4.14(c)). AIFO achieves a similar performance compared to the state-of-the-art approaches AFQ, SP-PIFO, PIFO and PIEO for both average FCT and tail FCT, and is significantly better than TCP and DCTCP. The FCT of AIFO for small flows is only 9.7% higher than AFQ and 3.6% higher than SP-PIFO, despite AIFO using only a single queue.

4.5.2 Testbed Experiments

We evaluate AIFO in the testbed. The testbed experiments are conducted in a hardware testbed with a 6.5Tbps Intel Tofino switch and five servers. Each server is configured with an 8-core CPU (Intel Xeon E5-2620 @ 2.1GHz) and a 40G NIC (Intel XL710). We run Ubuntu 16.04.6LTS with Linux kernel version 4.10.0-28-generic on the servers.

Both UDP traffic and TCP traffic are covered to examine the traffic differentiation with ranks. We use four servers as senders which send one flow each to a receiver. The four flows start one by one every five seconds. The link between the switch and the receiver is the bandwidth bottleneck. We manually tag different ranks for different flows, and the flow that starts later has a lower rank (i.e., higher priority): R(Flow 1) > R(Flow 2) > R(Flow 3) > R(Flow 4). For comparison, we also run FIFO and SP-PIFO in the same setting. For SP-PIFO, we enable 8 queues with strict priority in the traffic manager.

UDP. We first evaluate AIFO when the four flows are UDP flows. The four flows are all sending at 40Gbps (using DPDK [29]). Figure 4.15(a) shows that when the switch is running FIFO, the four flows converge to the same rate since they have the same sending rate and has the same possibility to be dropped. When AIFO is enabled (Figure 4.15(c)), as Flow 2 has a lower rank than Flow 1, packets from Flow 2 have a higher chance to get into the queue when the queue builds up. Consequently, Flow 2 gets all the bandwidth and the throughput to Flow 1 drops to zero when Flow 2 comes. Similarly, when Flow 3 comes, Flow 3 gets the bandwidth between 10 seconds and 15 seconds, and when Flow 4 comes, Flow 4 occupies most of the bandwidth. The result is almost identical to SP-PIFO (Figure 4.15(b)) as the flow with the lowest rank always occupies most of the bandwidth.

TCP. We also evaluate AIFO when there are four TCP flows. We use TCP Cubic

Resource Type	AIFO	SP-PIFO
Match Crossbars	10.94%	8.27%
Gateway	22.92%	17.71%
Hash Bits	3.91%	2.66%
SRAM	6.98%	15.31%
TCAM	0%	0.35%
Stateful ALUs	39.6%	16.67%
Logical Table IDs	25%	18.75%

Table 4-I. Resource consumption of AIFO and SP-PIFO prototypes on Intel Tofino. Each number indicates the percentage of resources consumed for the corresponding type.

as the congestion control algorithm on the servers. Figure 4-16 shows the results. In the beginning, Flow 1 reaches around 34Gbps as it occupies the entire link. As Flow 2, Flow 3, and Flow 4 start one by one, when the switch is running FIFO, the four flows converge to a similar rate as TCP congestion control provides fair bandwidth allocation. However, when AIFO is enabled, lower-rank flows get higher throughput: Flow 4 gets the highest throughput at about 30Gbps, while Flow 1 gets the lowest throughput at 200Mbps–1Gbps. SP-PIFO also delivers a similar result. Note that, compared with the results of UDP flows in Figure 4-15, AIFO acts less aggressively in the TCP scenario: the high-rank TCP flows can still get about 3Gbps–5Gbps throughput, while the throughput of the high-rank UDP flows in Figure 4.15(c) is close to 0. This is because in the UDP scenario, the four flows are sending at a fixed rate. This makes it easy for AIFO to enter a stationary state and AIFO would accept most of the low-rank packets. However, for the TCP scenario, as the flow rate is dynamic because of congestion control, the admission threshold of AIFO changes dynamically and the high-rank packets can have some chance to get into the queue.

Resource consumption. AIFO uses only one queue and achieves similar performance as SP-PIFO with eight queues. Table 4-I lists the consumption of other switch resources.

It shows that AIFO has a higher demand on Match Crossbars, Gateway, Hash Bits, ALUs and Logical TableIDs, while SP-PIFO has a higher demand on SRAM and TCAM.

4.6 Related Work

Programmable networking. The emergence of programmable networking has triggered many novel applications in network data plane [17, 39, 41, 42, 46–50, 52, 82, 90, 92, 94, 110]. Among them, programmable packet scheduling [63, 95] is an attracting direction. Programmable packet scheduling in the data plane as opposed to traditional fixed-function packet scheduling [55–58, 58, 66] is a relatively new concept. After PIFO [63] and UPS [95], several solutions for enabling programmable scheduling have proposed a combination of new abstractions, new algorithms, and new queue structures [64, 109, 111, 112]. However, many of these rely on new hardware designs. SP-PIFO [65] recently shows that efficient programmable packet scheduling can be approximated by using existing devices with as few as eight queues. In this chapter, we show that AIFO can closely approximate PIFO with just one queue.

Priority-based scheduling. Priority-based scheduling is a classic scheduling discipline [96] that is often used in the networking context to minimize the average completion time of flows [85, 113–115] and coflows [116, 117] in both clairvoyant (size is known a priori) and non-clairvoyant (unknown size) scenarios. In the latter case, most of the solutions boil down classic solutions such as Multi-level Feedback Queues (MLFQ) [118] and its continuous approximations [119, 120]. Programmable packet scheduling uses the notion of ranks, which is similar to priorities, but more general in the sense that the definition of ranks can be programmed based on the requirements of users. This general notion has been shown to be able to support a wide variety of different packet scheduling algorithms for different objectives [63].

Active queue management (AQM). Working in conjunction with packet scheduling algorithms, AQM performs admission control by probabilistically dropping packets to prevent congestion. AQM is simple and implemented widely in most switches (e.g., RED [102]). There are many variations: e.g., to improve fairness [103, 104] and to provide bounded worst-case packet queuing delay [105] to name a few. Unlike traditional AQM proposals, AIFO proactively drops packets based on their relative ranks instead of randomly dropping them.

4.7 Conclusion

We present AIFO, a new approach for programmable packet scheduling that only uses a single FIFO queue. AIFO computes a rank quantile for a coming packet and decides whether to admit the packet into the queue based on the rank quantile and the current queue length. We build a prototype for AIFO on programmable switches. Our simulations and testbed experiments show that AIFO delivers high performance and closely approximates PIFO. Besides, we also theoretically prove that AIFO provides bounded performance to PIFO. We believe AIFO is a promising solution for realizing programmable packet scheduling with minimal hardware resource consumption—as few as a single FIFO queue.

Chapter 5

Lumina: Fine-grained Analyzation Tool for Hardware Offloaded Network Stacks.

5.1 Introduction

Modern cloud applications demand high throughput and ultra-low processing latency (several μs) with low CPU overhead. Traditional TCP/IP stacks in operating system (OS) kernel are ill-suited to these requirements. Therefore, cloud providers tend to offload their network stacks into network interface card (NIC) hardware to achieve better performance and free-up CPU cycles. Hardware offloaded networking techniques (e.g., RDMA, SRD [121]) have been widely deployed in various of areas including data storage [122, 123], deep learning [124, 125], etc.

To best utilize the great performance brought by hardware offloading, network developers should be familiar with the behaviors of hardware network stacks. While reading specifications provided by vendors is helpful, the specifications may not provide enough details of actual implementations. In the past decades, there have been many tools [126–129] to test software network stack implementations. These tools enable users to test the correctness of the network stack implementation and understand the differences between specification and actual implementation. For

example, packetdrill [126] uses libpcap or TUN device as a "shim layer" to inject and consume packets. Users can customize test cases based on the scripting language provided by packetdrill to test their TCP/UDP/IP network stacks. However, testing tools for hardware network stacks are still in a very initial stage. For example, to test RDMA, network operators typically run synthetic workloads to measure end-to-end performance in testbed and test clusters. Despite its effectiveness to reveal significant functional bugs, this approach suffers from noises of different applications and network conditions, and cannot accurately capture micro-behaviors like per-packet transmission time.

Enabling precise and reproducible testing for hardware network stacks has two key challenges. First, as network stacks are implemented in the hardware without passing the kernel, we cannot directly inject packets/events or measure behaviors at the end host. Thus the testing tools [126] that use a shim layer to inject packets cannot work for hardware network stacks. Second, as hardware network stacks provide high throughput and ultra-low latency, the testing tool should be able to interact with them at high speed with low extra delay.

In this chapter, we design and implement Lumina, a tool to test the correctness and performance of hardware network stacks. To overcome the challenge that we cannot directly interact with the hardware implementation, Lumina adopts an in-network solution: we connect two under-test hardware network stacks to a programmable middlebox and use the programmable middlebox to inject network events and test the behaviors. As mentioned above, the programmable middlebox should not introduce substantial latency. With the rapid development of programmable network devices, we now have various of choices for such high-performance programmable middleboxes: programmable switches and smartNICs. Each of them can support programmability to some extent and high-speed packet processing.

However, due to the limited resource (e.g., on-chip memory, processing cycles) on

programmable network devices, it is hard to realize complex measurement and testing tasks fully in the network. To this end, we utilize the mirror feature to generate a complete clone of packets and dump the cloned packets using dedicated servers. The dumped packet traces are used for further analysis.

The design goal of Lumina is to reliably enable developers to write *reproducible* tests and inject *deterministic* events with *user-friendly* interfaces. To achieve this goal, we co-design three components: event injector (on middlebox), traffic generator (on traffic servers with under-test hardware), and packet dumper (on mirror servers). Lumina creates a dedicated connection between traffic generator and event injector to share the traffic metadata so that event injector can take user-friendly configurations as input and generates deterministic entries for the event injector. When doing mirroring, Lumina embeds important metadata (sequence number, timestamp, event type) in mirrored packets to enable reliable and reproducible tests. Besides, Lumina adopts per-packet load balancing to evenly distribute mirrored packets across all the CPU cores of traffic dumpers, which guarantees the reliability and efficiency.

In summary, we make the following contributions.

- We propose Lumina to enable testing the correctness and performance of hardware network stack implementation.
- We co-design event injector, traffic generator and packet dumper to provide reliable and user-friendly interfaces for network developers to write reproducible tests and inject deterministic events.
- We prototype Lumina to test RDMA NICs. Our microbenchmark experiments demonstrate the efficiency and negligible-overhead of Lumina. Besides, we present our test results on three widely used RDMA NICs, related to fast retransmission, timeout, and congestion notification.

5.2 Background and Motivation

In this section, we first introduce the background of hardware offloaded network stack. Then we present the motivation and challenges of hardware network stack testing.

5.2.1 Hardware offloaded network stacks

We list some representative hardware offloaded network stack techniques as follows.

TCP offload engine (TOE): Unlike techniques [130–132] that only offload some operations, TOE offloads processing of the *entire* TCP/IP stack to the NIC. This technology is supported by some vendors (e.g., Chelsio Terminator 5 [133] and Broadcom NetXtreme II 5708 [134]). TOE not only frees up CPU cycles, but also reduces PCIe traffic.

RDMA: RDMA enables NIC to transfer data from the pre-registered memory to the wire or from the wire to the memory. The networking protocol is implemented on the NIC, thus bypassing OS kernel. Unlike TCP, RDMA provides message semantics rather than stream semantics. The high performance computing (HPC) community has long deployed RDMA in special purpose clusters [135] using InfiniBand (IB) [136]. In recent years, cloud providers also deploy RDMA in datacenters over Ethernet and IP networks [122, 137] to free CPU cores and accelerate communication-intensive workloads, e.g., storage and machine learning (ML). To this end, most of cloud providers adopt RDMA over Converged Ethernet (RoCE) v2 given its large vendor ecosystem [138–142]. Another alternative is Internet Wide Area RDMA Protocol (iWARP) [143, 144] which runs RDMA over TCP/IP on the NIC.

Scalable reliable datagram (SRD): SRD is a customized transport protocol implemented in Amazon Web Services Nitro networking card [121]. SRD spays packets over multiple paths to minimize the chance of hotspots and use delay information for congestion control. Unlike TCP and RDMA reliable connection (RC), SRD provides

reliable but out-of-order delivery and leaves order restoration to applications.

5.2.2 Motivation

With wide adoptions of hardware offloaded network stacks, it is important for network operators to have in-depth understandings of their behaviors. Recent years have witnessed the progress [126, 127, 145–147] in testing software network stacks. However, testing hardware offloaded network stacks is left behind. For example, to safely enable RDMA, network operators run synthetic workloads (using traffic generators like `perftest` [148]) and application load tests in lab testbed and test clusters before production deployments [122, 137]. While this approach can measure overall performance and reveal significant functional bugs, it cannot accurately capture micro-behaviors and suffers from noises due to variations in application and network conditions, let alone its high resource consumption. Hence, performance anomalies and bugs in these areas are likely to remain unnoticed in tests, and then hit production networks. These problems will be magnified as the link speed keeps increasing and hardware network stacks are becoming more and more complex.

To illustrate this problem, we use NVIDIA Mellanox ConnectX-4 RDMA NIC as an example. Shpiner et al. evaluated its performance over a lossy testbed network, and found it could preserve high goodput under synthetic incast workloads (Figure 4 and 5 in [149]). However, we find that the retransmission delay of this NIC is actually around 200μs (Figure 5-8 and 5-9), which is about 100 base round-trip times (RTTs). Given these limitations, we need a tool that can enable developers to easily write precise and reproducible tests for hardware offloaded network stacks. To this end, the tool should be able to interact with hardware network stacks (e.g., inject packet losses) in a flexible and deterministic manner, and accurately capture their micro-behaviors (e.g., per-packet transmission time).

Compared to software network stacks, hardware network stacks impose some

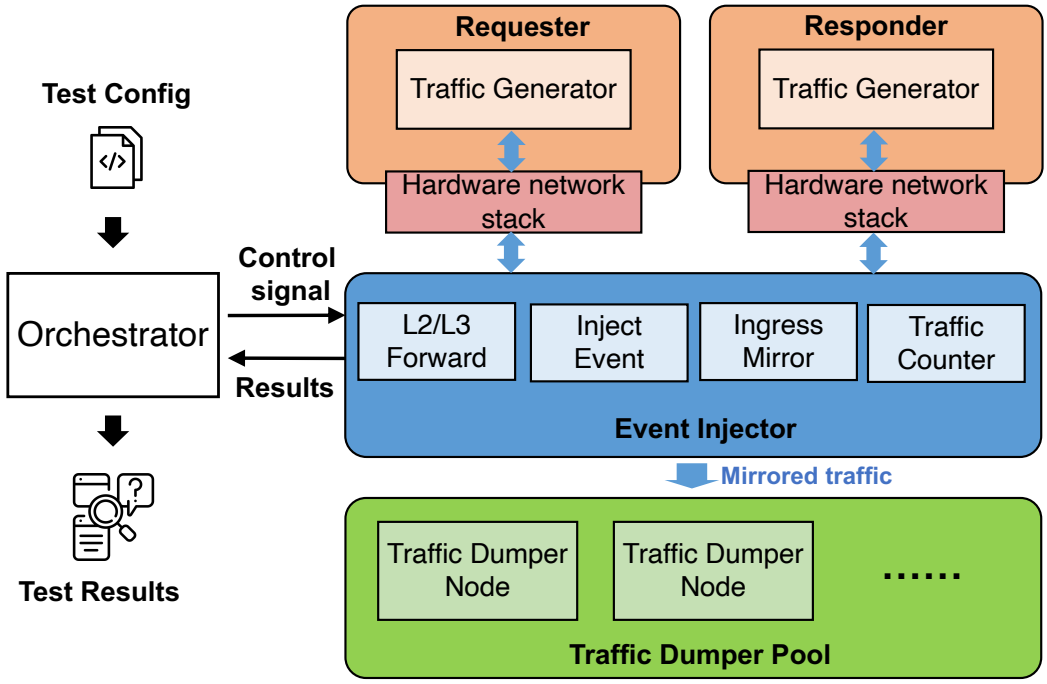


Figure 5-1. Lumina Overview.

unique challenges for testing tools. First, since hardware network stacks bypass kernel, we cannot interact with network stacks and measure their behaviors through shim layers, e.g., libpcap and TUN device [126] at the host as before. Second, hardware runs at high throughput and ultra-low latency. This translates to stringent performance requirements for test event injections and data plane measurement.

5.3 The Lumina Design

5.3.1 Overview

Motivated by above observations, we build Lumina, a tool that enables testing the correctness and performance of RDMA NIC implementations. In this chapter, we focus on RoCEv2[1] since it is the de facto hardware offloaded network stack technology in the cloud environment with wide support [138–140] and adoptions [122, 137, 150]. RoCEv2 encapsulates an IB transport packet using UDP and IP headers to enable routing over Ethernet/IP-based networks. We believe Lumina can be extended to

[1]In the following sections, we use RDMA, RoCE, and RoCEv2 interchangeably.

support other hardware network stack technologies.

In this section, we first present the design rationale of Lumina. Then we give an overview of Lumina. After that, we introduce the design of each mechanism in detail.

5.3.2 Design Rationale

The kernel bypass nature of hardware offloaded network stacks prevents us from directly injecting events and measuring behaviors at the end host. To meet this challenge, we embrace an in-network solution. We connect two hosts with hardware network stack under test to a *high-performance programmable middlebox*, which is used to emulate realistic and worst-case network scenarios. The middlebox should be able to forward traffic and be programmed to inject various events at the line rate with ultra-low extra processing delay. Recent industry progress on reconfigurable network hardware provides many options [19, 151–153] for the middlebox. In current prototype, we adopt the programmable switch.

However, the middlebox hardware may not have enough resource and flexibility (e.g., limited stateful memory and arithmetic operations) to realize complex measurement. Instead of in-device measurement, we dump all the packets by mirroring them from the middlebox to dedicated servers. After that, we reconstruct the complete packet trace from dumped packets for further processing and analysis.

As shown in Figure 5-1, Lumina has four components: *Orchestrator*, *Traffic Generator*, *Event Injector*, and *Traffic Dumper*. To run a test, the orchestrator takes a user configuration file as input, sets up the environment, and sends Remote Procedure Calls (RPCs) to each component to coordinate their executions.

Lumina uses two hosts with the same bandwidth capacity to generate traffic. Each host is equipped with the under-test hardware offloaded network stack and runs a traffic generator instance. We use one host as the requester and the other one as the responder. They generate traffic based on the configurations conveyed from the

```
requester:
  workspace: /home/foo/bar/
  username: test
  control-ip: cx4-testing-traffic-requester
  nic:
    type: cx4
    if-name: enp4s0
    switch-port: 144
    ip-list:
      - 10.0.0.2/24
      - 10.0.0.12/24
  roce-parameters:
    dcqcn-rp-enable: False
    dcqcn-np-enable: True
    min-time-between-cnps: 0
    adaptive-retrans: False
    slow-restart: True
```

Listing 5.1. Traffic Generation Host Configuration Snippet

orchestrator (§5.3.3).

The event injector forwards the traffic and injects configured events, e.g., ECN marks, packet losses and corruptions (§5.3.4). In the meantime, the event injector also mirrors all the RoCE packets to the traffic dumper pool, which consists of multiple servers, for offline analysis in the future (§5.3.5).

Once the traffic finishes, the orchestrator collects results from the other components, e.g., dumped packets, NIC counters and log files. It reconstructs the complete packet trace from dumped packets collected by traffic dumper servers. After that, users can parse the packet trace and other results to analyze behaviors of the hardware network stack (§5.3.6).

5.3.3 Traffic Generation

Before starting traffic generators, the orchestrator first configures IP addresses and network stack settings, e.g., congestion control and loss recovery parameter, of traffic generation hosts. Listing 5.1 gives an example.

After the configuration, the orchestrator starts traffic generator instances on both hosts. Our traffic generator adopts Reliable Connected (RC) transport, and supports RDMA send, receive, write, and read verbs. In this chapter, we use SEND, RECV, WRITE and READ to denote them, respectively. RDMA traffic generators

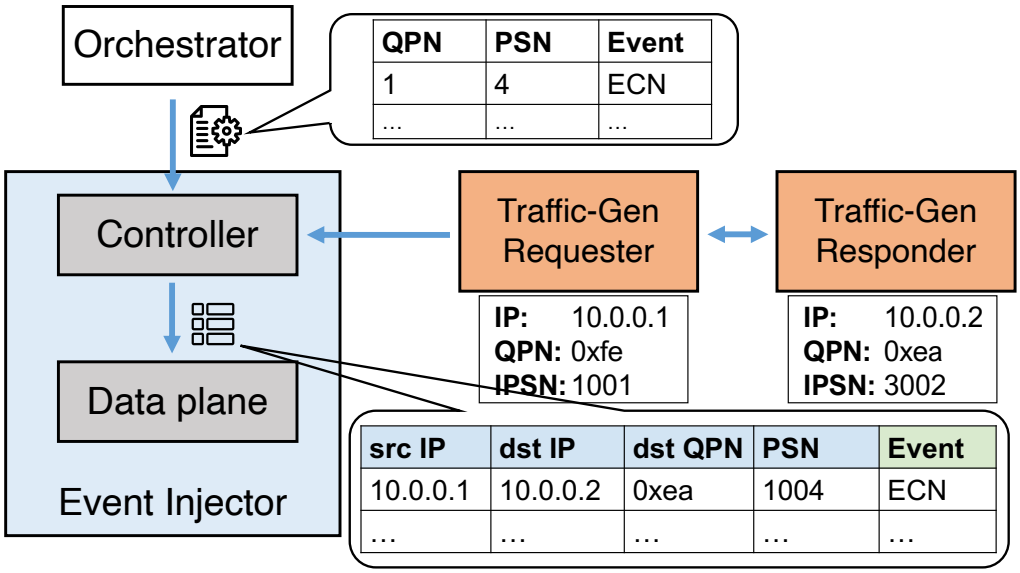


Figure 5-2. Lumina combines the runtime traffic metadata and intent-based traffic configuration to populate the match-action table for event injection.

communicate using one or multiple queue pairs (QPs). As shown in Listing 5.2, the user can configure many parameters of traffic generators, e.g., the number of QPs, retransmission timeout, and MTU.

After two traffic generators initialize objects such as QPs and memory regions (MRs), they exchange necessary metadata, e.g., QP number (QPN), packet sequence number (PSN), global identifier (GID), memory address and key, through a TCP connection. Since QPNs and PSNs are randomly generated at runtime and critical for the event injector to locate right packets, the traffic generator sends metadata information to the event injector as well (more details in §5.3.4).

After exchanging metadata and establishing QP connections, the requester posts work requests to generate RDMA traffic. The requester controls the total number of requests/messages and the maximum number of outstanding requests on each QP. In the case of SEND/RECV, the responder keeps posting RECV requests correspondingly. The requester can support barrier synchronization among QPs: the requester posts the next round of requests only after it gets the completions of the current round of requests across all the QPs.

```
traffic:
  num-connections: 2
  rdma-verb: write
  num-msgs-per-qp: 10
  mtu: 1024
  message-size: 10240
  multi-gid: true
  barrier-sync: true
  tx-depth: 1
  min-retransmit-timeout: 14
  max-retransmit-retry: 7
  data-pkt-events:
  # Mark the 4th pkt on the 1st QP conn
  - qpn: 1
    psn: 4
    type: ecn
    iter: 1
  # Drop the 5th pkt on the 2nd QP conn
  - qpn: 2
    psn: 5
    type: drop
    iter: 1
  # Drop the retrans 5th pkt on the 2nd QP conn
  - qpn: 2
    psn: 5
    type: drop
    iter: 2
```

Listing 5.2. Traffic and Event Injection Configuration Snippet

Finally, when the requester gets completions of all the requests, it calculates metrics such as request/message completion times and total goodput, and sends a completion notification to the responder through the TCP connection.

5.3.4 Event Injection

Lumina connects two traffic generation hosts to a high-performance programmable middlebox (event injector). To emulate realistic network scenarios like congestion and failures. the event injector can be programmed to inject packet corruptions, packet drops and ECN marks to RDMA data packets[2]. It is worthwhile to notice that the responder generates data packets for READ while the requester generates data packets for the other verbs.

While above events are not difficult to realize on reconfigurable network hardware, we find that the biggest challenge is to provide *user-friendly* interfaces for developers to express a series of *deterministic* injection events. This challenge is actually translated into two concrete requirements as follows.

[2]Currently Lumina does not support injecting events to control packets, e.g., ACK and NACK.

Deterministic: Since Lumina aims at precise and reproducible tests to understand micro-behaviors, it only accepts descriptions of deterministic injection events from the user. A description like "randomly drop 10% packets" is not deterministic as different rounds can drop different sequences of packets. In contrast, a description like "drop the first packet of the first QP" can generate deterministic injection behaviors.

User-friendly: Users should be able to express their high-level testing intents without the need to understand low-level details of Lumina. For example, the user should be able to tell the Lumina to drop the first packet of the second QP, then drop the retransmission of this packet. The user does not need to specify QPN and PSN for each QP, and understand how the event injector identifies the retransmitted packet.

Listing 5.2 gives a configuration example of event injections. There are three events across two QP connections. It is worthwhile to notice that Lumina only preserves the order of events on the same QP connection. The events of different QP connections are independent. On the first QP, we mark its fourth packet. On the second QP, we drop its fifth packet. When the sender retransmits this lost packet, we drop it again. Users just need to specify *relative* QPNs and PSNs, and can use *iter* field to express retransmission behaviors, thus to locate retransmitted packets which have same QPN and PSN as original packets.

Next, we will describe how Lumina translates high-level test intents (e.g., relative QPN and PSN) to the low-level configuration of the event injector, and uses an iteration number (ITER) to express per-connection retransmission behaviors.

Translate user intents to configurations: Since users only provide high-level intent information such as relative PSN and QPN, Lumina needs to translate this to the low-level configuration for the event injector. One straightforward solution is using the event injector to detect new QPs and parse their QPNs and initial PSNs on the data plane. While promising, this approach significantly complicates the data plane.

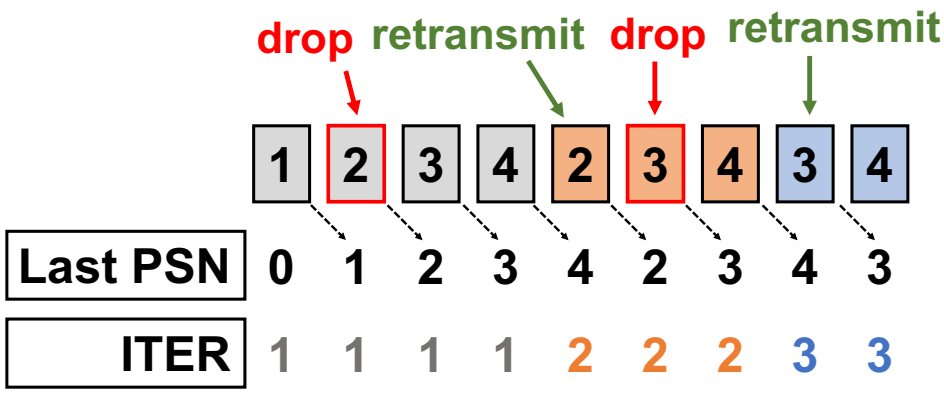


Figure 5-3. Lumina maintains ITER to differentiate packets in fine-grained. ITER denotes the rounds of (re)transmissions for a connection. If PSN of the current packet is no larger than that of the previous packet, ITER is increased by 1.

This is because, for every RDMA packet, the event injector first checks if it belongs to a new QP. If yes, the event injector further needs to initialize states for this new QP.

Instead of the above stateful approach, we take a stateless approach by leveraging traffic generators to provide runtime traffic metadata. As mentioned above (§5.3.3), traffic metadata like QPN and initial PSN (IPSN) is randomly generated at runtime. Once traffic generator instances finish exchanging metadata through TCP, the traffic requester sends the complete traffic metadata to the event injector through the control plane. The metadata is organized as a list of tuples. Each tuple contains the information for a certain QP connection: requester IP/QPN/IPSN and responder IP/QPN/IPSN. After that, the event injector combines the runtime traffic metadata from traffic generators and traffic configuration intents from the orchestrator to populate the match-action table for event injections. Only after the event injector populates the table, traffic generators can start RDMA traffic.

Figure 5-2 gives an example. The IP address, QPN, and IPSN of the requester's QP are 10.0.0.1, 0xfe, and 1001, respectively. The IP address, QPN, and IPSN of the responder's QP are 10.0.0.2, 0xea, and 3002, respectively. Data packets are sent from the requester to the responder. The user intends to drop the fourth packet of the first

QP connection. By combining above information, the event injector computes and inserts the following entry: if the source IP, destination IP, destination QPN, and PSN fields of a RoCEv2 packet are 10.0.0.1, 10.0.0.2, 0xea, and 1004 (1001 + 4 - 1), respectively, we should mark ECN for this packet.

Express retransmission behaviors: In many tests, the user needs to inject events to retransmitted packets to understand behaviors like retransmission timeout backoff. However, we cannot differentiate the retransmitted packet from the original packet by looking into the RoCEv2 packet header since they have the same IP addresses, UDP ports, QPN and PSN.

To realize this flexibility, we introduce an iteration number *ITER*, which denotes the rounds of (re)transmissions for a connection. ITER starts from 1 and is maintained by the event injector. For every arriving RDMA packet, the event injector compares its PSN with PSN of the last packet of its connection. If PSN of the current packet is *not larger than* that of the previous packet, the event injector identifies this as a new round of transmissions and increases ITER of this connection by 1. Regardless of the comparison result, the event injector always updates PSN of the last packet of the connection using PSN of the current packet. Lumina can use (PSN, ITER) to uniquely identify every packet in a connection. It is worthwhile to note that checking per-packet PSN and updating per-connection last PSN are done before event injections.

Figure 5-3 gives an example of how Lumina tracks ITER. In this example, there is only a single connection and the user intends to drop the second packet in the first round (PSN=2, ITER=1), and the third packet in the second round (PSN=3, ITER=2). The sender transmits four packets. ITER is initialized as 1 and the last PSN is set to IPSN-1, which is 0 in this case. In the first iteration, we drop packet 2. When packet 2 is retransmitted, current PSN (2) is smaller than the last PSN (4), thus triggering a new round of transmissions (ITER=2). Likewise, after we drop packet 3 in the second round, the retransmission of packet 3 triggers a new round

(ITER=3).

5.3.5 Traffic Dumping

Lumina aims to dump *all* the RDMA packets between traffic generators for offline analysis. A straightforward solution is using tools like `ibdump` to dump packets at the end host. However, it is unclear if traffic dumping at the end host will impact behaviors of network stacks. In addition, if we want to reconstruct the complete packet trace from packets dumped at both traffic generation hosts, we need to realize nanosecond-level clock synchronization, which is non-trivial [154].

Realizing these challenges, we adopt the event injector to mirror all the packets to a group of dedicated servers, which forms a traffic dumper pool. Packet mirroring essentially clones packets of specified interfaces and forward them to other interfaces for examination. It has been widely used for measurement and diagnosis purposes [155, 156]. We mirror all the RoCE packets at the ingress pipeline before actually dropping any packets in Memory Management Unit (more details in §5.5). We choose ingress mirroring instead of egress mirroring because we want to capture original behaviors of hardware network stacks.

For the ease of integrity check and traffic analysis, we leverage the event injector to embed some important metadata in mirrored packets. To avoid losses during packet dumping, we develop a per-packet load balancing mechanism to evenly distribute mirrored packets across CPU cores of traffic dumpers. We will describe them in detail.

Embedding metadata in mirrored packets: For the ease and efficiency of testing, the event injector embeds three types of metadata, *mirror sequence number*, *event type* and *mirror timestamp*, in mirrored packets for the following purposes.

1. **Integrity check.** As we use mirrored packets to understand behaviors of the hardware network stack, the first and most important step is to make sure we

mirror and dump all the packets. To this end, the event injector maintains a global variable, *mirror sequence number*, which is incremented for every arriving RDMA packet and embedded in each mirrored packet. Together with switch port counters, we can easily verify if there is any packet loss. If we dump all the packets, we should see consecutive mirror sequence numbers, and the largest mirror sequence number matches the total number of RX packets.

2. **Indicating events.** For the ease of analyzing mirrored packets, we embed an *event type* in each packet to indicate the injected event, currently including ECN marking, drop, and corruption, and none. Note that we mirror all the packets at the ingress pipeline before Memory Management Unit (MMU) executes dropping actions.

3. **Fine-grained measurement.** To accurately measure behaviors of the hardware offloaded network stack, we embed a *mirror timestamp* in each mirrored packet, which carries the nanosecond-level time when the original packet enters the ingress pipeline. Since the event injector adds timestamps to all the packets, it does not require clock synchronization.

To embed above metadata in mirrored packets, a straightforward solution is expanding packets with new fields storing these metadata. However, this may overload the bandwidth capacity of mirroring ports if original traffic's throughput is close to line rate. To avoid this, we rewrite existing header fields that are not involved in traffic analysis to store above metadata. We use the Time to Live (TTL) field, the source MAC address field, and the destination MAC address field, to store *event type*, *mirror sequence number*, and *mirror timestamp*, respectively.

Per-packet load balancing. In our initial design, we used two hosts to dump mirrored packets generated by the requester and the responder, respectively. Traffic dumping hosts had the same bandwidth capacity as traffic generation hosts. Despite

our optimization efforts on the traffic dumping program, we still occasionally observed few packets discards (`rx_discards_phy` in our testbed) on the NIC when receiving line-rate mirrored packets. Although we can identify such *invalid* tests through integrity checks (§5.3.6), this degrades the efficiency of Lumina. Beyond that, this design requires powerful traffic dumpers which have enough capacity, e.g., network bandwidth, memory bandwidth, and CPU, to dump packets sent by traffic generators at line rate. This degrades the flexibility of hardware choices of Lumina.

Realizing above limitations, we develop a per-packet load balancing mechanism to evenly distribute mirrored packets across CPU cores of all the traffic dumpers. Instead of using two powerful hosts, we organize several hosts as a traffic dumper pool. The user can flexibly set up hosts as long as the total capacity of the traffic dumper pool is enough, e.g., process bi-directional line-rate traffic with minimum-sized packets. As shown in Figure 5-4, we use a weighted round-robin scheduler at the event injector to forward mirrored packets to different traffic dumpers based on their processing capacity. Though the requester and responder generate traffic at heterogeneous rates, the event injector can evenly distribute mirrored packets to homogeneous traffic dumpers.

At each traffic dumping host, we leverage Receive Side Scaling (RSS) to distribute packets across multiple CPU cores. However, RSS preserves flow to CPU affinity by hashing certain packet fields to select a CPU core. As a result, the CPU processing capacity depends on the number of flows in the test. To fully exploit CPU cores, we use the event injector to rewrite the UDP destination port to a random number. Note that UDP destination port number 4791 is reserved for RoCEv2. By rewriting this well-known port number, we can create an illusion of many concurrent flows to RSS, thus maximizing CPU processing capacity.

After the traffic finishes, the orchestrator sends a TERM message to stop all the traffic dumpers. The traffic dumper catches the message, *recovers* the UDP destination port of all the packets, and writes packets to a disk file. We show the benefits of our

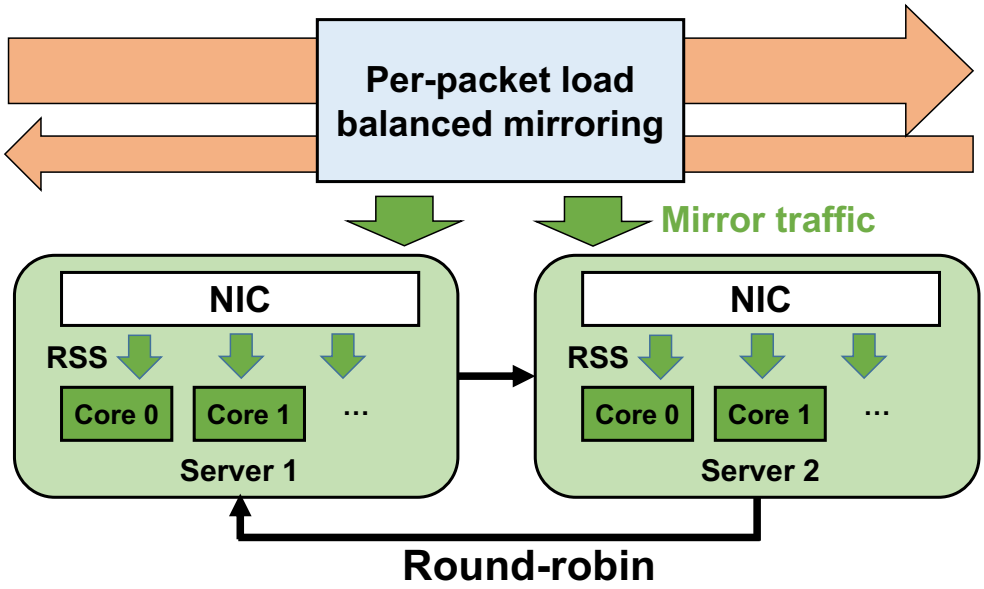


Figure 5-4. Per-packet load balancing to distribute mirrored packets across all the CPU cores of traffic dumpers.

Name	Content Description
Dumped packets	Packets collected by all the hosts of the traffic dumper pool
Network stack counters	Link/Network/Transport layer counters
Traffic generator log	Application level metrics, e.g., goodput and message completion time.
Switch counters	TX/RX/mirrored packet counters for each switch port

Table 5-I. Results collected by the orchestrator

traffic dumping design in §5.6.1.

5.3.6 Result Collection and Integrity Check

Once traffic generators stop, the orchestrator terminates all the other components and collects various result files given in Table 5-I. The orchestrator collects dumped packets from all the traffic dumpers, hardware network stack counters and traffic generator log files from traffic generation hosts, and switch counters from the event injector.

Upon collecting all the result files, the orchestrator reconstructs the packet trace

from packets collected by all the hosts of the traffic dumper pool. Since the event injector maintains the mirror sequence number and stores this on the source MAC address field of every mirrored packet (§5.3.5), the orchestrator simply sorts all the packets based on their mirror sequence numbers.

After the packet trace reconstruction, the orchestrator runs an *integrity check* using the following four conditions to determine if the packet trace is complete without losses of any packets during traffic mirroring and dumping:

1. Mirror sequence numbers in the trace are consecutive.
2. Mirror timestamps in the trace keep increasing unless the timestamp wraps around.
3. The number of packets in the trace equals the total number of packets mirrored by the event injector.
4. The number of packets in the trace equals the total number of RDMA packets received by the event injector.

Only if all the conditions hold simultaneously, we can ensure that Lumina reconstructs a *complete* packet trace. Otherwise, Lumina reports an "invalid test" error to stop users from running any analysis on this test.

5.4 Built-in Analyzers

Once tests pass integrity checks, users can parse reconstructed packet traces, log files, and counters to realize their customized requirements. For the ease of analysis in common cases, we provide a set of built-in analyzers for some widely used features in RDMA, e.g., Go-back-N retransmission [137] and ECN-based congestion control [157]. We use these analyzers in our later experiments (§5.6).

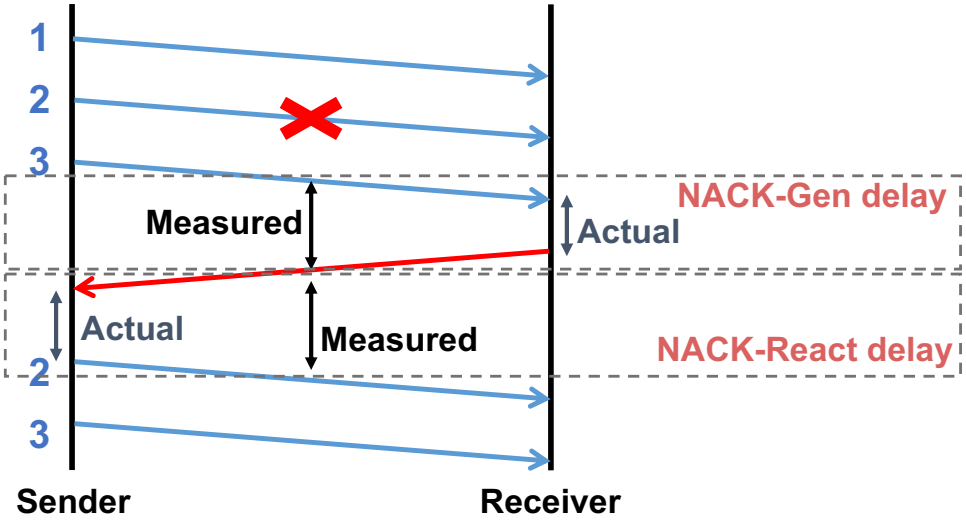


Figure 5-5. Retransmission latency breakdown.

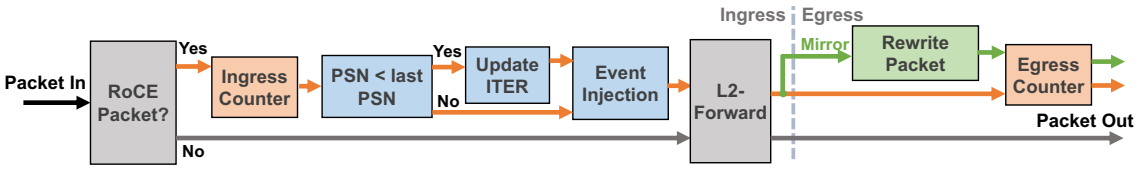


Figure 5-6. Pipeline layout of Lumina switch data plane.

Retransmission logic. Retransmission is crucial for reliable delivery. Even in *lossless* networks, RDMA NICs (RNICs) still need effective retransmission *mechanisms* to handle non-congestion losses [137], no to mention lossy networks without Priority-based Flow Control (PFC).

To this end, we develop a retransmission logic analyzer to check if the RNIC's behaviors under packet losses follow the specifications, e.g., if the Go-back-N receiver generates a NACK packet correctly when it observes out of order arriving packets. To realize this, we translate the specification of Go-back-N, the de facto retransmission algorithm of RNICs [137], into a finite-state machine (FSM) and feed the reconstructed packet trace into this FSM. If the packet trace leads to a wrong state, we can determine the RNIC's retransmission implementation does not fully comply with the specification.

Retransmission performance. Many efforts have been made to enable RDMA

over lossy networks [149, 158–160]. Lossy RDMA technologies heavily rely on efficient retransmission *implementations*. For example, when the RNIC receives a NACK/SACK packet, it should start the retransmission *immediately*, rather than wait for a long time.

To help users understand the retransmission performance of RNICs, we develop a retransmission performance analyzer. Note that this tool should be used in combination with the above retransmission logic analyzer to determine if the RNIC under test has a *correct* and *efficient* retransmission implementation. The retransmission performance analyzer can deal with both fast retransmissions (triggered by NACK/SACK) and timeout retransmissions (due to tail losses), and provide the performance breakdown to help users to identify the bottleneck.

Figure 5-5 shows how we break a NACK-triggered retransmission into two phases: NACK generation phase (receiver side) and NACK reaction phase (sender side). The NACK generation phase is the time between the receiver detects an out-of-order packet and it transmits a NACK. The NACK reaction phase is the time between the sender gets a NACK and it starts retransmission. We note that there is a half-RTT deviation since the timestamp is added by the middlebox rather than the end host. This deviation can be reduced by pre-measuring the RTT of the testbed.

Congestion notification. DCQCN [157] is the de facto RoCEv2 congestion control protocol implemented in Mellanox ConnectX-3 Pro and later RNICs. Once the DCQCN notification point (NP, receiver) receives ECN-marked packets, it notifies the reaction point (RP, sender) to reduce the rate using Congestion Notification Packets (CNPs). Recent Mellanox RNICs extend DCQCN to lossy networks. When the NP receives out-of-order packets, it generates *both* NACKs and CNPs to notify the RP to start the retransmission and lower the sending rate. In addition, to reduce the volume of CNP traffic and the CNP processing overhead, Mellanox RNICs also have a CNP pacer at the NP side, which determines the minimum interval between two

consecutive generated CNPs [157].

In summary, the generation of CNPs depends on ECN-marked packets, packet losses and the CNP pacer configuration. We develop a CNP analyzer to check if CNPs are generated as expected under various network conditions and CNP pacer configurations.

Hardware network stack counter. We also develop a counter analyzer to check if counters of the hardware network stack are updated correctly. Currently, we support counters related to retransmission, timeout, congestion and packet corruption: counters of sent/received packets, sequence errors, out-of-sequences, timeouts (and retry), packets with iCRC errors, discarded packets, CNPs sent/handled.

5.5 Implementation

We have built a prototype of Lumina using Tofino-based programmable switches and commodity servers equipped with NVIDIA Mellanox RNICs.

The data plane of the event injector is implemented with 668 lines of code (LoC) in P4-16 [161] and is compiled to Intel Tofino ASIC [19] using BF SDE 9.4.0. Figure 5-6 shows the data plane pipeline layout of the event injector. The event injection module modifies packets and sets the drop flag at the ingress pipeline (§5.3.4). The events are injected by manipulating the packet field or intrinsic metadata: packet drop is enabled by setting The RoCE packets are mirrored from ingress to egress. The egress pipeline includes a module to rewrite packet fields of mirrored packets (§5.3.5). We track both incoming and outgoing RoCE packets, including mirrored packets, on each port for integrity check (§5.3.6). The switch control plane is implemented with 815 LoC in Python, which translates RPC calls to configure the data plane modules and dumps port counters after the experiment finishes.

The traffic generator is implemented with 3116 LoC in C. It uses `Libibverbs` to

generate RDMA traffic over RC transport. The traffic generator controls the GID (IPv4 address) associated with each QP to emulate traffic from multiple hosts. It reports total goodput and average request/message completion times for each QP.

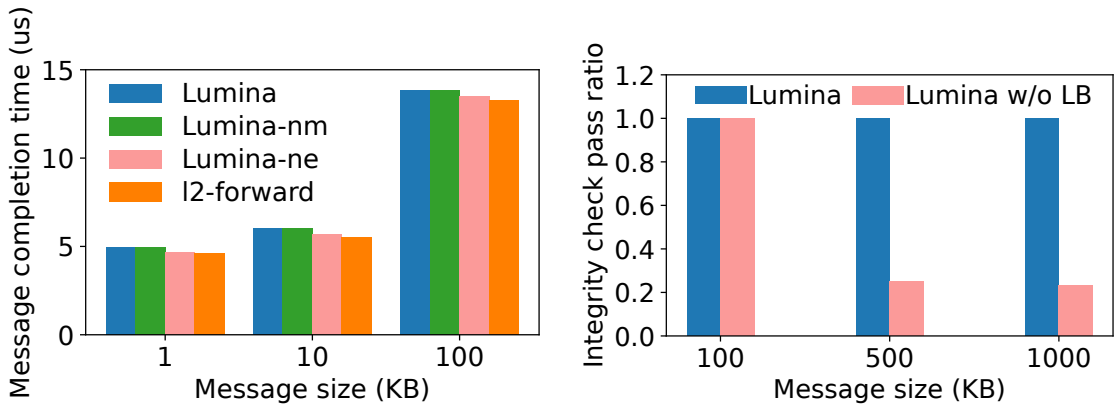
The packet dumper is implemented with 584 LoC in C. To maximize the performance and efficiency with multi-core processing, it uses DPDK [29] and Receive Side Scaling (RSS) to dispatch the packets among the RX queues and cores. It buffers packets in the pre-allocated memory during the experiment and writes them to the disk upon receiving the TERM message from the orchestrator.

The orchestrator is implemented with 1198 LoC in Python, and the built-in analyzers are implemented on top of it with 1726 LoC in Python.

5.6 Evaluation and Case Studies

We test three NVIDIA Mellanox RDMA NICs: ConnectX-4 Lx MCX4131A 40GbE, ConnectX-5 MCX515A 100GbE, and ConnectX-6 Dx MCX623105AN 100GbE. In the rest of this section, we refer them as CX4 CX5 and CX6, respectively. The experiments are conducted on three testbeds: namely *cx4-testbed*, *cx5-testbed*, and *cx6-testbed*. Each testbed has four servers connected to an Edgecore Wedge100BF-65X switch with Intel Tofino ASIC, which works as the event injector. Each server in *cx4-testbed* has an 8-core CPU (Intel Xeon E5-2450) and a CX4 NIC. Each server in *cx5-testbed* has a 16-core CPU (Intel Xeon Silver 4216) and a CX5 NIC. Each server in *cx6-testbed* has a 16-core CPU (AMD EPYC 7302) and a CX6 NIC. All the servers run Ubuntu 20.04.3 LTS and MLNX_OFED_LINUX-5.4-3.0.3.0. For each testbed, we use two hosts to generate traffic and the rest hosts to dump packets. The MTU is set to 1024B for all the experiments. We use WRITE verb by default.

In the rest of this section, we first use microbenchmark experiments to evaluate the basic performance of Lumina. (§5.6.1). Then we present some interesting results



(a) Overhead.

(b) Load balance.

Figure 5-7. Microbenchmark results of Lumina's overhead (left) and load balance scheme (right).

we find on the target devices using Lumina (§5.6.2, §5.6.3, and §5.6.4).

5.6.1 Microbenchmark

We conduct microbenchmark experiments for two purposes: (1) measure the overhead of event injections and mirroring on the data path, and (2) evaluate the benefit of per-packet load balancing in traffic dumping.

Overhead of event injection and mirroring. We find that Lumina adds little overhead to under-test traffic on the data path. In this experiment, the traffic generator keeps sending 1000 messages with fixed size over a single QP, and we measure the average message completion time (MCT). The messages are sent back-to-back and we run the experiment with different message sizes: 1KB, 10KB and 100KB.

We use a simple L2-Forwarding program as a baseline. For Lumina, we keep all the match-action tables but disable the exact "drop" behavior to avoid retransmissions. We also implement two variants of Lumina: Lumina without event injection (Lumina-ne) and Lumina without mirroring (Lumina-nm) for comparison. As shown in Figure 5.7(a), event injection introduces negligible overhead. The MCT of Lumina is only 4.1–7.2% higher than that of Lumina-ne and l2-forward. In the meantime, mirroring actually

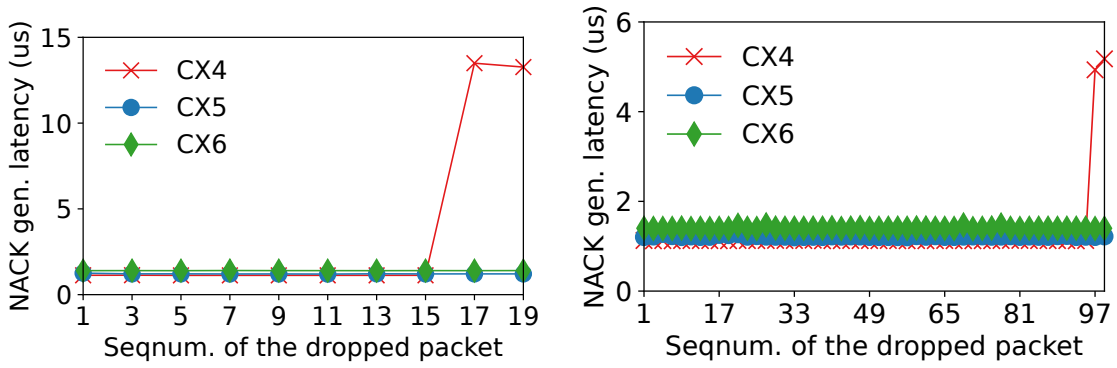
adds almost no overhead to the under-test traffic as Lumina delivers almost the same message completion time with or without mirroring.

Benefit of per-packet load balancing. The efficiency of Lumina relies on the reliability of packet dumping. In the experiment, we show how our per-packet load balancing can guarantee efficiency by minimizing during traffic mirroring and dumping phase. We send messages with different sizes (100KB, 500KB and 1MB respectively) at line rate and let the switch mirror all the RoCE packets. A tweaked version of Lumina without per-packet load balancing (mirror traffic from an ingress port to a specific server) is implemented as a comparison. We run the experiment for 100 rounds, and measure the ratio of rounds that pass the integrity check. Figure 5.7(b) shows that as the message gets larger, the pass ratio of the tweaked version is as low as 23%. However, with per-packet load balancing, Lumina can achieve 100% pass ratio by capturing all the packets.

5.6.2 Fast Retransmission

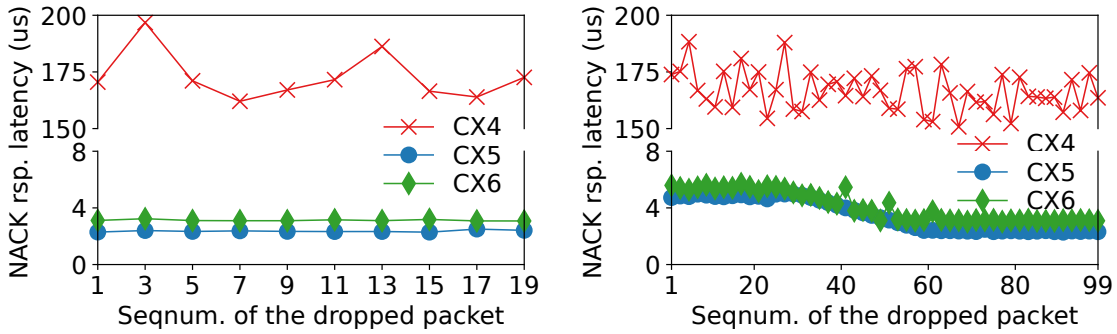
When packets in the middle are dropped, the receiver can observe out-of-order packets and generate NACK or SACK to trigger *fast* retransmissions. CX4, CX5 and CX6 adopt Go-back-N as the default fast retransmission algorithm. Here we use Lumina to evaluate RNICs' fast retransmission behaviors by deliberately dropping packets in the middle. First, we would like to note that all the RNICs pass our FSM-based retransmission logic check (§5.4) in a set of cunning and aggressive test cases. This indicates that their retransmission implementations strictly follow the specification. Then we present our findings about their fast retransmission performance.

Setting. In this experiment, the traffic generator uses one connection to generate WRITE traffic with only a single outstanding request. For each message, we drop one packet with a different (relative) PSN. We fix the message size as 20KB and 100KB respectively. The experiment runs 1000 iterations and we compute the average



(a) Message size = 20KB. (b) Message size = 100KB.

Figure 5-8. NACK generation latency v.s. sequence number of the dropped packet.

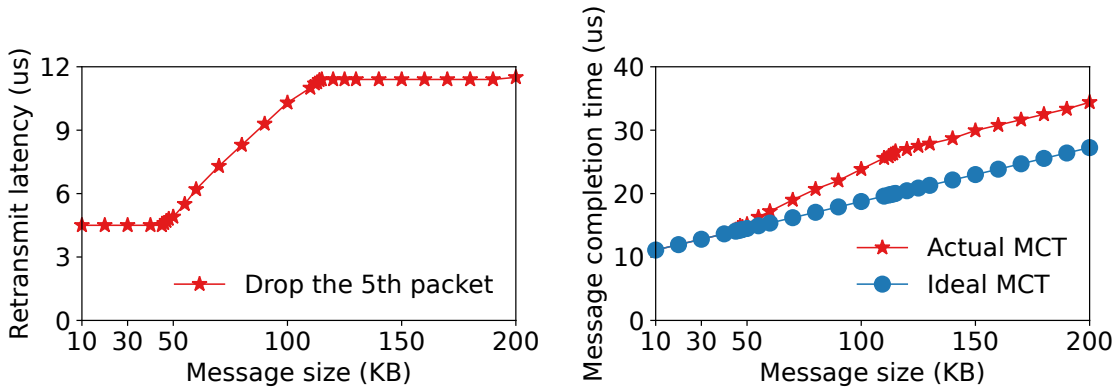


(a) Message size = 20KB. (b) Message size = 100KB.

Figure 5-9. NACK reaction latency v.s. sequence number of the dropped packet.

latency. As shown in Figure 5-5, we break the Go-back-N retransmission latency into two parts: the NACK generation latency, and the NACK reaction latency. We show NACK generation latency results in Figure 5-8 and NACK reaction latency results in Figure 5-9.

Performance improvement from CX4 to CX5 and CX6. From the results, we can clearly observe *significant* improvement on retransmission performance from CX4 to CX5 and CX6. As shown in Figure 5-8, if we drop one of the last several packets (e.g., the 19-th packet in Figure 5.8(a)) for CX4, the NACK generation latency (13–1μs) is much larger than the latency when we drop other packets (about 1.1μs). While for CX5 and CX6, the NACK generation latency statically stays around 1.1μs.



(a) Retransmission latency.

(b) Message completion time.

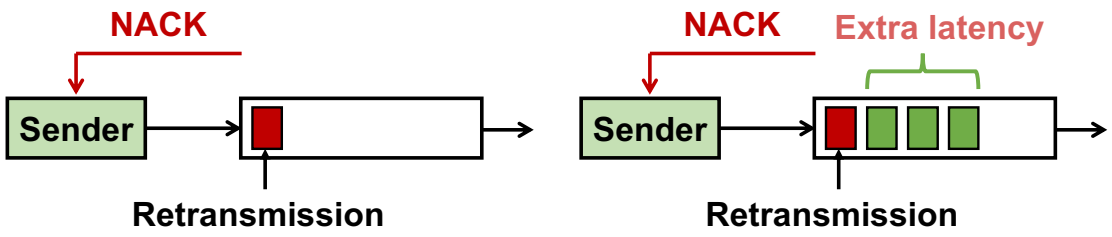
Figure 5-10. Effect of dropping the fifth packet for different message size on Mellanox CX6.

We can find things more interesting in Figure 5-9: the NACK reaction latency of CX4 (150–200μs) is much larger than that of CX5 and CX6 (3–6μs). Combine the two parts together, CX5 and CX6 have a huge improvement ($\sim$ 200μs v.s. $\sim$ 4.5μs) over CX4 in terms of retransmission performance.

Remark 1. Above observations confirm Mellanaox's significant efforts to enable lossy RDMA, e.g., move retransmission from firmware (CX4) to hardware (CX5 and later) [162].

Retransmission might be blocked. While CX5 and CX6 deliver low NACK reaction latency, the NACK reaction latency still varies. As shown in Figure 5.9(b), when the message size is 100KB, the NACK reaction latency is about 6μs if we drop one of the first 20 packets, while it is about 3μs if we drop a latter packet.

We conduct another set of experiments to further investigate this behavior. This time, we would like to analyze the effect of message size. To do this, we keep dropping the fifth packet of a message and vary the message size from 10KB to 200KB. The packets are sent back-to-back. While CX5 and CX6 have very similar behavior in this experiment, here we only show the results for CX6. Figure 5.10(a) plots the



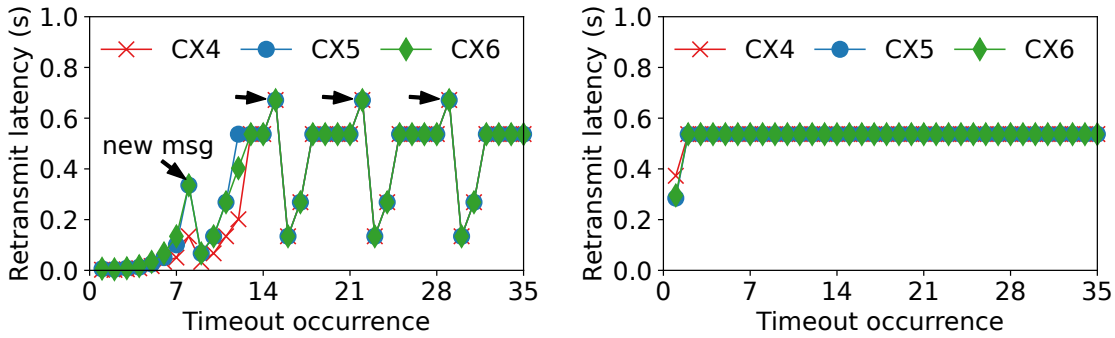
(a) Pipeline is empty when retransmission happens.

(b) Pipeline is not empty when retransmission happens.

Figure 5-11. Our hypothesis for NACK reaction behavior. The retransmitteed packets and the original packets share the same pipeline.

retransmission latency for different message sizes when we drop the fifth packet. The retransmission latency starts growing when the message size is around 50KB, and becomes constant after the message size is larger than 120KB. This trend reflects in the message completion time (MCT). In Figure 5.10(b), we plot the "Actual MCT" based on the output from traffic generator, and the "Ideal MCT" which is the sum of MCT without retransmission and a fixed "ideal" latency 4.5μs. When message size is between 50KB to 120KB, the "Actual MCT" grows faster than "Ideal MCT", which is consistent with Figure 5.10(a).

We have a hypothesis for this anomaly: retransmitted packets and original packets share the same transmission pipeline on the NIC, and retransmitted packets cannot *preempt* original packets that are already in the pipeline. As a result, retransmitted packets may be delayed. We illustrate this hypothesis with Figure 5-11. When the sender transmits a packet, it pushes the packet to the tail of the pipeline no matter whether it's a retransmitted packet or a normal packet. If the message is short (e.g., 10KB), the pipeline is already empty when the retransmission happens (Figure 5.11(a)) because all the packets have been transmitted. Otherwise, the pipeline might still retain a few packets that haven't been sent when a packet is going to be retransmitted (Figure 5.11(b)), if the message is relatively large (e.g., 100KB). Note that this guess may or may not be correct, but we believe Lumina helps users build a clearer profile



(a) Adaptive.

(b) Non-adaptive.

Figure 5-12. Retransmission latency v.s. Timeout occurrence. *timeout* is set as 14, *retry_cnt* is set as 7. We send 5 WRITE messages and keep dropping the last packet of each message for 7 times.

of what is going on inside the blackbox/hardware.

5.6.3 Timeout Retransmission

When tail packets or retransmitted packets are dropped, the sender can only use retransmission timer to recover them. Inappropriate timeout values can lead to either spurious retransmissions or poor tail performance. In this section, we report our findings related to timeout retransmissions.

Setting. When creating QPs, `Libibverbs` provides an interface to configure the *timeout* and *retry_cnt* value. We use the default values. *timeout* is set to 14, meaning that the *minimum* timeout is $4.096\mu s * 2^{timeout} = 0.0671s$ [163]. *retry_cnt* indicates the maximum number of times that the QP will try to resend the packets before reporting an error. We set it to 7.

In the experiment, we keep dropping the tail packet to trigger timeouts. Specifically, we use one connection to send 5 WRITE messages. For each message, the size is 10KB, and we drop the tenth (tail) packet for 7 (*retry_cnt*) times. We run the experiments in both adaptive [3] and non-adaptive mode (default) respectively. The results are

[3] Adaptive retransmission is a new feature in recent Mellanox RNICs to improve RDMA's resiliency over lossy networks.

Setting		Max. Retry of Msg1–Msg5				
Mode	NIC	1st	2nd	3rd	4th	5th
Adaptive	CX4	13	8	7	7	7
Adaptive	CX5	13	8	8	8	8
Adaptive	CX6	13	8	8	8	8
Non-adaptive	All	7	7	7	7	7

Table 5-II. Maximum retry times. *retry_cnt* is 7. We send 5 WRITE messages and keep dropping the last packet of each message until it reports an error.

shown in Figure 5-12.

Unexpected timeout value. When adaptive retransmission is enabled, the actual timeout value changes according to the packet loss frequency. It is worth noting that except for the first message, the first timeout of a message jumps to a high level unexpectedly (e.g., 0.267s for the second message and 0.671s for the latter messages, as pointed with black arrows in Figure 5.12(a)). Besides, the timeout value is not bounded by the pre-configured 0.0671s: for the first message, some of the retransmission timeouts are smaller than 0.0671s: 0.0056s, 0.0041s, 0.0084s, 0.0167s, 0.0251s, 0.0671s, and 0.1342s. For non-adaptive retransmission (Figure 5.12(b)), the timeout value obeys the specification: the first timeout is around 0.2–0.4s, then the following timeouts are static at about 0.537s. All these values are larger than the *minimum* timeout 0.0671s.

Retry times. We also find that the maximum retry times is correctly enforced on non-adaptive mode, but not on adaptive mode. We send 5 WRITE messages and keep dropping the last packet of each message until it reports an error. Table 5-II shows the results for every message and every NIC in both adaptive mode and non-adaptive mode. All the NICs work correctly under non-adaptive mode as it can retry for 7 times exactly as *retry_cnt* specifies. However, under adaptive mode, the first message can retry for 13 times and the later message can retry for 7 or 8 times. The maximum retry attempts also vary between different NIC models. For example, for the third

message, CX4 can retry 7 times, while CX5 and CX6 can retry 8 times. The result might make sense, since the timeout value is static in non-adaptive mode, so it is easier to ensure QoS by enforcing a fixed total retry count. While for adaptive mode, as the timeout is adaptively changing, it might be better to adjust the retry count according to other patterns (e.g., time spent for retransmission).

Remark 2. Adaptive retransmission is a new feature that has been rarely explored in the research community. We cannot find a public available specification that accurately explains its behaviors. Despite this, Lumina can still give us some viability into its micro-behaviors.

5.6.4 CNP Generation

In this section, we focus on CNP generation which is crucial for congestion control [157]. More specifically, we try to resolve the following doubt: "Will the CNP generation of one connection be affected by ECNs or losses from another connection".

Setting. We conduct this experiment using three connections, each sending a 1MB WRITE message. All the three connections have the same responder address (10.0.0.1) but different requester addresses (10.0.0.11, 10.0.0.12, 10.0.0.13) to simulate a scenario that three requesters are sending WRITE traffic to one responder (as shown in Figure 5-13). We denote the QPs at the responder side as QP1, QP2 and QP3 respectively. Mellanox NICs use a parameter named *min_time_between_cnps* to control the CNP generation interval. In this experiment, we set *min_time_between_cnps* to 50μs. The three connections are sending traffic simultaneously, and we mark ECN for the 50-th packet and the 950-th packet of each connection. We disable the DCQCN reaction functionality in the requester so that it won't adjust the sending rate upon receiving CNPs. After the experiment, we check how many CNPs does each (sender) QP receives. Table 5-III and Table 5-IV present the results.

NIC Model	Time of 1st ECN (μs)			Time of 2st ECN (μs)		
	QP1	QP2	QP3	QP1	QP2	QP3
CX4	10	31	45	616	630	643
CX5	8	17	18	246	255	258
CX6	9	16	17	246	254	259

Table 5-III. Timestamp of ECN-marked packets.

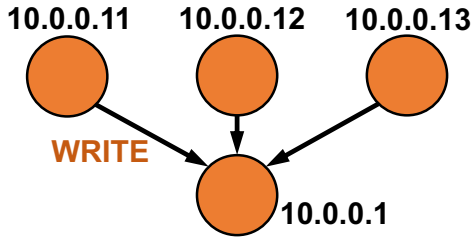


Figure 5-13. Simulated setting

NIC Model	Number of CNPs		
	QP1	QP2	QP3
CX4	2	2	2
CX5	2	2	2
CX6	2	0	0

Table 5-IV. CNP interval.

Per-port interval or per-dstIP interval. As shown in the Table 5-III, the first ECN-marked packet of each connection arrives at a relatively close time frame (within 50μs). Then after a long time period (about 600μs for *cx4-testbed*, 200μs for *cx5-testbed* and *cx6-testbed*), the second ECN-marked packet arrives. Table 5-IV shows how many CNPs each (receiver) QP sends out on each testbed. It's surprising to see that these NICs deliver different results. For CX6, only QP1 sends two CNPs while QP2 and QP3 do not reply to the ECN-marked packets. However, for CX4 and CX5, each of the three QPs sends two CNPs. According to our conversation with the vendor, there are at least two modes to enforce CNP intervals: per-port and per-destination IP. With per-port mode, all connections share a same CNP timer. While with per-destination IP mode, only the connections with the same destination IP share a same CNP timer. Different NICs may use different modes. It explains what we observe: CX6 enforces CNP interval with per-port mode, so that after a CNP is generated for an ECN-marked packet at time 9μs, the next CNP should be generated at least after 50+9=59μs. As a result, only QP1 sends two CNPs. While CX4 and CX5 use per-destination IP mode, the three QPs use separate timers and each can send two CNPs. While this is not necessarily a bug, it causes bias when users try to understand the NICs' behaviors.

We hope Lumina could mitigate such bias for users.

5.7 Discussion

Lossy RoCE. RoCEv2 originally relied on PFC to provide a lossless fabric. However, there has been a lot of discussion on lossy RoCE network [122, 149, 158]. While both industry and academia are taking the temperature of deploying lossy RDMA [122, 158], we believe that having a deep understanding of RDMA NICs' retransmission behavior and performance is essential. By anatomizing the retransmission process, Lumina can help us gain deep understanding about the micro-behaviors behind it.

Flexibility. We choose programmable switch as our middlebox implementation solution for Lumina because it provides the set of easy-to-use functionalities we need and it is accessible. However, our middlebox design is not restricted to programmable switch. The middlebox can be any programmable high-performance hardware or software. One of our future visions is to deploy Lumina with a FPGA board so that it is more light-weight and users can directly plug-and-test.

Extensibility. We do not intent to implement our traffic generator to cove all the traffic patterns for various application scenarios. Instead, we design Lumina in the extensible way. Users can customize their own traffic generators to test their own application-specific traffic patterns while no changes for other components are needed. Lumina is also extensible in terms of different transport protocols. While we start with RDMA as the target, Lumina can be extended with reasonable effort to test and measure other hardware network stacks.

Fuzzing and auto testcase generation. Lumina is suitable for integration with fuzzing or model-based testing as it provides well-structured input and output. It is also possible to leverage reinforcement learning to find potential issues by defining anomaly as rewards. We leave these as future work.

5.8 Related Work

Network protocol testing. There are many tools and research works focusing on testing network protocol implementation [126–129, 147]. Among them, packetdrill [126] is most related. Packetdrill is a scripting tool that enables tests for entire TCP/IP network stack. To interact with the local and remote network stack, packetdrill uses libpcap and TUN device as a "shim layer" to inject or consume packets. Packetdrill has also been utilized to test QUIC [127] and for educational purposes [164]. Similarly, Packet Shell [128] and Orchestra [129] also test the conformance of TCP implementation to its specification by injecting packets or events. DETER [147] focuses on deterministic TCP replay to reproduce performance problems and support tracing of TCP executions. Compared to them, Lumina focuses on hardware offloaded network stacks.

Performance anomaly and security of RDMA. With the wide adoption of RDMA in datacenters, the performance anomaly and security of RDMA have drawn a lot of attention. Collie [165] is a tool to systematically find RDMA performance anomalies caused by NIC resource contention. Kalia et al. [166] studied the scalability limits of RDMA: RDMA caches connection states in NICs which leads to scalability bottlenecks. Rothenberger et al. [167] demonstrated that the design and implementation of IB-capable NICs contain vulnerabilities and design flaws. Compared to them, Lumina focuses on transport protocol behaviors.

Network testing with programmable networks. Reconfigurable and programmable networks are becoming more relavant than ever before. In recent years, there are many works using smartNICs or programmable switches for applications acceleration [41, 168], network telemetry [49, 91], and achieve novel applications [169, 170].

Among them, Hypertester [169] and IMap [170] test the network environment by injecting packets into the network using switches. Hypertester applies programmable switches as network testers to generate and capture test traffic at line rate, and then use switch CPU to analyze the statistics. IMap implements a network scanner with programmable switches. It uses switch CPU to generate probe packets. The switch data plane is responsible for replicating the probe packets by recirculation. Lumina is the first work that leverages network programmability to test hardware network stacks, to the best of our knowledge.

5.9 Conclusion

We present Lumina, a tool to test the correctness and performance of hardware network stack implementation. We find several interesting micro-behaviors on RDMA NICs using Lumina. We believe Lumina can help network developers understand the micro-behaviors of complex hardware network stacks.

Chapter 6

Conclusion

6.1 Summary of Contributions

In this dissertation, we realize network programming and present multiple novel systems that enhance the performance, QoS, and reliability of today's data centers. We summarize our contributions as follows.

Algorithm, software, and hardware co-design. We propose efficient data structures and algorithms for data plane programs. The algorithms are carefully designed to adapt the network context. In NetLock, we design a memory allocation algorithm to maximize the use of switch resource. In HCSFQ, we extend CSFQ to HCSFQ to support hierarchical fair queueing. We conduct several approximation for the algorithms to make it deployable on the hardware. In AIFO, our programmable packet scheduling algorithm saves hardware resources—requires only one physical FIFO queue in the switch. All these algorithms have gone through attentive considerations and trade-offs. For the quantile computation module in AIFO, we have designed a theoretically stronger algorithm called QPipe [50]. Although QPipe provides stronger theoretical guaranttee, it occupies too many stages in the current hardware. So we eventually adopt the simpler sliding-window-basd solution to keep the quantile computation part concise and leave the space for other logics.

Resource-efficient co-design of switches and servers. To overcome the challenge

of limited switch functionalities and resources, we propose resource-efficient co-design of switches and servers. In NetLock, our memory management mechanism seamlessly integrates the switch and server memory. We only offload the popular locks to the switch and leave other locks to servers. In Lumina, we use a programmable switch as the middlebox to inject events to the under-test traffic and a couple of servers to capture the mirrored traffic for offline investigation.

System implementation and extensive evaluation. We build prototypes and conduct real-world experiments to evaluate our systems. In NetLock, we implement a centralized lock manager with an Intel Tofino switch and several DPDK servers. We run TPC-C workload and benchmark NetLock with state-of-the-art solutions. Lumina is implemented with a switch and four servers. Lumina's traffic generator supports various RDMA traffic and can be analyzed under different scenarios. We have found several bugs and performance issues of different RDMA interfaces with Lumina. Besides, we also perform simulations to evaluate our systems under scenarios beyond the scale of our testbeds. In HCSFQ and AIFO, we conduct large-scale simulation experiments under data center settings in addition to the rack-scale hardware experiments.

6.2 Future Works

Multi-rack resource management. NetLock and many other in-network applications focus on single-rack scale. In order to make the systems more scalable, cross-rack applications may be considered. Many problems should be considered to support multi-rack, including but not limited to: the memory sharing between multiple switches, consistency issue between all switches, how to diagnose and recover from failures. Further, both the industry and research community are exploring resource disaggregation in recent years. With resource disaggregation, the whole data center can be regarded as a huge computer with network as its backbone. Efficient and scalable in-network

resource management can bring tremendous benifit for disaggregated data center.

AI-informed network management. In AIFO and HCSFQ, we have to manually tune some parameters for better performance. Although we have theories to guide the parameter tuning and we can obtain a set of parameters that works for most of the cases, there are still cases where our system will fall short on the performance. Manually tuning parameters for every scenario is tedious and inefficient. By establishing an effective feedback loop, it is possible to compose a model to guide the parameter tuning. Besides, in Lumina's current practice, writing configurations requires some degree of networking expertise. Porting it with an automatic config generation module will definitely make our tool more friendly to users.

Reliable and unified network interfaces. Lumina can be further expanded to help analyze and test different smart network devices. The design of lumina does not necessarily requires a programmble switch. We can explore the possibility of replacing it with a high performance general DPDK host or a plug-and-use FPGA-based solution. On the other hand, we can go one step further: propose a unified abstraction for heterogeneous network interfaces with the help of tools like Lumina. Unified abstraction can accelerate hardware innovation and software evolution. Just like SoNiC and SAI (Switch Abstraction Interface) for switches, we believe host network interfaces also need such a unified abstraction.

6.3 Concluding Remarks

In this dissertation, we design and build several systems that empower various aspects of cloud data centers with network programming. We present: (a) NetLock for fast and centralized lock management; (b) HCSFQ for hierarchical fair queueing; (c) AIFO for programmable packet scheduling with a single queue; (d) Lumina for fine-grained analyzation of hardware network stacks. There have been arguments about what a

role should network play: should the switch perform various in-network applications or just simply do packet forwarding? While both sides hold rational points and design philosophy [1], we believe that as the programmable networking ecosystem gets mature and reliable with handy tools, programmable networks can empower computing systems in various aspects.

[1] Part of the reason why system research is appealing.

References

[1] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM CCR*, July 2014.

[2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," *SIGCOMM CCR*, vol. 38, April 2008.

[3] "Amazon Web Services," 2022. https://aws.amazon.com/.

[4] "Microsoft Azure," 2022. https://azure.microsoft.com/.

[5] "Google Cloud," 2022. https://cloud.google.com/.

[6] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro, "No compromises: Distributed transactions with consistency, availability, and performance," in *ACM SOSP*, October 2015.

[7] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen, "Fast in-memory transaction processing using RDMA and HTM," in *ACM SOSP*, October 2015.

[8] D. Y. Yoon, M. Chowdhury, and B. Mozafari, "Distributed lock management with RDMA: Decentralization without starvation," in *ACM SIGMOD*, June 2018.

[9] "Teradata: Business Analytics, Hybrid Cloud & Consulting," 2018. http://www.teradata.com/.

[10] A. B. Hastings, "Distributed lock management in a transaction processing environment," in *Symposium on Reliable Distributed Systems*, October 1990.

[11] C. Yan and A. Cheung, "Leveraging lock contention to improve oltp application performance," in *Proceedings of the VLDB Endowment*, September 2016.

[12] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea, "Deadlock immunity: Enabling systems to defend against deadlocks," in *USENIX OSDI*, December 2008.

[13] J. Huang, B. Mozafari, G. Schoenebeck, and T. F. Wenisch, "A top-down approach to achieving performance predictability in database systems," in *ACM SIGMOD*, May 2017.

[14] A. Devulapalli and P. Wyckoff, "Distributed queue-based locking using advanced network features," in *ICPP*, June 2005.

[15] S. Narravula, A. Marnidala, A. Vishnu, K. Vaidyanathan, and D. K. Panda, "High performance distributed lock management services using network-based remote atomic operations," in *IEEE CCGrid*, May 2007.

[16] D. Kim, Y. Zhu, C. Kim, J. Lee, and S. Seshan, "Generic external memory for switch data planes," in *ACM SIGCOMM HotNets Workshop*, November 2018.

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

[18] L. Lamport, "A new solution of Dijkstra's concurrent programming problem," *Communications of the ACM*, August 1974.

[19] "Intel Tofino," 2022. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html.

[20] "Broadcom Ethernet Switches and Switch Fabric Devices," 2022. https://www.

broadcom.com/products/ethernet-connectivity/switching.

[21] "Cavium XPliant," 2022. https://www.cavium.com/.

[22] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," in *ACM SIGCOMM*, August 2013.

[23] A. Kalia, M. Kaminsky, and D. G. Andersen, "FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs.," in *USENIX OSDI*, November 2016.

[24] X. Wei, Z. Dong, R. Chen, and H. Chen, "Deconstructing RDMA-enabled distributed transactions: Hybrid is better!," in *USENIX OSDI*, October 2018.

[25] S. Fitzgerald, I. Foster, C. Kesselman, G. Von Laszewski, W. Smith, and S. Tuecke, "A directory service for configuring high-performance distributed computations," in *IEEE HPDC*, August 1997.

[26] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free coordination for Internet-scale systems," in *USENIX ATC*, June 2010.

[27] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro, "No compromises: distributed transactions with consistency, availability, and performance," in *ACM SOSP*, pp. 54–70, ACM, 2015.

[28] C. Gray and D. Cheriton, "Leases: An efficient fault-tolerant mechanism for distributed file cache consistency," in *ACM SOSP*, December 1989.

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

[30] "CloudLab." https://www.cloudlab.us.

[31] "TPC-C." http://www.tpc.org/tpcc/.

[32] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi,

"Calvin: Fast distributed transactions for partitioned database systems," in *ACM SIGMOD*, May 2012.

[33] J. C. Corbett *et al.*, "Spanner: Google's globally distributed database," in *USENIX OSDI*, October 2012.

[34] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li, "Transaction chains: achieving serializability with low latency in geo-distributed storage systems," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 276–291, ACM, 2013.

[35] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li, "Extracting more concurrency from distributed transactions.," in *OSDI*, vol. 14, pp. 479–494, 2014.

[36] C. Xie, C. Su, M. Kapritsos, Y. Wang, N. Yaghmazadeh, L. Alvisi, and P. Mahajan, "Salt: Combining acid and base in a distributed database.," in *OSDI*, vol. 14, pp. 495–509, 2014.

[37] E. Zamanian, C. Binnig, T. Harris, and T. Kraska, "The end of a myth: Distributed transactions can scale," *Proceedings of the VLDB Endowment*, vol. 10, no. 6, pp. 685–696, 2017.

[38] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Coordination avoidance in database systems," in *Proceedings of the VLDB Endowment*, August 2015.

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

[40] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman, "Be fast, cheap and in control with SwitchKV," in *USENIX NSDI*, March 2016.

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

SOSP, October 2017.

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

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

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

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

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

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

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

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

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

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

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

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

[54] J. Nagle, "On packet switches with infinite storage," *IEEE Transactions on Communications*, April 1987.

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

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

[57] P. E. McKenney, "Stochastic fairness queueing.," in *IEEE INFOCOM*, June 1990.

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

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

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

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

[62] J. C. Bennett and H. Zhang, "Hierarchical packet fair queueing algorithms," in

ACM SIGCOMM, August 1996.

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

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

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

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

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

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

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

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

[71] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang, "SecondNet: A data center network virtualization architecture with bandwidth guarantees," in *ACM CoNEXT*, November 2010.

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

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

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

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

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

[77] A. Shieh, S. Kandula, A. G. Greenberg, C. Kim, and B. Saha, "Sharing the data center network," in *USENIX NSDI*, March 2011.

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

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

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

[81] M. Chowdhury, Z. Liu, A. Ghodsi, and I. Stoica, "HUG: Multi-resource fairness

for correlated and elastic demands," in *USENIX NSDI*, March 2016.

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

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

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

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

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

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

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

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

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

[91] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L. J. Wobker, "In-band network telemetry via programmable dataplanes," in *ACM SIGCOMM*, August 2015.

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

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

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

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

[96] L. E. Schrage and L. W. Miller, "The queue M/G/1 with the shortest remaining processing time discipline," *Operations Research*, vol. 14, no. 4, pp. 670–684, 1966.

[97] J. Y.-T. Leung, "A new algorithm for scheduling periodic, real-time tasks," *Algorithmica*, vol. 4, no. 1, pp. 209–219, 1989.

[98] H. Sariowan, R. L. Cruz, and G. C. Polyzos, "Sced: A generalized scheduling policy for guaranteeing quality-of-service," *IEEE/ACM Transactions on Networking*, vol. 7, no. 5, pp. 669–684, 1999.

[99] P. Goyal, H. M. Vin, and H. Cheng, "Start-time fair queueing: A scheduling algorithm for integrated services packet switching networks," *IEEE/ACM Transactions on Networking*, vol. 5, no. 5, pp. 690–704, 1997.

[100] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda, "Less is more: Trading a little bandwidth for ultra-low latency in the data center,"

in *USENIX NSDI*, April 2012.

[101] G. Kumar, N. Dukkipati, K. Jang, H. M. Wassel, X. Wu, B. Montazeri, Y. Wang, K. Springborn, C. Alfeld, M. Ryan, D. Wetherall, and A. Vahdat, "Swift: Delay is simple and effective for congestion control in the datacenter," in *ACM SIGCOMM*, August 2020.

[102] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *IEEE/ACM Transactions on Networking*, vol. 1, no. 4, pp. 397–1413, 1993.

[103] R. Pan, B. Prabhakar, and K. Psounis, "CHOKe: A stateless active queue management scheme for approximating fair bandwidth allocation," in *IEEE INFOCOM*, March 2000.

[104] R. Pan, L. Breslau, B. Prabhakar, and S. Shenker, "Approximate fairness through differential dropping," in *ACM SIGCOMM*, August 2003.

[105] K. Nichols and V. Jacobson, "Controlling queue delay: A modern aqm is just one piece of the solution to bufferbloat," in *ACM Queue*, 2012.

[106] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking, "Packet transactions: High-level programming for line-rate switches," in *ACM SIGCOMM*, August 2016.

[107] R. Mittal, V. T. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats, "Timely: Rtt-based congestion control for the datacenter," *SIGCOMM CCR*, August 2015.

[108] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center TCP (DCTCP)," in *ACM SIGCOMM*, August 2011.

[109] V. Shrivastav, "Fast, scalable, and programmable packet scheduler in hardware," in *ACM SIGCOMM*, August 2019.

[110] Z. Yu, J. Wu, V. Braverman, I. Stoica, and X. Jin, "Twenty years after: Hierarchical core-stateless fair queueing," in *USENIX NSDI*, April 2021.

[111] N. K. Sharma, C. Zhao, M. Liu, P. G. Kannan, C. Kim, A. Krishnamurthy, and A. Sivaraman, "Programmable calendar queues for high-speed packet scheduling," in *USENIX NSDI*, February 2020.

[112] A. Saeed, Y. Zhao, N. Dukkipati, E. Zegura, M. Ammar, K. Harras, and A. Vahdat, "Eiffel: Efficient and flexible software packet scheduling," in *USENIX NSDI*, February 2019.

[113] C.-Y. Hong, M. Caesar, and P. Godfrey, "Finishing flows quickly with preemptive scheduling," in *ACM SIGCOMM*, August 2012.

[114] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang, "Information-agnostic flow scheduling for commodity data centers," in *USENIX NSDI*, May 2015.

[115] S. Hu, W. Bai, G. Zeng, Z. Wang, B. Qiao, K. Chen, K. Tan, and Y. Wang, "Aeolus: A building block for proactive transport in datacenters," in *ACM SIGCOMM*, August 2020.

[116] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with Varys," in *ACM SIGCOMM*, August 2014.

[117] M. Chowdhury and I. Stoica, "Efficient coflow scheduling without prior knowledge," in *ACM SIGCOMM*, August 2015.

[118] F. J. Corbató, M. Merwin-Daggett, and R. C. Daley, "An experimental time-sharing system," in *Spring Joint Computer Conference*, pp. 335–344, 1962.

[119] I. A. Rai, G. Urvoy-Keller, and E. W. Biersack, "Analysis of LAS scheduling for job size distributions with high variance," *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 1, pp. 218–228, 2003.

[120] M. Nuyens and A. Wierman, "The Foreground–Background queue: A survey,"

Performance Evaluation, vol. 65, no. 3, pp. 286–307, 2008.

[121] L. Shalev, H. Ayoub, N. Bshara, and E. Sabbag, "A cloud-optimized transport protocol for elastic and scalable hpc," *IEEE/ACM MICRO*, 2020.

[122] Y. Gao, Q. Li, L. Tang, Y. Xi, P. Zhang, W. Peng, B. Li, Y. Wu, S. Liu, L. Yan, *et al.*, "When cloud storage meets rdma," in *USENIX NSDI*, 2016.

[123] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using RDMA efficiently for key-value services," in *ACM SIGCOMM*, August 2014.

[124] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo, "A unified architecture for accelerating distributed dnn training in heterogeneous gpu/cpu clusters," in *USENIX OSDI*, November 2020.

[125] J. Xue, Y. Miao, C. Chen, M. Wu, L. Zhang, and L. Zhou, "Fast distributed deep learning over rdma," in *EuroSys*, March 2019.

[126] N. Cardwell, Y. Cheng, L. Brakmo, M. Mathis, B. Raghavan, N. Dukkipati, H.-k. J. Chu, A. Terzis, and T. Herbert, "packetdrill: Scriptable network stack testing, from sockets to packets," in *USENIX ATC*, 2013.

[127] V. Goel, R. Paulo, and C. Paasch, "Testing quic with packetdrill," in *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*, 2020.

[128] S. Parker and C. Schmechel, "The packet shell protocol testing tool," *Software distribution at http://playground.sun.com/psh*, 1996.

[129] S. Dawson, F. Jahanian, and T. Mitton, "Experiments on six commercial tcp implementations using a software fault injection tool," *Software: Practice and Experience*, 1997.

[130] J. S. Chase, A. J. Gallatin, and K. G. Yocum, "End system optimizations for high-speed tcp," *IEEE Communications Magazine*, vol. 39, no. 4, pp. 68–74,

2001.

[131] A. Menon and W. Zwaenepoel, "Optimizing tcp receive performance," in *USENIX ATC*, 2008.

[132] "Segmentation Offloads," 2022. https://www.kernel.org/doc/html/latest/networking/segmentation-offloads.html.

[133] "Chelsio Terminator 5 ASIC," 2022. https://www.chelsio.com/terminator-5-asic/.

[134] "Broadcom NetXtreme Ethernet Adapters," 2022. https://www.broadcom.com/how-to-buy/hardware-partners/ethernet-network-adapters/broadcom.

[135] T. S. Woodall, G. M. Shipman, G. Bosilca, R. L. Graham, and A. B. Maccabe, "High performance rdma protocols in hpc," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, Springer, 2006.

[136] G. F. Pfister, "An introduction to the infiniband architecture," *High performance mass storage and parallel I/O*, 2001.

[137] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn, "Rdma over commodity ethernet at scale," in *ACM SIGCOMM*, 2016.

[138] "NVIDIA ConnectX-6 Dx," 2022. https://www.nvidia.com/en-us/networking/ethernet/connectx-6-dx/.

[139] "Intel Ethernet Network Adapter E810," 2022. https://www.intel.com/content/www/us/en/products/details/ethernet/800-network-adapters/e810-network-adapters.html.

[140] "Broadcom M1100G16 100GbE OCP 2.0 Adapter," 2022. https://www.broadcom.com/products/ethernet-connectivity/network-adapters/100gb-nic-ocp/m1100g.

[141] "Marvell FastLinQ 41000 Series Ethernet NICs," 2022. https://www.marvell.com/products/ethernet-adapters-and-controllers/41000-ethernet-adapters.html.

[142] "OneConnect OCe14000-Series Adapters," 2022. https://docs.broadcom.com/doc/12356182.

[143] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia, "A remote direct memory access protocol specification," tech. rep., RFC 5040, October 2007.

[144] V. Cerf, S. Burleigh, A. Hooke, L. Torgerson, R. Durst, K. Scott, K. Fall, and H. Weiss, "Delay-tolerant networking architecture," 2007.

[145] V. Paxson, "Automated packet trace analysis of tcp implementations," in *ACM SIGCOMM*, 1997.

[146] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough, "Rigorous specification and conformance testing techniques for network protocols, as applied to tcp, udp, and sockets," in *ACM SIGCOMM*, 2005.

[147] Y. Li, R. Miao, M. Alizadeh, and M. Yu, "Deter: Deterministic tcp replay for performance diagnosis," in *USENIX NSDI*, February 2019.

[148] "OFED perftest," 2022. https://github.com/linux-rdma/perftest.

[149] A. Shpiner, E. Zahavi, O. Dahley, A. Barnea, R. Damsker, G. Yekelis, M. Zus, E. Kuta, and D. Baram, "Roce rocks without pfc: Detailed evaluation," in *Proceedings of the Workshop on Kernel-Bypass Networks*, 2017.

[150] D. Mudigere, Y. Hao, J. Huang, Z. Jia, A. Tulloch, S. Sridharan, X. Liu, M. Ozdal, J. Nie, J. Park, *et al.*, "Software-hardware co-design for fast and scalable training of deep learning recommendation models," *arXiv preprint arXiv:2104.05158*, 2021.

[151] "Intel Tofino 2," 2022. https://www.intel.com/content/www/us/en/

products/network-io/programmable-ethernet-switch/tofino-2-series.html.

[152] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, *et al.*, "Azure accelerated networking:smartnics in the public cloud," in *USENIX NSDI*, 2018.

[153] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore, "Netfpga sume: Toward 100 gbps as research commodity," *IEEE/ACM MICRO*, 2014.

[154] Y. Li, G. Kumar, H. Hariharan, H. Wassel, P. Hochschild, D. Platt, S. Sabato, M. Yu, N. Dukkipati, P. Chandra, *et al.*, "Sundial: Fault-tolerant clock synchronization for datacenters," in *USENIX OSDI*, November 2020.

[155] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, *et al.*, "Packet-level telemetry in large datacenter networks," in *ACM SIGCOMM*, August 2015.

[156] J. Rasley, B. Stephens, C. Dixon, E. Rozner, W. Felter, K. Agarwal, J. Carter, and R. Fonseca, "Planck: Millisecond-scale monitoring and control for commodity networks," *SIGCOMM CCR*, August 2014.

[157] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang, "Congestion control for large-scale rdma deployments," *SIGCOMM CCR*, August 2015.

[158] R. Mittal, A. Shpiner, A. Panda, E. Zahavi, A. Krishnamurthy, S. Ratnasamy, and S. Shenker, "Revisiting network support for rdma," in *ACM SIGCOMM*, 2018.

[159] Y. Lu, G. Chen, Z. Ruan, W. Xiao, B. Li, J. Zhang, Y. Xiong, P. Cheng, and E. Chen, "Memory efficient loss recovery for hardware-based transport in datacenter," in *Asia-Pacific Workshop on Networking*, August 2017.

[160] "NVIDIA Zero Touch RoCE (ZTR)," 2021. https://tinyurl.com/yc6tnv7h/.

[161] "P4-16 Language Specification," 2021. https://p4.org/p4-spec/docs/P4-16-v1.2.2.html.

[162] "Hardware Offloading To RDMA and Beyond." https://conferences.sigcomm.org/sigcomm/2018/files/slides/kbnet/keynote_2.pdf.

[163] "InfiniBand Architecture Specification Volume 1 Release 1.5," 2021. https://www.infinibandta.org/ibta-specification/.

[164] O. Bonaventure, Q. De Coninck, F. Duchêne, A. Gego, M. Jadin, F. Michel, M. Piraux, C. Poncin, and O. Tilmans, "Open educational resources for computer networking," *SIGCOMM CCR*, August 2020.

[165] X. Kong, Y. Zhu, H. Zhou, Z. Jiang, J. Ye, C. Guo, and D. Zhuo, "Collie: Finding performance anomalies in rdma subsystems," in *USENIX NSDI*, April 2022.

[166] A. Kalia, M. Kaminsky, and D. Andersen, "Datacenter rpcs can be general and fast," in *USENIX NSDI*, February 2019.

[167] B. Rothenberger, K. Taranov, A. Perrig, and T. Hoefler, "Redmark: Bypassing rdma security mechanisms," in *USENIX Security*, August 2021.

[168] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang, "KV-Direct: High-performance in-memory key-value store with programmable NIC," in *ACM SOSP*, October 2017.

[169] D. Zhang, Y. Zhou, Z. Xi, Y. Wang, M. Xu, and J. Wu, "Hypertester: high-performance network testing driven by programmable switches," *IEEE/ACM Transactions on Networking*, 2021.

[170] G. Li, M. Zhang, C. Guo, H. Bao, M. Xu, H. Hu, and F. Li, "Imap: Fast and scalable in-network scanning with programmable switches," in *USENIX NSDI*, April 2022.