In-Network Congestion Management for Security and Performance

PhD Thesis by Albert Gran Alcoz

Diss. ETH No. 30290

DISS. ETH NO. 30290

IN-NETWORK CONGESTION MANAGEMENT FOR SECURITY AND PERFORMANCE

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES
(Dr. sc. ETH Zürich)

presented by

ALBERT GRAN ALCOZ
MSc Telecommunications Engineering
ETSETB UPC
born on 19.11.1994

accepted on the recommendation of

Prof. Dr. Laurent Vanbever
Prof. Dr. Nick McKeown
Prof. Dr. Mohammad Alizadeh

2024

Diss. ETH No. 30290
TIK-Schriftenreihe-Nr. 214

ABSTRACT

It was during the early days of the ARPANET that researchers first realized the crucial role that congestion would play in the Internet's performance. Since then, numerous scholars have dedicated themselves to developing a variety of algorithms to proactively manage it. Today, 50 years later, the Internet has undergone significant evolution. Yet, network congestion remains one of the biggest open challenges in current Internet design.

In this dissertation, we propose techniques aimed at managing network congestion while enhancing the performance and security of the Internet. Our approach is grounded in data-plane programmability—a recent technological paradigm in the networking field that has fundamentally reshaped how we design and reason about network architectures. Additionally, we address state-of-the-art congestion types, such as pulse-wave denial-of-service (DoS) attacks, which pose a growing threat to existing infrastructures.

First, we introduce SP-PIFO and PACKS, two frameworks that enable programmable in-network congestion management on existing routers. Operators assign ranks to packets to indicate how they should be prioritized during congestion. SP-PIFO and PACKS then admit and schedule packets based on these ranks. To run on existing devices, they build on a set of priority queues and decide which packets to admit and how to map admitted packets to the different queues. SP-PIFO operates on a per-packet basis, while PACKS enhances SP-PIFO's performance by incorporating rank-distribution information and queue-occupancy levels during enqueue.

Next, we present QVISOR, a hypervisor that extends SP-PIFO and PACKS to support multi-tenancy, allowing different tenants to specify their own priorities while sharing a common set of hardware resources. Within QVISOR, tenants define their traffic prioritization preferences, while the operator determines how the resources should be allocated. QVISOR then synthesizes a joint scheduling strategy and implements it on the underlying hardware.

Finally, we introduce ACC-Turbo, a pulse-wave denial-of-service defense that demonstrates the advantages of in-network congestion management in the context of security. ACC-Turbo detects attacks at line rate and in real time by applying online clustering techniques in the network and mitigates them on a per-packet basis using programmable packet scheduling.

ZUSAMMENFASSUNG

In den Anfängen des ARPANET erkannten die Forscher zum ersten Mal, welch entscheidende Rolle Überlastungen für die Leistung des Internets spielen würden. Seitdem haben sich zahlreiche Wissenschaftler der Entwicklung von Techniken und Algorithmen gewidmet, um Überlastungen proaktiv zu bewältigen. Heute, 50 Jahre später, hat sich das Internet erheblich weiterentwickelt. Dennoch ist Netzwerküberlastung nach wie vor eine der größten offenen Herausforderungen bei der Gestaltung des Internets.

In dieser Dissertation schlagen wir Techniken vor, die darauf abzielen, Netzüberlastungen zu bewältigen und gleichzeitig die Leistung und Sicherheit des modernen Internets zu verbessern.

Zunächst stellen wir SP-PIFO und PACKS vor, zwei Frameworks, die ein programmierbares netzinternes Staumanagement auf bestehenden Routern ermöglichen. Betreiber weisen Paketen Ränge zu, um anzugeben, wie diese im Falle einer Überlastung priorisiert werden sollen. SP-PIFO und PACKS akzeptieren und ordnen dann Pakete auf der Grundlage dieser Ränge. Um auf bestehenden Geräten zu laufen, bauen sie auf Prioritätswarteschlangen auf und entscheiden, welche Pakete zugelassen werden und wie die zugelassenen Pakete den verschiedenen Warteschlangen zugeordnet werden. SP-PIFO entscheidet auf Basis einzelner Pakete, während PACKS Informationen über die Rangverteilung und Belegung der Warteschlangen einbezieht, um die Leistung von SP-PIFO zu verbessern.

Als Nächstes stellen wir QVISOR vor, einen Hypervisor, der SP-PIFO und PACKS erweitert, um Multi-Tenancy zu unterstützen. Innerhalb von QVISOR legen die Nutzer ihre Präferenzen für die Priorisierung des Datenverkehrs fest, während QVISOR eine kombinierte Strategie erstellt und diese auf der zugrunde liegenden Hardware implementiert.

Schließlich stellen wir ACC-Turbo vor, eine neuartige Pulswellen-Denial-of-Service-Verteidigung, die die Vorteile des netzwerkbasierten Überlastungsmanagements im Kontext der Sicherheit hervorhebt. ACC-Turbo erkennt Angriffe bei Leitungsrate und in Echtzeit durch die Anwendung fortschrittlicher Online-Clustering-Techniken im Netzwerk und entschärft sie auf paketweiser Basis durch eine programmierbare Paketeinteilung.

PUBLICATIONS

This dissertation is based on previously published conference proceedings. The list of accepted and submitted publications is presented hereafter.

SP-PIFO: Approximating Push-In First-Out Behaviors using Strict-Priority Queues

Albert Gran Alcoz, Alexander Dietmüller, Laurent Vanbever

USENIX NSDI, Santa Clara, CA, USA, 2020.

Aggregate-Based Congestion Control for Pulse-Wave DDoS Defense

Albert Gran Alcoz, Martin Strohmeier, Vincent Lenders, Laurent Vanbever

ACM SIGCOMM, Amsterdam, Netherlands, 2022.

QVISOR: Virtualizing Packet Scheduling Policies

Albert Gran Alcoz, Laurent Vanbever

ACM HotNets, Cambridge, MA, USA, 2023.

Everything Matters in Programmable Packet Scheduling

Albert Gran Alcoz, Balázs Vass, Pooria Namyar, Behnaz Arzani, Gábor Rétvári, Laurent Vanbever

USENIX NSDI, Philadelphia, PA, USA 2025.

The following publications were part of my PhD research and are referenced in this thesis, but they were led by other researchers.

Principles for Internet Congestion Management

Lloyd Brown, Albert Gran Alcoz, Frank Cangialosi, Akshay Narayan, Mohammad Alizadeh, Hari Balakrishnan, Eric Friedman, Ethan Katz-Bassett, Arvind Krishnamurthy, Michael Schapira, Scott Shenker

ACM SIGCOMM, Sydney, Australia, 2024.

The following publications were part of my PhD research, but are not covered in this dissertation.

Reducing P4 Language's Voluminosity using Higher-Level Constructs

Albert Gran Alcoz, Coralie Busse-Grawitz, Eric Marty, Laurent Vanbever

ACM EuroP4, Rome, Italy, 2022.

FitNets: An Adaptive Framework to Learn Accurate Traffic Distributions

Alexander Dietmüller, Albert Gran Alcoz, Laurent Vanbever

arXiv preprint, arXiv:2405.10931, 2024.

Inter-Cloud QoS with FlexEgress

Sarah McClure, Albert Gran Alcoz, Laurent Vanbever, Sylvia Ratnasamy, Scott Shenker

Under submission, 2024.

Spoof-Shield: Mitigating IP Spoofing Attacks at the Internet's Edge

Vasileios Giotsas, Albert Gran Alcoz, Lucas Castanheira, Theophilus Benson, Georgios Smaragdakis, Marwan Fayed

Under submission, 2024.

ACKNOWLEDGMENTS

I am immensely grateful to the broad set of brilliant individuals who have contributed to shaping this PhD journey. Their support, guidance, and encouragement have been invaluable. I am deeply indebted to all of them.

First and foremost, I would like to thank Professor Laurent Vanbever for his mentorship during these years. From my Master's thesis, to the internship, to the PhD, I always felt that he believed in me, trusted my work, and granted me freedom to pursue my interests. Everything I know about research today is thanks to him. He helped me "push" for the various papers, opened the doors to his contact network, and offered me counsel when I needed it the most. Laurent has been the best advisor I could have asked for. He has changed my life for good, and I will be forever grateful.

I would like to extend my gratitude to all the members of the Networked Systems Group at ETH Zürich. They have enriched this journey with enlightening discussions, fun retreats, engaging activities and memorable trips. Being part of the NSG family has been an honor, and I hope that the relationship we have forged these years endures over time.

I sincerely thank Professors Nick McKeown and Mohammad Alizadeh for serving as co-examiners for this thesis. All their comments have contributed positively to this dissertation. I extend the gratitude to Nick for his warm reception during my visit at Stanford and Barefoot Networks in 2020.

I am also thankful to my second advisor, Professor Kaveh Razavi, for his unwavering support and encouragement throughout these years.

During the first years of my doctoral journey, I was privileged to receive support from Armasuisse and guidance from Vincent Lenders and Martin Strohmeier. This support enabled me to pursue this PhD. I am indebted to them and I hope that some day I can reciprocate their generosity.

I am immensely thankful to Professors Sylvia Ratnasamy and Scott Shenker for hosting me during the summer of 2023. I have long admired their contributions to our field and working alongside them has been a dream come true. I extend my appreciation to the Berkeley NetSys Lab for welcoming me with open arms. Witnessing their groundbreaking work

firsthand was truly inspiring, and I consider myself incredibly fortunate to have had the opportunity to be a part of their research community.

I would like to thank Marwan Fayed for giving me the chance to expand my industry experience at Cloudflare. He not only recognized the value of my work but also granted me complete flexibility to make the most of my internship. I sincerely appreciate his advice on navigating the next steps of my career and the wealth of knowledge he generously shared. I extend this gratitude to the Cloudflare Research team and to the Cloudflarians across the Infrastructure, DoS, Networking, and Protocols teams with whom I had the pleasure of interacting. Collaborating with them was both enjoyable and insightful. I owe a special thank to the Munich office for creating an unforgettable experience, both within and outside the workplace. I hope that we can continue 'making the Internet better' for many years to come.

I deeply appreciate Alexander Dietmüller, Balázs Vass, Gábor Rétvári, Pooria Namyar and Behnaz Arzani for their priceless contributions to our efforts in making scheduling programmable and for ensuring mathematical rigor in our papers. I extend this gratitude to all the researchers with whom I have had the privilege to collaborate and all the students I have had the pleasure of supervising. Their work has significantly enriched this thesis.

I am very thankful to Beat Futterknecht for his exceptional administrative assistance, which has been an invaluable support throughout my journey. His expertise, and meticulous attention to detail have been a lifesaver amidst the bureaucratic challenges of this international adventure.

My heartfelt appreciation also goes to all my friends in Zurich, Munich, Berkeley, and San Francisco, who have each made those places feel like home. Additionally, I am thankful to my friends in Barcelona for preserving our long-lasting friendships despite the distance that separates us.

Lastly, but most importantly, I am deeply grateful to my family for their inconditional love and support throughout this journey.

A vosaltres, papis. Per tot, sempre.
No hi ha paraules que expressin tot el que us estimo.

Albert Gran Alcoz
June 2024

CONTENTS

1

INTRODUCTION

October 1986. Berkeley, California: The ARPANET, the first version of the Internet, has been operational for 20 years. It connects approximately 80 computers across campuses in the United States of America and Europe [10]. The new Transmission Control Procotol (TCP) has been incorporated into the Berkeley Unix operating system, BSD 4, and the network is experiencing significant growth [11]. It would still take another two years until the World Wide Web is invented, marking the Internet's transition to mainstream use.

At the Lawrence Berkeley Lab, researchers identify a significant drop in data throughput on the link connecting the lab with the University of California, Berkeley. Despite being a connection of just 400 meters and two hops, the bandwidth drop is on the order of a thousand: from 32 Kbps to 40 bps. John Nagle had already anticipated it two years earlier [12]. They are experiencing the first-ever *congestion collapse* in the Internet's history [13].

Over the last decades, the Internet has become the largest system ever built, connecting more than 20 billion devices and playing a big role in the world's global economy [14]. Despite its radical evolution, some challenges from its early days remain unchanged. One of them stands out: *congestion*.

Congestion is an inherent condition in the Internet's design that stems from an imbalance between the capacity of a given link and its traffic demands. It occurs when multiple data streams compete for the limited bandwidth of the link, generating more packets than it can actually process. As the link becomes oversubscribed, excess packets are temporarily buffered, leading to increased packet latency. If the situation persists, the router's buffer space eventually becomes exhausted, resulting in packet drops.

In its most basic form, congestion occurs due to three main factors: (i) the limited resources of the network infrastructure or the need for it to run at high efficiency, (ii) the distributed nature of the Internet architecture, and (iii) the bursty nature of its transmissions. Indeed, to achieve a reasonable efficiency, the Internet relies on packet-level statistical multiplexing, allowing multiple connections to share the same resources. This allows

networks to operate at high utilization, but induces the risk of congestion. At the same time, end-hosts act autonomously and independently of each other. Often, they require access to the same network resources, having to compete for them, which also leads to contention. Finally, even when a single end-host is active, its transmission patterns tend to be bursty due to the mismatch between application-level message sizes and network-level packet sizes, which may itself overload the resources of forwarding devices.

Since the first congestion collapse in 1986, researchers have been aware of the negative implications of congestion on the Internet's performance. As a result, they have proposed a broad set of techniques to proactively manage it. These techniques can be classified into two main categories: *end-to-end control* mechanisms and *in-network congestion management* techniques.

End-to-end control mechanisms notify end-hosts of network congestion, allowing them to adjust their sending behavior to prevent congestion from worsening. Examples include *congestion control*, which dynamically adjusts the number of in-flight packets a sender can maintain based on observed network conditions such as latency or packet loss; *flow control*, which adapts the sender's transmission rate based on the receiver's processing capacity; and *active queue management*, which uses network logic to explicitly notify end-hosts of congestion or enhance the notification signals (e.g., through proactive packet dropping or tagging), accelerating their response times.

While these techniques play a role in preventing long-term congestion, they have two significant limitations. First, their feedback loop lacks the speed required to address congestion effects in the short term (e.g., within milliseconds) or during transient congestion events like microbursts. Second, they rely on end-hosts' willingness to adjust their sending behavior upon notification. While this may hold in specific scenarios (e.g., within a private network), it is not always the case on today's Internet.

Despite numerous efforts by the networking community to encourage the adoption of "friendly" congestion-control algorithms, major Internet players still rely on aggressive congestion-control algorithms to enhance their performance in significant portions of their traffic [5]. Even more concerning, malicious actors have discovered that they can exploit limited bandwidth resources at specific network bottlenecks, effectively weaponizing congestion to disrupt the operations of other entities. These attack vectors, known as denial-of-service attacks, occur thousands of times daily and occasionally manage to disrupt the connectivity of entire countries [15].

In-network congestion management techniques aim to overcome the limitations of end-to-end controls. They do so by running directly in the network (i.e., within a router), processing traffic at line-rate, and quickly making local decisions to mitigate the impact of congestion as soon as it emerges. These techniques are typically responsible for three key decisions: *admission control*, determining which packets to accept into the buffer; *buffer management*, deciding how to allocate buffer space among admitted packets; and *scheduling*, selecting which packets to forward next from those in the buffer.

Extensive research over decades has also focused on these techniques, resulting in the proposal of numerous algorithms. Unlike end-to-end control mechanisms, the primary barrier hindering the widespread adoption of in-network congestion management techniques has been their reliance on dedicated hardware support. Implementing such techniques in the network requires the development of new ASIC designs, which is a time-consuming and expensive process [16]. Consequently, out of the myriad techniques proposed, only a handful have successfully transitioned to production.

Fortunately, this landscape has recently changed thanks to two key contributions that have the potential to revolutionize how we approach congestion management: *programmable scheduling* and *programmable data planes*.

Programmable scheduling aims at representing a wide range of in-network congestion management techniques [17–20] within a common abstraction [1]. The key insight is that, if we develop an abstraction that can generalize the behavior of multiple techniques, and which can be implemented on hardware, we can use it to execute any of the individual techniques without requiring new ASIC designs. Consequently, such an abstraction facilitates both the design of new techniques and their deployment on hardware.

The first programmable scheduling abstraction is "PIFO" [20], which relies on the observation that we can divide most in-network congestion management techniques into two components: a ranking algorithm that determines the priority for processing each packet (available at enqueue), and a queue structure capable of admitting packets, allocating buffer space, and scheduling packets based on their determined priority. One natural candidate for such a queue structure is the Push-In First Out (PIFO) queue, which can sort packets at line rate by "pushing" them into arbitrary positions based on their ranks and serving them from the head of the queue.

1 Despite its name, programmable scheduling extends beyond mere scheduling; it integrates the three primary types of in-network congestion management techniques—admission control, buffer management, and packet scheduling—into a cohesive and unified framework.

Thus, PIFO queues are proposed as the queuing structure enabling programmable scheduling and give the abstraction its name. The value in the PIFO abstraction is in its generalization: it can represent many common work-conserving scheduling algorithms such as weighted fair queuing, strict priorities, and earliest-deadline first [20]. The main challenge of the PIFO abstraction, however, is its hardware implementation. Implementing PIFO queues in hardware is difficult because they must sort packets at line rate and may need to drop high-rank packets after they have been enqueued (e.g., if a low-rank packet arrives).

Programmable data planes have been recently proposed as a new generation of packet processing pipelines that can be flexibly customized to achieve different functionalities while maintaining high-speed packet processing at terabits per second (Tbps) [21]. These devices represent a paradigm shift compared to conventional fixed-function switches, as they enable the deployment of new network functions without the need for new hardware, thereby significantly reducing the time and costs associated.

In this dissertation, we argue that the benefits of programmable data planes can be leveraged to bridge the gaps in programmable scheduling. The strategic integration of these two techniques can pave the way to programmable in-network congestion management on existing devices. Furthermore, we argue that, upon achieving this integration, in-network congestion management can play a pivotal role in enhancing the performance and security of the modern Internet. It has the potential to effectively address even the most challenging types of congestion, which pose significant threats to the stability of the current Internet architecture.

To realize these goals, we introduce four systems: (i) SP-PIFO and PACKS, which approximate the PIFO abstraction on programmable data planes, to enable programmable in-network congestion management on existing devices; (ii) QVISOR, which extends the previous frameworks to accommodate multi-tenant networks such as data centers or cloud environments; and (iii) ACC-Turbo, which leverages the benefits of in-network congestion management to mitigate the latest types of denial-of-service attacks.

The rest of the dissertation is organized as follows:

In Chapter 2, we introduce SP-PIFO, an abstraction to support in-network congestion management on existing programmable data planes. SP-PIFO achieves this by emulating the behavior of PIFO queues atop a collection of strict-priority queues, available on existing devices. It dynamically adjusts the mapping between packet ranks and priority queues to minimize

scheduling errors compared to an ideal PIFO queue. SP-PIFO operates on a per-packet basis and does not require any prior traffic knowledge.

In Chapter 3, we present PACKS, a programmable packet scheduler that outperforms SP-PIFO in emulating the behavior of PIFO queues. PACKS also runs on top of a set of priority queues but it uses packet-rank information and queue-occupancy levels during enqueue to determine whether to admit each incoming packet and how to best map it to the available queues. By combining an admission control and a queue-mapping mechanism, PACKS effectively captures all the necessary behaviors of a PIFO queue.

In Chapter 4, we introduce QVISOR, an extension of the previous frameworks to support multiple congestion-management policies simultaneously, especially tailored for multi-tenant networks. QVISOR enables tenants to define their preferred scheduling policies, and automatically merges and deploys them on top of the underlying hardware resources, streamlining the process and ensuring efficient resource utilization.

In Chapter 5, we present ACC-Turbo, an in-network congestion management technique designed to mitigate the latest form of distributed denial-of-service (DDoS) attacks observed in the wild—pulse-wave DDoS attacks. ACC-Turbo infers attack patterns by applying online-clustering techniques in the network and mitigates them using programmable scheduling.

In Chapter 6, we conclude the dissertation by summarizing our findings, identifying open problems, and proposing new research directions in the field of in-network congestion management for security and performance.

2

SP-PIFO

2.1 INTRODUCTION

In this chapter, we introduce SP-PIFO, a programmable scheduling abstraction designed to operate on existing commodity switches.

Programmable scheduling allows operators to "program" their desired scheduling policies, by tagging packets with *ranks*, which indicate their scheduling priority. *Programmable schedulers* process these packets, and schedule them following the order of their ranks. The main challenge of implementing programmable packet scheduling, is to find an abstraction which is flexible enough to express a wide variety of scheduling algorithms and yet can be implemented efficiently in hardware. In [20], Push-In First-Out (PIFO) queues were proposed as such an abstraction. PIFO queues allow packets to be pushed into arbitrary positions in the queue (according to the packets rank) while only draining packets from the head.

While PIFO queues enable programmable scheduling, implementing them in hardware is hard due to the need to arbitrarily sort packets at line rate. [20] described a hardware design (not implementation) supporting PIFO on top of Broadcom Trident II [22]. While promising, realizing this design in an ASIC is likely to take years [23], not including deployment. Even ignoring deployment considerations, the design of [20] is limited as it only supports ~1000 flows and relies on the assumption that packet ranks increase monotonically within each flow, which is not always the case.

Our work In this chapter, we ask whether it is possible to approximate PIFO queues at scale, in existing programmable data planes. We answer positively and present SP-PIFO, an adaptive scheduling algorithm that closely approximates PIFO behaviors on top of widely-available Strict-Priority (SP) queues—*at line rate, at scale, and on existing devices*. The key insight behind SP-PIFO is to dynamically adapt the mapping between packet ranks and available strict-priority queues in order to minimize the amount of scheduling errors relative to a hypothetical ideal PIFO implementation. We present a mathematical formulation of the problem

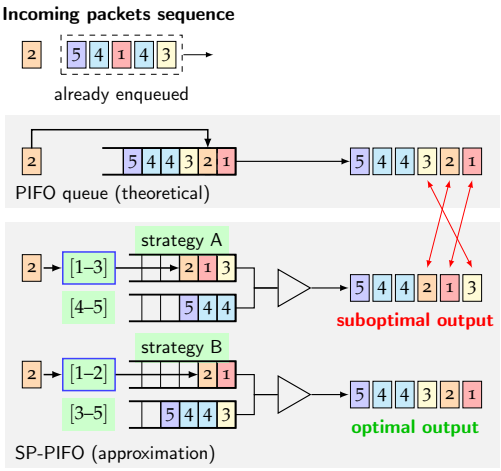


FIGURE 2.1: SP-PIFO approximates the behavior of PIFO queues by adapting how packet ranks are mapped to priority queues.

and derive an adaptation technique which closely approximates the optimal queue mapping without any traffic knowledge in advance.

Example First, we provide an intuition how SP-PIFO approximates PIFO behaviors using SP queues in Fig. 2.1. The example illustrates the scheduling behavior of two SP-PIFO systems which receive the input packet sequence 2 5 4 1 4 3. We write the first packet being enqueued on the far-right (3) and the last one on the far-left (2). Similarly to [20], we also consider that lower-rank packets have higher priority (and use corresponding color codes). The figure illustrates the scheduling decision of each system for the sixth packet (2), assuming the first 5 have been enqueued already.

A PIFO queue always schedules incoming packets perfectly, leading to the sorted output 5 4 4 3 2 1. In contrast, the quality of the scheduling of a SP-PIFO scheme depends on: (i) the number of SP queues available (here, two); and (ii) the mapping of packet ranks to those queues. Fig. 2.1 illustrates two such mapping strategies. Strategy A maps ranks 1–3 (resp. 4–5) to the highest (resp. lowest) SP queue, while Strategy B maps ranks 1–2 (resp. 3–5) to the highest (resp. lowest) SP queue. We see that Strategy B is capable of perfectly sorting the input sequence, i.e. it behaves like a

perfect PIFO queue. In contrast, Strategy A leads to sub-optimal packet inversions, e.g. 1 is incorrectly scheduled after 3.

Insights The key challenge in SP-PIFO is to design adaptation strategies that can: (i) closely approximate PIFO behavior; and (ii) be implemented in programmable data planes. These are hard challenges as the best mapping strategy depends on the traffic mix and the actual ranks being enqueued, both of which can change on a per-packet basis.

SP-PIFO approximates the best mapping strategy by dynamically shifting the ranks mapped to each queue to reduce the scheduling mistakes it observes in real time.

We show that SP-PIFO's adaptation strategy achieves almost the same performance as provably-correct adaptation strategies while being implementable in programmable data planes.

Performance We use SP-PIFO to implement a wide variety of scheduling objectives ranging from minimizing flow completion times to achieving max-min fairness. For all cases, we show that SP-PIFO achieves performance on-par with the state-of-the-art. We also demonstrate that SP-PIFO runs at line rate on existing programmable hardware, at line rate, by deploying it on a Intel Tofino programmable switch.

Contributions The main contributions of this chapter are:

- A novel approach for approximating PIFO queues on top of a set of strict-priority queues (§2.3).
- A greedy, gradient-based algorithm which provably converges to the optimal queue mapping (§2.4).
- An adaptation algorithm which dynamically adapts the queue mapping according to the network conditions, closely-approximating an optimal scheme (§2.5).
- An implementation of SP-PIFO in Java and P4 (§2.6).
- A comprehensive evaluation showing SP-PIFO effectiveness in approximating perfect PIFO behavior with as little as 8 queues and on actual hardware switches (§2.7).

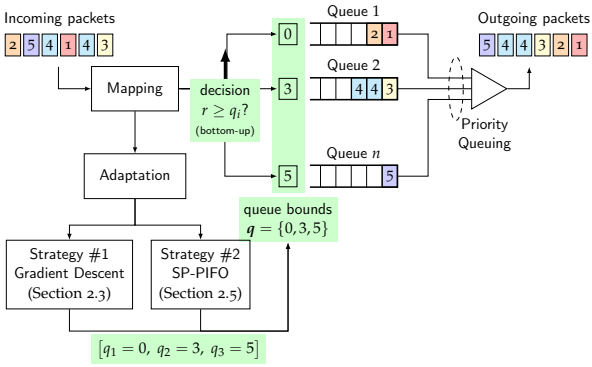


FIGURE 2.2: Overview of SP-PIFO data-plane pipeline.

2.2 OVERVIEW

In this section, we provide an informal overview of how SP-PIFO manages to closely approximate PIFO behaviors. At a high level, SP-PIFO is a priority-queuing scheduling discipline (see Fig. 2.2) which maps incoming packets to n priority queues. SP-PIFO assumes that packets are tagged with a rank indicating the intended scheduling order, with lower ranks being preferred over higher ones. Packets enqueued in a queue are scheduled according to their order of arrival (i.e., First-In First-Out), after *all* packets enqueued in any higher-priority queue have been scheduled. Unlike classical priority-queuing disciplines [24], SP-PIFO dynamically adapts the mapping between the packet ranks and the priority queues according to the observed network conditions. In particular, SP-PIFO adapts the mapping so as to minimize the scheduling "unpifoness", that is, the number of times a higher-rank packet is scheduled *before* an enqueued lower-rank packet. We refer to such scheduling mistakes as *inversions*.

Mapping SP-PIFO maps each incoming packet to queues according to the queue *bounds*. These queue bounds identify, for each queue i, the smallest packet rank that can be enqueued. Whenever a packet is received, SP-PIFO scans the queue bounds bottom-up, starting from the lowest-priority queue, and enqueues the packet in the first queue with a bound smaller or equal to the packet rank. Given a packet with rank $r \in \mathbb{Z}_{\geq 0}$ and n priority queues, let $\boldsymbol{q}$ be the vector of queue bounds $(q_1, \cdots, q_n) \in \mathbb{Z}^n$ such that $0 \leq q_1 \leq q_2 \leq \cdots \leq q_n$. For instance, consider a vector $\boldsymbol{q} = \{0, 3, 5\}$

indicating the bounds of 3 priority queues, with 0 (resp. 5) indicating the bound of the highest- (resp. lowest-) priority queue. Given $\boldsymbol{q}$, SP-PIFO enqueues packets with rank 2 in the first (highest-priority) queue, packets with rank 3 in the second queue and packets with rank 10 in the third (lowest-priority) queue.

Adaptation "Unpifoness" can be minimized across multiple packets, e.g. by monitoring the rank distribution over periodic time windows and adapting the bounds through a gradient descent, or on a per-packet basis (see Fig. 2.2). Depending on the characteristics of the rank distribution, the first strategy can provably converge to the optimal mapping. Unfortunately, its requirements exceed the capabilities of existing programmable data planes. SP-PIFO addresses these two limitations: it works for *any* rank distribution, on existing hardware. SP-PIFO dynamically adapts $\mathbf{q}$ such that the resulting scheduling closely approximates an ideal PIFO queue, minimizing the amount of observed *inversions* by dynamically shifting the ranks mapped to each queue. SP-PIFO operates online, *without* prior knowledge of the incoming packet ranks.

SP-PIFO's adaptation mechanism consists of two stages: a *push-up* stage where future low-rank (i.e., high-priority) packets are pushed to higher-priority queues; and a *push-down* stage where future high-rank (i.e., low-priority) packets are pushed down to lower queues.

Stage 1: Push-up Whenever SP-PIFO enqueues a packet, it updates the corresponding queue bound to the rank of the enqueued packet. Doing so, SP-PIFO aims at ensuring that future lower-ranked packets will not be enqueued in the same queue, but in a more preferred one. Intuitively, SP-PIFO "pushes up" packets with low ranks to highest-priority queues, where they will be drained first. Of course, as the number of queues is finite—and often, much smaller than the number of ranks—this is not always possible, leading to inversions.

Stage 2: Push-down Whenever SP-PIFO detects an inversion in the highest-priority queue (i.e., the packet rank is smaller than the highest-priority queue bound), it decreases the queue bound of *all* queues. Doing so, SP-PIFO ensures that future higher-rank packets will be enqueued in lower-priority queues. Intuitively, after an inversion, SP-PIFO "pushes down" packets with high ranks to the lower-priority queues in order to prevent them from causing inversions in the highest-priority queue. SP-PIFO decreases the queue bounds according to the magnitude of the inversion,

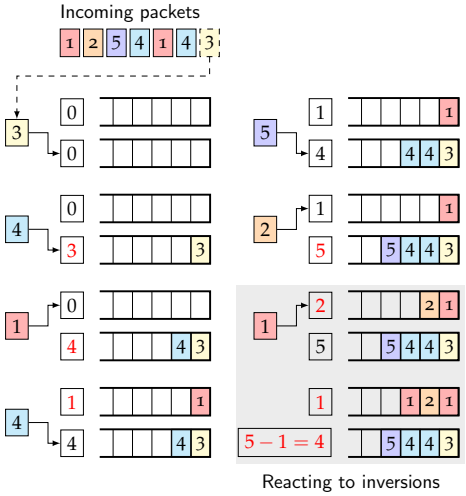


FIGURE 2.3: SP-PIFO mapping and adaptation mechanisms.

i.e. the difference between the packet rank and the corresponding queue bound: the bigger the inversion, the more ranks are pushed down.

Example Fig. 2.3 illustrates the execution of SP-PIFO with two priority queues when receiving 1 2 5 4 1 4 3. Without loss of generality, we consider that the queue bounds are initialized to 0. SP-PIFO enqueues the first packet (3) in the lowest-priority queue and updates its queue bound to 3. Likewise, SP-PIFO also enqueues the second packet, 4, in the lowest-priority queue. As its rank (4) is higher than the queue bound (3), it then updates the queue bound to 4.

The same process is applied to the subsequent packets until the second 1 is encountered, creating an inversion (grayed area in Fig. 2.3). Indeed, SP-PIFO enqueues 1 in the highest-priority queue *after* having enqueued 2. Once the inversion is detected, SP-PIFO adapts the queue bounds to 1 and $5 - 1 = 4$, respectively. Observe that if 1 and 2 keep arriving, the bound of the lowest-priority queue will decrease, eventually reaching 2. At this point, future 1 will not experience inversions anymore as they will have a dedicated queue.

2.3 SP-PIFO DESIGN

In this section, we describe the theoretical basis supporting the design of SP-PIFO. We first phrase the problem of finding the optimal queue bounds as an empirical risk minimization problem in which a loss function—how "unpifo" the current mapping is—is minimized (§2.3.1). We then develop an algorithm based on gradient descent which provably converges to the optimal bounds for stable rank distributions (§2.3.2). We show how the convergence requirements make the algorithm impractical (§2.4.4). In the following, we present SP-PIFO which relaxes the optimality requirements at the benefit of practicality (§2.5).

2.3.1 *Problem statement*

Let $\mathcal{U} : \mathrm{R}^n \times \mathrm{R}_{\geq 0} \rightarrow \mathrm{R}_{\geq 0}$ be a loss function such that $\mathcal{U}(\boldsymbol{q}, r)$ quantifies the approximation error of scheduling a packet with rank r based on queue bounds $\boldsymbol{q}$ compared to an ideal PIFO queue. Intuitively, a smaller loss equals a better approximation. Note that $\mathcal{U}$ stands for *unpifoness*.

The adaptation goal is to find the optimal queue bounds $\boldsymbol{q}^*$ that minimize the expected loss for all possible ranks. Let $\mathcal{Q}$ be the space of all valid bound vectors and $\mathcal{R}$ the distribution of packet ranks, then the optimal queue bounds $\boldsymbol{q}^*$ are:

$$\boldsymbol{q}^* = \underset{\boldsymbol{q} \in \mathcal{Q}}{\arg\min} \underset{r \sim \mathcal{R}}{\mathbb{E}} [\, \mathcal{U}(\boldsymbol{q}, r) \,] \tag{2.1}$$

Finding $\boldsymbol{q}^*$ directly is intractable though. Indeed, evaluating the expected loss $\mathcal{U}$ is impossible since the distribution of packet ranks $\mathcal{R}$ is unknown. We address this problem by considering the *empirical loss* $\mathcal{U}_{emp}$ observed over a set $\mathcal{D}$ of i.i.d. rank samples. Doing so, we phrase the problem of finding $\boldsymbol{q}^*$ as an *empirical risk minimization* (ERM) problem:

$$\boldsymbol{q}^* = \underset{\boldsymbol{q} \in \mathcal{Q}}{\arg\min} \frac{1}{|\mathcal{D}|} \sum_{r \in \mathcal{D}} \mathcal{U}_{emp}(\mathcal{D}, \boldsymbol{q}, r) \tag{2.2}$$

Evaluating empirical losses For a given rank r, we measure the empirical loss $\mathcal{U}_{emp}$ as the *expected* number of inversions that r would encounter, if the rank distribution $\mathcal{D}$ was scheduled given the queue bounds $\boldsymbol{q}$, weighted by the *cost* that each inversion would cause to the system performance. This cost can be just a constant value, if all inversions are treated the same, or it

can measure the magnitude of the inversion (i.e., how big is the difference between ranks causing it). Since r receives inversions only from higher ranks in the distribution, $\mathcal{U}_{emp}$ can be rewritten as:

$$\mathcal{U}_{emp}(\mathcal{D}, \boldsymbol{q}, r) = \frac{1}{|\mathcal{D}|} \sum_{\substack{r' \in \mathcal{D} \\ r' > r}} \text{cost}_{\boldsymbol{q}}(r', r) \tag{2.3}$$

Having formulated the adaptation goal as an empirical risk minimization, we aim to solve it by analyzing how changes in $\boldsymbol{q}$ influence the empirical risk, and trying to design an iterative algorithm capable of converging to the minimal risk.

2.3.2 *Gradient-based adaptation algorithm*

We first introduce a greedy, gradient-based algorithm, which provably converges to the optimal queue bounds $\boldsymbol{q}^*$ provided that the rank distribution stays constant. The algorithm builds upon the fact that inversions cannot occur between ranks mapped to different priority queues. This allows to instantiate the empirical risk minimization in eq. 2.2 at a *queue level* by simply adding the individual losses of each queue. Letting $\mathcal{U}(q_i)$ be the loss function corresponding to the queue with bound q_i, this is:

$$\boldsymbol{q}^* = \underset{\boldsymbol{q} \in \mathcal{Q}}{\arg\min} \sum_{q_i \in \boldsymbol{q}} \mathcal{U}(q_i) \tag{2.4}$$

Letting $p_{\mathcal{D}}(r)$ and $p_{\mathcal{D}}(r')$ be the empirical probability of ranks r and r', respectively, both mapped to the queue with bound q_i, we can define the unpifoness of the queue as:

$$\mathcal{U}(q_i) = \sum_{\substack{q_i \leq r < q_{i+1} \\ r < r' < q_{i+1}}} p_{\mathcal{D}}(r) \cdot p_{\mathcal{D}}(r') \cdot \text{cost}(r', r) \tag{2.5}$$

Overview Considering this problem instantiation, the greedy algorithm first computes the rank distribution over a set of k packets before minimizing the expected per-queue unpifoness by incrementing (resp. decrementing) the queue bounds. Specifically, after processing the k-th packet, the greedy algorithm selects, for each queue, the bound that most decreases the overall system unpifoness. Although comparing the performance of

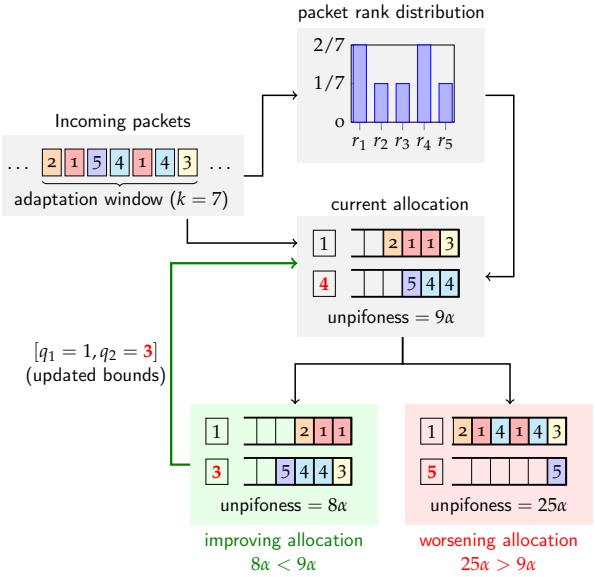


FIGURE 2.4: The gradient-based algorithm greedily minimizes the *unpifoness*.

all bound combinations is not possible, we introduce an efficient computation mechanism that allows to prune the search space while preserving convergence. We prove the optimality of the algorithm in §2.5.2.

Example We illustrate the execution of the algorithm in Fig. 2.4. We assume a system with two priority queues and assume that the packet sequence 2 1 5 4 1 4 3 is received over and over again. We set the adaptation window k to 7 packets. We initialize the queue bounds to 1 and 4.

The algorithm starts by computing the observed rank distribution after receiving the 7-th packet. Here, it estimates the probability of receiving a packet of rank 1 as $p(1) = 2/7$. Similarly, $p(2) = 1/7$, $p(3) = 1/7$, $p(4) = 2/7$ and $p(5) = 1/7$. It then computes the expected unpifoness that this distribution would have generated with the current queue bounds (eq. 2.3). For the higher-priority queue, this is $\mathcal{U}_1 = p(1) \cdot p(2) \cdot cost(2,1) + p(1) \cdot p(3) \cdot cost(3,1) + p(2) \cdot p(3) \cdot cost(3,2) = (2/7 \cdot 1/7) \cdot (2-1) + (2/7 \cdot 1/7) \cdot (3-1) + (1/7 \cdot 1/7) \cdot (3-2)$. This equation can be simplified to $\mathcal{U}_1 = 7\alpha$ where $\alpha = (1/7 \cdot 1/7)$. Similarly, $\mathcal{U}_2 = p(4) \cdot p(5) \cdot cost(5,4) = 2\alpha$, adding up a total of $\mathcal{U} = 9\alpha$.

Next, the algorithm compares the expected unpifoness that would be obtained if the queue bound was incremented (gradient up) or decremented (gradient down) and adapts the queue bound in the direction resulting in the biggest decrease of unpifoness.

Gradient up Incrementing q_2 from 4 to 5 means that only rank $\{5\}$ would be mapped to the lower-priority queue. The resulting unpifoness is $\mathcal{U} = 25\alpha$. The higher unpifoness (25α instead of 9α) indicates that, by incrementing q_2, the system gets further away from the PIFO behavior. Note that the increase in unpifoness comes from the higher-priority queue as rank $\{5\}$ gets an exclusive queue.

Gradient down In contrast, the system unpifoness reduces from 9α to 8α when decrementing q_2 from 4 to 3. Indeed, $\mathcal{U}_1 = p(1) \cdot p(2) \cdot cost(2,1) = 2\alpha$, and $\mathcal{U}_2 = p(3) \cdot p(4) \cdot cost(4,3) + p(3) \cdot p(5) \cdot cost(5,3) + p(4) \cdot p(5) \cdot cost(5,4) = 6\alpha$, adding up to $\mathcal{U} = 8\alpha$. As such, the adaptation mechanism updates the queue bound: $q_2 = 3$.

The above process repeats every 7-th packet, estimating the rank distribution before greedily adapting the queue bounds.

2.4 GRADIENT-BASED ALGORITHM

In this section, we detail the greedy iterative algorithm presented in §2.3.2 and prove how it converges to the optimal solution. Then, we show how to effectively prune the search space making computation efficient while keeping convergence (§2.4.1). Finally, we analyze its implementation (§2.4.2) and convergence requirements (§2.4.3).

The algorithm (alg. 1) iteratively minimizes the risk by adjusting queue bounds, one queue and one step at a time, until reaching convergence. At each iteration, the algorithm predicts, for every q_i, whether moving the bound by one (in either direction) decreases the expected risk, and moves the bound in the direction of maximum decrease. In the following, we discuss first, how the algorithm can predict the expected change in risk, and second, why checking a single step is sufficient to converge.

Risk difference In §2.3.2, we demonstrated that the risk can be analyzed on a per-queue basis from the cost of mapping packets with different ranks to the same queue. Consequently, changes in the risk resulting from changing the bound vector $\boldsymbol{q}$ can be analyzed by comparing the risk difference in

Algorithm 1 Greedy optimization

Require: k: Step size, $\boldsymbol{q}_{\text{init}}$: Initial bounds

procedure ADAPTATION
  $\mathcal{D} \leftarrow \emptyset$
  $\boldsymbol{q} \leftarrow \boldsymbol{q}_{\text{init}}$ ▷ Initialize bounds
  for all p: incoming packet **do**
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{rank(p)\}$ ▷ Collect samples
    if $|\mathcal{D}| = k$ **then** ▷ Adapt bounds
      $\mathcal{P} \leftarrow$ COMPUTERANKPROBABILITES$(\mathcal{D})$
      repeat
        $\boldsymbol{q} \leftarrow$ UPDATEMAPPING$(\boldsymbol{q},\ \mathcal{P})$
      until $\boldsymbol{q}$ converges
      $\mathcal{D} \leftarrow \emptyset$ ▷ Reset samples
    end if
  end for
end procedure
function UPDATEMAPPING$(\boldsymbol{q},\ \mathcal{P})$
  for $q_i \in \boldsymbol{q}$ **do**
    $\Delta^+ \leftarrow$ RISKFROMINCREMENT$(q_i,\ \mathcal{P})$
    $\Delta^- \leftarrow$ RISKFROMDECREMENT$(q_i,\ \mathcal{P})$
    if $(\Delta^+ \leq 0)$ and $(\Delta^+ \leq \Delta^-)$ **then**
      $q_i \leftarrow q_i + 1$
    else if $(\Delta^- \leq 0)$ and $(\Delta^- < \Delta^+)$ **then**
      $q_i \leftarrow q_i - 1$
    end if
  end for
  return $\boldsymbol{q}$
end function

affected queues. To be precise, every change of a single element q_i in $\boldsymbol{q}$ affects two queues, queue i and $i-1$, as ranks are either moved from i to $i-1$ (increase in q_i) or moved from $i-1$ to i (decrease in q_i).

Theorem 2.1. *Let $r^* = q_i$, let Q_i be the set of ranks mapped to queue i (before any changes). Increasing q_i by 1 changes the risk by:*

$$\Delta_i^+ = p(r^*)\Big(\sum_{r \in Q_{i-1}} p(r)cost(r^*, r) - \sum_{r \in Q_i} p(r)cost(r, r^*)\Big) \tag{2.6}$$

Let $r^* = q_i - 1$. *Decreasing* q_i *by* 1 *changes the risk by:*

$$\Delta_i^- = p(r^*)(\sum_{r \in Q_i} p(r) cost(r^*, r) - \sum_{r \in Q_{i-1}} p(r) cost(r, r^*)) \tag{2.7}$$

Proof: Increasing q_i removes the lowest rank from queue i, which now becomes the highest rank in queue $i-1$. As the new highest rank in queue $i-1$, it causes possible inversions and therefore risk for *all* other ranks in queue $i-1$, resulting in the first, positive term in eq. 2.6. Conversely, as the lowest rank in queue, it was prone to receive inversions from any other element in the queue, supposing a risk in queue i that is removed with the change. This risk reduction results in the second, negative, term.

The proof for decreasing q_i is symmetrical, with the main difference that now, rank q_{i-1} is the one changing from queue $i-1$ to queue i.

Greedy step Based on the theory presented, the algorithm computes the risk and either (for every q_i):

(a) Does not move q_i, if neither incrementing or decrementing reduces the expected risk.

(b) Increments q_i, if incrementing decreases the risk more than decrementing.

(c) Decrements q_i, if decrementing decreases the risk more than incrementing.

This effectively prunes the search space. At every iteration, the algorithm only requires a constant amount of comparisons, and it does not explore directions further in case they increase the risk. In the following, we show why deciding not to explore a direction further after a single step is reasonable.

Theorem 2.2. *Let* Δ_i^+ *and* Δ_i^- *denote the prospective in- and decreases from incrementing/decrementing* q_i *by* 1. *Let* Δ_i^{++} *and* Δ_i^{--} *denote the in- and decreases from incrementing/decrementing* q_i *by more than* 1. *Let the cost function used to compute the differences be non-decreasing in* $|r^* - r|$ *and* 0 *if and only if* $r^* = r$. *Then:*

1. *If* $\Delta_i^+ > 0$, *then* $\Delta_i^{++} > 0$.
2. *If* $\Delta_i^- > 0$, *then* $\Delta_i^{--} > 0$.

Proof:

1: If $\Delta_i^+ > 0$,

$$\sum_{r \in Q_{i-1}} p(r)\text{cost}(r^*, r) > \sum_{r \in Q_i} p(r)\text{cost}(r, r^*) \tag{2.8}$$

Let $r^{**} = q_i + 1$, i.e. the second-lowest rank in queue i, which would be moved if we move the queue bound by more than 1. Moving both r^* and r^{**} would cause the following change in risk:

$$\Delta_i^{++} = \tag{2.9}$$
$$p(r^*)(\sum_{r \in Q_{i-1}} p(r)\text{cost}(r^*, r) - \sum_{r \in Q_i} p(r)\text{cost}(r, r^*)) + \tag{2.10}$$
$$p(r^{**})(\sum_{r \in Q_{i-1}} p(r)\text{cost}(r^{**}, r) - \sum_{r \in Q_i} p(r)\text{cost}(r, r^{**})) \tag{2.11}$$

Note that we can omit the cost between r^* and r^{**} in eq. 2.11: as the cost function is by definition symmetric, the additional increase in the left-hand term is exactly equal in magnitude to the additional decrease in the right-hand term, and thus they cancel each other. Thus we omit the term to not clutter the notation. Next, again by definition of the cost function, if $r^{**} > r^* > r$, then $cost(r^{**}, r) \geq cost(r^*, r)$, and if $r > r^{**} > r^*$, then $cost(r, r^{**}) \leq cost(r, r^*)$. Additionally, we note that the order of arguments in the cost function does not matter, as it is symmetrical. Applied to the risk of the lower- and higher-priority queue respectively (eq. 2.11), this gives:

$$\begin{aligned} \sum_{r \in Q_{i-1}} p(r)cost(r^{**}, r) &\geq \sum_{r \in Q_{i-1}} p(r)cost(r^*, r) \\ \sum_{r \in Q_i} p(r)cost(r, r^{**}) &\leq \sum_{r \in Q_i} p(r)cost(r, r^*) \end{aligned} \tag{2.12}$$

And in conclusion, the left hand term in eq. 2.11 is larger than the left hand term in eq. 2.10, and the right hand term in eq. 2.11 is smaller then the left hand term in eq. 2.10. Consequently, if eq. 2.10 is positive, eq. 2.11 must also be positive (as probabilities are always positive), proving that if one step does increase the risks, two steps will also increase the risk. The exact same procedure can be repeated for larger step sizes, which we omit here.

2: This proof is conceptually identical to the other direction, and we will thus omit it. The guiding principle is the same: moving more than

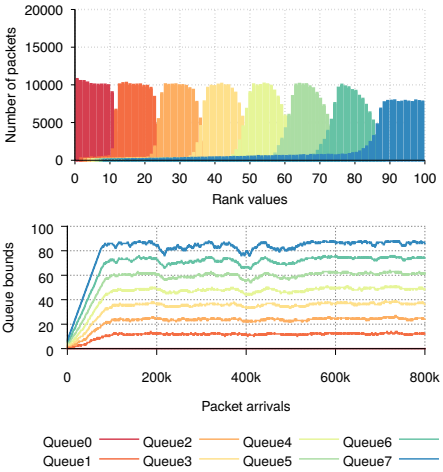


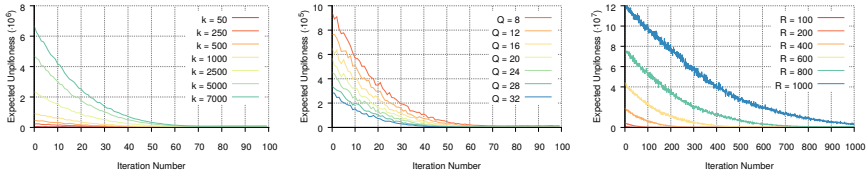
FIGURE 2.5: Greedy convergence for uniform rank distribution.

one rank can only cause higher increase in risk in the queue the ranks are moved to, and lower decrease in risk in the queue the ranks are taken from, compared to the previous ranks. Thus, if already moving one rank causes a higher increase in risk in one queue than decrease in the other, moving additional ranks does not change this.

Conclusion We have explained how the greedy algorithm only requires exploring the direction which offers a potential decrease in risk, and we have proved how the risk does not decrease with the distance between ranks (it cannot be better to have a bigger inversion, only equal or worse). This allows the greedy algorithm to quickly decide if a direction is not worth investigating, effectively pruning the search space.

2.4.1 *Efficient computation*

As tracking the complete rank distribution at each iteration might be too expensive in terms of memory, and repeating the adaptation until convergence too costly in terms of complexity, we show in the following lines how the mathematical formulation of the problem allows a simplified implementation which only requires 4 counters per queue.



(a) Convergence vs. adaptation window

(b) Convergence vs. number of queues

(c) Convergence vs. rank range

FIGURE 2.6: Greedy algorithm adaptation microbenchmark.

From the empirical probability definition, $p_{\mathcal{D}}(r) = |r_{\mathcal{D}}|/|\mathcal{D}|$, we can rewrite eq. 2.6 and eq. 2.7 as:

$$\begin{aligned} \Delta_i^+ &= \frac{|q_i|}{|\mathcal{D}|^2} \cdot \Big(\sum_{r \in Q_{i-1}} |r| \text{cost}(q_i, r) - \sum_{r \in Q_i} |r| \text{cost}(r, q_i) \Big) \\ \Delta_i^- &= \frac{|q_i - 1|}{|\mathcal{D}|^2} \cdot \Big(\sum_{r \in Q_i} |r| \text{cost}(q_i - 1, r) - \sum_{r \in Q_{i-1}} |r| \text{cost}(r, q_i - 1) \Big) \end{aligned} \tag{2.13}$$

Since the queue bound q_i stays constant throughout the adaptation window, each of the summations in eq. 2.13 can be implemented through a counter which gets updated every time a new packet arrives, with its carried rank. Note that the number of counters required increases linearly with the number of queues. Also, observe that the counters in eq. 2.13, only allow the computation of one step in the gradient. However, this is enough since the one-step version manages to converge in practice (cf. Fig. 2.5).

2.4.2 *Implementation requirements*

With the computation presented in §2.4.1, implementing the gradient-based algorithm on top of n priority queues, requires n registers for queue-bound storage and $(4 \cdot n)$ registers for the gradient computation. The mapping process §2.2 requires packets to potentially read all the queue-bound values (i.e., for packets scheduled in the highest-priority queue). In the same direction, while most packets only need to update the two counters corresponding to their queue, the k_{th} packet in each sequence needs to access *all* counters to perform the adaptation decision. This supposes being able to read $n + (4 \cdot n)$ different registers for a single packet (without even considering the updates). Since existing devices only support up to

12-16 stages, with a single register access per stage [25], the implementation of the greedy algorithm is not feasible for a practical number of queues.

2.4.3 *Convergence analysis*

We now show how the greedy-algorithm performance varies when modifying the three main degrees of freedom: (i) the adaptation window (i.e., the number of packets that are monitored before the adaptation mechanism is executed); (ii) the number of queues available in the strict-priority scheme; and (iii) the number of ranks in the distribution. For that, we analyze the unpifoness evolution of a single switch running the greedy algorithm for a uniform rank distribution from 0 to 100 until convergence. We compute unpifoness as specified in §2.3.1, based on the packets scheduled and the queue bounds used during the adaptation window.

Effects of varying the adaptation window Fig. 2.6a shows the unpifoness evolution when we run the greedy algorithm on top of a strict-priority scheme of 8 queues, and we vary the adaptation window from 50 to 7000 packets. We observe that, for the algorithm to converge, the adaptation window needs to be broad enough to cover a *complete* sample of the rank distribution (i.e., one that characterizes all its representative behaviors). In our case, any adaptation window below 100 packets can not characterize completely the rank distribution. Indeed, Fig. 2.6a depicts how the greedy algorithm correctly converges as soon as more than 200 packets are monitored per iteration. In general, the broader the adaptation window, the more precise the rank distribution estimate, and the better the adaptation decision. However, while a too narrow adaptation window can suppose missing important information of the rank distribution and breaking convergence guarantees, a too broad adaptation window can make the algorithm too slow to converge, negatively impacting the performance.

Finally, the greedy algorithm only converges if the rank distribution has a smaller variability than the adaptation rate (i.e., the rank distribution is stable during the time it takes for the algorithm to converge). Relating it to the previous point, simpler rank distributions, which require narrower adaptation windows, can afford higher levels of variability. In contrast,

complex distributions which take longer to adapt and are required to keep stable longer for the algorithm to converge.

Effects of varying the number of queues Fig. 2.6b depicts the case in which we fix an adaptation window of 1000 packets, and modify the number of queues from 8 to 32. All queues have a constant size of 10 packets. We see how the higher number of queues the lower the unpifoness, and the better the PIFO approximation. This is expected since each queue can be perceived as an opportunity to sort packets with different ranks, and therefore to reduce the number of inversions. Also, we can see how the number of iterations required by the algorithm to converge does not directly depend on the number of queues. This results from the fact that each adaptation decision analyzes (and, if required, updates) potential redesigns for *all* the different queue bounds.

Effects of varying the number of ranks Fig. 2.6c presents the effects of modifying the range of the uniform rank distribution from 100 to 1000 ranks, when we fix the number of queues to 8 and the adaptation window to 1000 packets. As expected, under the same number of queues, a higher number of ranks implies an increase in unpifoness. Also, as the rank ranges get closer to the adaptation window, the distribution estimates get worse, and the adaptation gets tougher.

2.4.4 *Limitations*

While the adaptation algorithm described above provably converges to the optimal mapping (see §2.4), two key limitations make it impractical. First, it is not currently implementable in existing programmable data planes due to resource constraints. Second, the algorithm only converges for stable rank distributions, which is rarely the case, and its convergence time directly depends on the distribution size, which can be large. We explain how to overcome these limitations in §2.5.

Hardware restrictions Monitoring the rank distributions over periodic adaptation windows requires a high amount of memory and computational resources, both of which are scarce in current programmable data planes. In particular, implementing the greedy algorithm in hardware (see §2.4.1) requires to: (i) store the value of each queue bound; (ii) compute the current unpifoness; and (iii) estimate the unpifoness obtained by incrementing or

decrementing each queue bound. As we explain in §2.4.2, the amount of resources required to run the algorithm on a practical number of queues (8 queues or more) exceeds the capabilities of current switch designs.

Convergence In §2.4.3, we study the performance of the gradient-based algorithm and analyze the effects on convergence when the adaptation window, the number of queues, and the rank range is modified. We show that, for the algorithm to converge, the rank distribution needs to be stable in time. However, this is unrealistic in most practical scenarios where not only the rank distribution *is unknown* but also varies through time (e.g., virtual times in fair-queuing schemes).

2.5 OUR APPROACH: SP-PIFO

We now present SP-PIFO, an approximation of the gradient-based adaptation algorithm (§2.3.2) which is implementable in existing data planes and rapidly adapts to varying rank distributions. SP-PIFO substitutes the gradient computation by a simpler adaptation process which minimizes the probability of inversions *per packet*, rather than per k-packets.

In the following, we first show how to instantiate the empirical risk minimization problem (eq. 2.2) at the packet level and describe how SP-PIFO solves it (§2.5.1). We then systematically characterize how SP-PIFO handles inversions (§2.5.3).

2.5.1 *Per-packet adaptation algorithm*

The SP-PIFO adaptation algorithm (alg. 2) is based on two competing stages that act in opposing direction. We show that this combination manages to strike a balance in the number of inversions observed by all queues, resulting in a good PIFO approximation. In the following, we first show how to phrase the empirical risk minimization problem at the per-packet level before describing both mechanisms.

Problem statement In contrast to §2.3.2, we aim at minimizing the cost generated by scheduling each individual packet. Formally, we aim to find

the optimal bound vector $\boldsymbol{q}^*$ that minimizes the unpifoness for all enqueued packets $\mathcal{P}$:

$$\boldsymbol{q}^* = \arg\min_{\boldsymbol{q} \in \mathcal{Q}} \mathcal{U}(\mathcal{P}, \boldsymbol{q}) \tag{2.14}$$

Let $r(p)$ be the rank of a given packet $p \in \mathcal{P}$, and let $r_p(p,\ \boldsymbol{q})$ be the rank *perceived* as a result of the mapping decision, which is identified as the highest rank amongst those of packets sharing the same queue. Considering that the objective for the bound vector $\boldsymbol{q}$ is to perfectly approximate PIFO behaviors, we can estimate the unpifoness at enqueue as:

$$\mathcal{U}(\mathcal{P}, \boldsymbol{q}) = \sum_{p \in \mathcal{P}} \text{cost}_{\boldsymbol{q}}(p) \tag{2.15}$$

where

$$\text{cost}_{\boldsymbol{q}}(p) = r_p(p,\ \boldsymbol{q}) - r(p) \tag{2.16}$$

Computing the rank perceived requires determining the highest rank among all packets sharing the queue at any given moment. This not only requires to keep track of all ranks in each queue, but also selecting the highest, which is computationally expensive. Since one of the premises of SP-PIFO is to be implementable in the data plane, we relax this condition and keep track of only a single parameter q_i per queue. These parameters, $\boldsymbol{q}$, which we refer to as *queue bound*, simplifies the cost estimation of a potential mapping decision at enqueue. We discuss how we update these parameters as well as the tradeoffs of this relaxation below.

Stage 1: "Push-up" The first stage increases $\boldsymbol{q}$ to minimize the unpifoness of the queue to which the incoming packet is mapped. Specifically, the mapping process scans the queues bottom-up and enqueues the packet in the first queue that satisfies $r(p) \geq q_i$. It then increases q_i to the rank of the enqueued packet. By doing so, the mechanism minimizes (i) the cost for each packet p (at enqueue time); as well as (ii) the impact that this decision may have on future packets.

This mapping process guarantees a zero-cost packet allocation for all packets within a queue. That is, as we effectively keep track of the highest rank per queue, we ensure that no packet with lower rank is mapped to the same queue. This holds for all queues except for the highest-priority queue. There, packets are enqueued even if $r(p) < q_1$.

Stage 2: "Push-down" As illustrated in §2.2, the first stage can lead to inversions in the highest-priority queue. The second stage aims at counteracting that effect by reducing the number of ranks enqueued in the

Algorithm 2 SP-PIFO adaptation algorithm

Require: An incoming packet with rank r.

```
1:  procedure PUSH-UP
2:      for q_i : q_1 to q_n, q_i ∈ q do                ▷ Scan bottom-up
3:          if r ≥ q_i or i = 1 then
4:              q_i ← r                                  ▷ Update queue bound
5:              ENQUEUE(r, i)                            ▷ Select queue
6:          end if
7:      end for
8:  end procedure
9:  procedure PUSH-DOWN
10:     if r < q_1 then                                  ▷ Detect inversion
11:         cost ← q_i − r                               ▷ Compute cost inversion
12:         for q_j ∈ q, j ≠ i do
13:             q_j ← q_j − cost                         ▷ Adapt queue bounds
14:         end for
15:     end if
16: end procedure
```

highest-priority queue. This is achieved by decreasing *all* queue bounds by some given amount. The exact decrease applied to each q_i introduces a tradeoff between the packets that can be mapped ($\exists i \; s.t. \; r \geq q_i$) and packets that cause inversions in the highest-priority queue ($r < q_1$). Different decreasing strategies exist. In SP-PIFO, we decrease each q_i proportionally to the cost of the inversion. That is, we decrease all queue bounds by $q_1 - r(p)$. This choice is both (i) practical, as it can be efficiently implemented in hardware; and (ii) functional, as it results in a reasonable balance between inversions in the highest-priority queue and shifts in the other queues. Below, we provide some insights on the nature of this balance and why it is important for a good PIFO approximation. We simulate the performance of different decreasing strategies in §2.5.3.

Tradeoffs Unlike the gradient-based algorithm (§2.3.2), SP-PIFO may converge to a sub-optimal solution exhibiting inversions. One can distinguish three sources of inversions. First, there can be inversions in the highest-priority queue. These inversions are proportional to the probability of observing packets with rank $r(p) < q_1$. Second, after the "push-down" stage, the queue bounds do not necessarily match the highest rank packet

in the queue anymore. This may lead to inversions for future packets and is proportional to how often, and how much, queue bounds are decreased. Finally, because only the highest rank in a queue is tracked, it can happen that a packet is enqueued in a higher-priority queue because $r(p) < q_i$, while $r(p)$ is greater than the *lowest* rank in queue i, causing an inversion. This is proportional to the number of ranks between the minimum rank in the queue and the queue bound.

Average-case analysis The exact amount of inversions introduced by each of these three sources is hard to quantify as queue bounds are shifting with (almost) every packet. Yet, *on average*, we can show that the dynamics of SP-PIFO counteract all three sources. On the one hand, it equalizes the probability of $r(p) < q_1$ with the probability of packets being mapped to a specific queue, striking a balance between inversions because there are no higher-priority queues, and inversions because of queue bound mismatch. Furthermore, for this equalizing, the probabilities of specific ranks are weighted more if they are far away from queue bounds, which keeps queues more compact to reduce the chance of overlap.

As a result, on average workloads, SP-PIFO provides a good approximation, and can adapt to arbitrary rank distributions. Nevertheless, there are adversarial packet orderings circumventing these mechanisms, resulting in large unpifoness (§2.8).

2.5.2 *Theoretical analysis of SP-PIFO*

We now provide the theoretical foundations behind SP-PIFO. SP-PIFO is a highly-dynamic probabilistic system. In particular, its queue bounds $\boldsymbol{q}$ change with nearly every incoming packet. Nevertheless, in this section we show that the system has an attractive equilibrium $\boldsymbol{q}^*$ (2.5.2), how this equilibrium balances the different causes of inversions (2.5.2), and we discuss the limitations and open question of our analysis (2.5.2).

Stable equilibrium

Queue-bound dynamics Consider SP-PIFO as a discrete-time system, where each time step corresponds to an arriving packet. Let $\boldsymbol{q}^t$ be the queue bounds at step t, when the t-th packet arrives. Then, the bounds at step $t+1$ are:

$$\boldsymbol{q}^{t+1} = \boldsymbol{q}^t + \Delta(r^t) \tag{2.17}$$

where r^t is the rank of the t-th packet, and $\Delta(r^t)$ is the change this packet causes on the queue bounds. The queue-bound change is given by the "push-down" and "push-up" stages of SP-PIFO, respectively. If the packet causes an inversion in the highest-priority queue, all queue bounds are *decreased* by $q_1^t - r^t$. Otherwise, there is exactly one queue i such that $q_i^t \leq r^t < q_{i+1}^t$, and only q_i is set to r^t, or equivalently, is *increased* by $r^t - q_i^t$. Finally, let $p(r^t)$ be the probability of rank r for the t-th packet. Then, the expected value of the queue bounds at step $t+1$, and the expected difference to the queue bounds at step t are, respectively: [1]

$$\mathrm{E}\left[q_i^{t+1}\right] = \mathrm{E}\left[q_i^t\right] \tag{2.18}$$

$$+ \underbrace{\sum_{q_i^t \leq r^t < q_{i+1}^t} p(r^t)(r^t - q_i^t)}_{\Delta_i^+(\boldsymbol{q}^t, r^t)} \tag{2.19}$$

$$- \underbrace{\sum_{r^t < q_1^t} p(r^t)(q_1^t - r^t)}_{\Delta^-(\boldsymbol{q}^t, r^t)} \tag{2.20}$$

$$\Leftrightarrow \mathrm{E}\left[q_i^{t+1} - q_i^t\right] = \Delta_i^+(\boldsymbol{q}^t, r^t) - \Delta^-(\boldsymbol{q}^t, r^t) \tag{2.21}$$

Equilibrium As expected, we can see from eq. 2.21 that the change of queue bounds is determined by the "push-up" (Δ_i^+) and "push-down" (Δ^-) stages working against each other. Indeed, if Δ_i^+ is larger than Δ^-, the queue bound increases, and vice versa. The system has an equilibrium $\boldsymbol{q}^*$, where $\Delta_i^+ = \Delta^-$ and the expected change is 0. Note that this equilibrium depends on the rank probability.

Attraction The equilibrium $\boldsymbol{q}^*$ is attractive, i.e. if $q_i^t < q_i^*$, $\mathrm{E}[q_i^{t+1} - q_i^t] > 0$, and vice versa. For small perturbations, this is straightforward. Assume that

1 For queue $i = n$, there is no q_{i+1}^t and there is no upper bound on r^t.

all queue bounds are in equilibrium, except q_i. If $q_i^t < q_i^*$, then $\Delta_i^+(\boldsymbol{q}^t, r^t) > \Delta_i^+(\boldsymbol{q}^*, r^t)$, because the sum in eq. 2.19 has (i) more (non-negative) terms; and (ii) each term is weighted stronger, as the difference $r^t - q_i^t$ is larger. On the other hand, $\Delta^-(\boldsymbol{q}^t, r^t)$ is either equal to $\Delta^-(\boldsymbol{q}^*, r^t)$ (for $i > 1$) or even smaller (for $i = 1$, as there are less, and lesser weighted, terms in the sum 2.20). Thus, the increase is larger than the decrease, and the expected change to q_i is positive. The argument for $q_i^t > q_i^*$ is symmetrical.

For larger disturbances, the equilibrium is also attractive, but it might take more than a single time step, as the "push-up" stage for q_i also depends on q_{i+1}: if both $q_i < q_i^*$ and $q_{i+1} < q_{i+1}^*$, the "push-up" might be too weak to pull q_i towards the equilibrium. However, this is not the case for the lowest-priority queue q_n, for which the "push-up" does not depend on another queue. Thus, lower-priority queues (at least q_n) might be pulled towards the equilibrium at first, while other q_i are not. Notice that an expected increase of q_{i+1}^t increases the "push-up" mechanism for q_i^{t+1} and decreases it for q_{i+1}^{t+1} (eq. 2.19). Eventually, as the lower-priority queue bound is getting closer to the equilibrium, the higher-priority queue bound is also pulled towards the equilibrium. This continues until the highest-priority queue, where an expected increase of q_1^t also increases the "push-down" mechanism for all bounds at step $t + 1$ (eq. 2.20). As a result, over multiple time steps, the expected effects of the "push-up" and "push-down" stages equalize, eventually pulling all q_i towards q_i^*.

Balance

As explained in §2.5, there are three main reasons for unpifoness: (i) inversions in the highest-priority queue, after which all queue bounds are decreased; (ii) inversions in a lower-priority queue after its queue bound has been decreased; (iii) inversions in a lower-priority queue, if its highest rank "overtakes" the lowest rank of a higher-priority queue.

As we can see in eq. 2.19, eq. 2.20, and eq. 2.21, all these factors play a role in the dynamics of SP-PIFO. At the equilibrium, the probability of "push-down", which is exactly the probability of an inversion in the highest-priority queue (weighted by its severity), is equalized with the probability of a packet being mapped to any other queue (again weighted, more on this below). While this does not directly correspond to inversions, the more packets are mapped to lower-priority queues, the higher is the probability of an inversion in those queues after a "push-down". SP-PIFO thus keeps a

balance between inversions (i) and (ii), as decreasing (i) would require a stronger "push-down", which would then increase (ii), and vice versa.

Finally, as mentioned above, the ranks in a queue are weighted by how far they are away from the queue bound ($r^t - q_i^t$). This penalizes long (in terms of distinct ranks) queues, which helps to reduce (iii), as the probability for one queue "overtaking" another increases the further the actual queue bound is from the highest-rank packet in the queue, which increases with the length of the queue.

Assumptions and limitations

The analysis presented above is based on a few assumptions, which we argue are justified, yet pose some open questions.

First, we assume that there exists a finite distribution of ranks. This is given in practice. Since ranks need to be processed and stored in hardware, which offers restricted resources, rank ranges must have a limited size.

Second, although SP-PIFO can rapidly adapt to varying rank distributions (in particular faster than the greedy algorithm), we assume that the rank distribution is stable enough such that an equilibrium can exist at all. However, it remains an open question whether there is a point in which the rank-distribution variation might be too fast for the system to actually converge to an equilibrium. In that (hypothetical) case, the analysis presented herein would not be useful to provide any additional insights on the performance of SP-PIFO.

Finally, we assume that the ranks appear in random order, independently from each other. At the first glance, this may seem irrational, as many scheduling algorithms have some structure in the way how ranks are assigned to packets for a given flow. Nevertheless, in practical scenarios, many flows are scheduled together, and even though the ranks for individual flows might be structured, the combined ranks of packets across flows become randomized.

Adversarial workloads Based on the previous assumptions, we have shown that SP-PIFO is attracted towards an expected equilibrium, in which the different sources of unpifoness are balanced. However, there are also some limitations. On the one hand, this equilibrium exists only in expectation, and the queue bounds are also only attracted to it in expectation. The actual queue bounds depend on the order in which packets arrive,

as do inversions. So, even though on average, assuming a random rank ordering, the system might be balanced, there exist particular *adversarial* rank orderings, which "outplay" the two stages to create events of large unpifoness. An adversary might attempt to abuse this by coordinating a large number of flows to force an adversarial ordering of packet ranks. As an example, she might try to increase all queue bounds as much as possible before triggering a "push-down" reaction (e.g., by generating sequences of monotonically-increasing packet ranks). With the sudden decrease in queue-bound values, the high-rank packets mapped in the queues would generate inversions to the new packets.

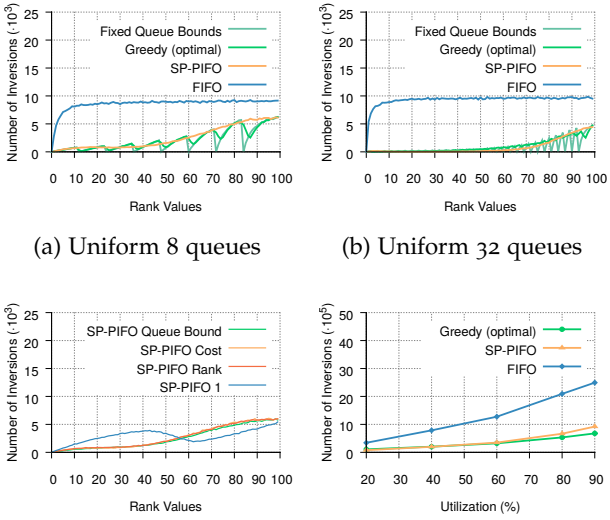
Nevertheless, any non-malicious coexisting flow can easily thwart such strategies, by just randomly breaking the adversarial order. Still, it might be interesting to classify all adversarial orderings, and subsequently monitor the network to actively detect such type of attacks.

2.5.3 *SP-PIFO analysis*

We now dive deeper into understanding SP-PIFO using switch-level simulations. We compare its behavior to that of an ideal PIFO queue, along with several well-known scheduling schemes (e.g., FIFO). We first describe the high-level behavior using a uniform rank distribution (§2.5.3), before systematically exploring the design space (§2.5.3).

Methodology We implement various scheduling schemes (including SP-PIFO, FIFO, and our gradient-based algorithm) in Netbench [26, 27], a packet-level simulator. We analyze the performance of a single switch scheduling 1500 flows of 1MB (fixed), which start according to a Poisson distribution. We run the simulation during one second. We limit the transmission through an output link of 10 Gbps which corresponds to an average port utilization of 75%. We measure the number of inversions *generated by each rank* at *dequeue*. Whenever a packet is polled, we check whether its rank is higher than any of the ranks remaining at any of the queues. When this occurs, we count an inversion *to the rank generating it* (i.e., the one of the polled packet), making sure that inversions are counted at most once per polled-packet, regardless of the number of packets affected by it.

We compare four scheduling schemes: (i) SP-PIFO (§2.5); (ii) the gradient-based algorithm (§2.3, see implementation in 2.4.1); (iii) a strict-priority scheme fixed to the optimal mapping for a uniform distribution (i.e., bounds



(a) Uniform 8 queues (b) Uniform 32 queues

(c) Adaptation strategies (d) Utilization

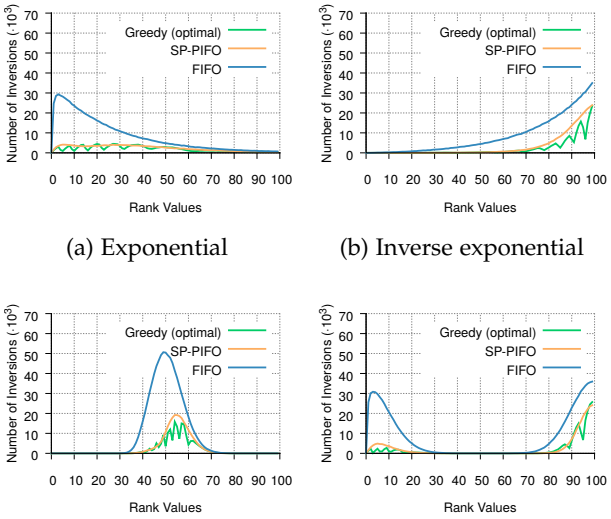
FIGURE 2.7: SP-PIFO performance (uniform rank distribution).

distributed uniformly across ranks, $q_i = 12i$); and (vi) a FIFO queue, as baseline. All strict-priority schemes (SP schemes) use 8 queues of 10 packets, while the FIFO queue has a capacity of 80 packets.

Characterizing general SP-PIFO behavior

We start by showcasing how SP-PIFO handles inversions by analyzing its behavior under a uniform rank distribution. That is, we tag the packets with a rank drawn from a uniform distribution (between 0 and 100).

Fig. 2.7a illustrates the number of inversions generated by each rank for the different SP schemes in comparison with FIFO. We see that a FIFO queue generates a uniform number of inversions across all ranks (since they all share the same queue). In contrast, SP schemes (all the others in Fig. 2.7a) generate a progressively-higher number of inversions as rank values increase. This occurs as higher ranks are mapped to lower-priority queues, which drain packets less frequently. Since those queues have a higher occupancy on average, the potential number of inversions increases. This behavior, however, is not preserved for the lowest-priority queue (the far-right peak in the graph) as a result of starvation. Despite having the



(a) Exponential (b) Inverse exponential

(c) Poisson (d) Convex

FIGURE 2.8: SP-PIFO performance (alternative distributions).

largest average queue size, this queue drains fewer packets and, as such, the number of inversions it sees decreases.

For the fixed-queue bounds, we see that a saw-shape delineates the inversions observed across ranks in different queues, reaching the x axis for the ranks corresponding to the queue bounds. Indeed, the lowest rank within each queue never generates inversions since the other ranks sharing the queue have higher values. The second-lowest rank can only generate inversions to the lowest, and the progression continues until the highest rank, which can generate inversions to all the lower ranks in the queue.

When considering the gradient-based greedy algorithm and SP-PIFO, we see that the saw-shape vanishes. This is because queue bounds are not fixed anymore and successive packets of a given rank can be mapped to multiple queues. Since the rank distribution sampled at each adaptation window varies, the queue-bound design in the gradient-based algorithm oscillates. In SP-PIFO, as a higher variability is produced, the number of inversions delineates the *envelope* of the optimal scheme.

Characterizing SP-PIFO design space

We now systematically explore the design space of SP-PIFO along four dimensions: the number of queues, the adaptation strategy in the push-down stage (§2.5.1), the utilization levels, and the rank distributions. SP-PIFO manages to approximate the optimal algorithms in all rank distributions and utilization levels, with as little as 8 queues. The best performances are obtained under low utilization and with 32 queues.

Number of queues (Fig. 2.7b) When using only 8 queues, SP-PIFO is already within ∼20–29% of the gradient-descent algorithm and the optimal mapping. With 32 queues, it gets even closer, producing only ∼22% more inversions than the optimal and achieving on-par behavior to the gradient-descent algorithm. Overall, it improves FIFO performance ∼3.3× (resp. ∼10×) when only 8 (resp. 32) queues are used.

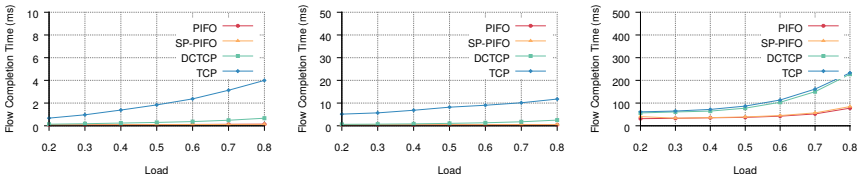
Push-down strategies (Fig. 2.7c) We evaluate four adaptation strategies for decreasing each queue bound in the push-down stage: (i) to the value of the next-higher queue bound ("Queue Bound"); (ii) by the cost of the inversion ($q_1 - r(p)$, the strategy in SP-PIFO, "Cost"); (iii) by the rank of the packet causing the inversion ("Rank"); and (iv) by 1 ("1").

The best performance is obtained for "Queue Bound", which produces ∼15% more inversions than the gradient-based algorithm. This is followed by "Cost" and "Rank", with ∼22%, and "1" with ∼33%. While the three first techniques produce similar results, the "push down" effect of "1" is too small to balance the "push up" stage, resulting in many inversions. While "Queue Bound" is marginally better than "Cost", it is more costly to implement, thus SP-PIFO uses the latter.

Utilization (Fig. 2.7d) SP-PIFO performance is close to the gradient-based algorithm. For utilizations below 60%, SP-PIFO is on-par with the gradient-based algorithm. The number of inversions slightly increases at higher utilizations: 26% and 38% for 80% and 90%.

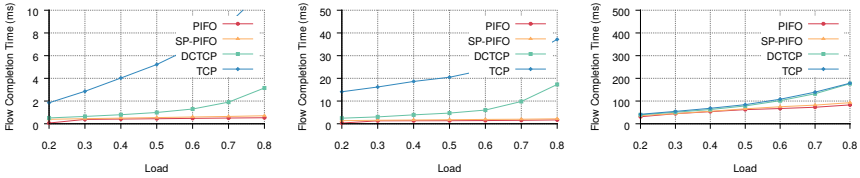
Rank distributions (Fig. 2.8) We analyze the performance of SP-PIFO under four alternative rank distributions: exponential, inverse exponential, Poisson and convex. SP-PIFO performs better than FIFO and is close to the gradient-based algorithm for each distribution.

The performance of SP-PIFO is better for rank distributions in which more ranks appear in higher-priority queues. The number of inversions for



(a) (0,100KB): Average (b) (0,100KB): 99th perc. (c) [1MB,∞): Average

FIGURE 2.9: pFabric: FCT statistics across flow sizes in data mining workload.



(a) (0,100KB): Average (b) (0,100KB): 99th perc. (c) [1MB,∞): Average

FIGURE 2.10: pFabric: FCT statistics across flow sizes in web search workload.

SP-PIFO in convex and exponential distributions is only ∼21–24% higher than the gradient-based algorithm. The corresponding numbers for Poisson and inverse exponential amount to ∼49–55%. In all cases, SP-PIFO performs between ∼2.5–3.5× better than a FIFO, with only 8 priority queues.

2.6 IMPLEMENTATION

In this section, we describe our implementation of SP-PIFO in $P4_{16}$ [28] and $P4_{14}$. Our implementation follows the algorithm described in §2.5 and spans 190 ($P4_{16}$) and 735 ($P4_{14}$) lines of code. It performs three main operations: (i) computing/extracting the rank from a packet header; (ii) mapping packets to queues (§2.2); and (iii) updating the queue bounds.

Rank computation We implemented and tested multiple rank computation functions such as LSTF [17], STFQ [20], and FIFO+ [29] in $P4_{16}$. We note that the reduced memory usage in SP-PIFO leaves room to compute ranks

directly on the switch. That said, most ranking algorithms can directly be computed by the end-hosts [17].

Mapping We store the queue-bound values in individual registers and access them sequentially using an `if-else` conditional tree. For each register access, we leverage the ALU to perform three operations: (i) we read the queue-bound value and compare it to the packet rank; (ii) we notify the queue-selection result to the control flow using a single-bit metadata; and (iii) we update the queue-bound value to the packet rank if the queue is selected. In the ALU of the last queue, we check whether an inversion has occurred before transferring the potential inversion cost using metadata.

Adaptation When the mapping process detects an inversion, we need to update all queue bounds. While accessing multiple registers is not restricted by the P4 specification [30], current architectures do not support it, to guarantee line rate. We address this problem by relying on the packet-resubmission primitive to access the queue bounds a second time and update them with the measured inversion cost. While resubmission can possibly break the line-rate guarantees, we only require it upon inversions.

Memory requirements Our implementation only requires n registers where n is the number of queues. We leverage n ALUs to access registers during the mapping process and $n-1$ additional ALUs to update registers from the resubmission pipeline in case of inversions. We use $n-1$ bits of metadata to access the mapping results of non-top-priority queues in their respective ALUs from the control flow (i.e., a single 1-bit metadata field for each queue) and an extra 32-bit field for the top-priority queue to (potentially) transfer the inversion cost.

Regarding the number of stages, our implementation uses more stages than the number of queues in order to perform the sequential access to queue-bound registers during the mapping process. Alternative designs would be possible but would come at the expense of line-rate guarantees.

2.7 EVALUATION

We now evaluate SP-PIFO performance and practicality. We first use packet-level simulations to evaluate how SP-PIFO approximates well-known scheduling objectives under realistic traffic workloads (§2.7.1). We then evaluate SP-PIFO's performance when deployed on hardware switches (§2.7.2).

2.7.1 *Performance analysis*

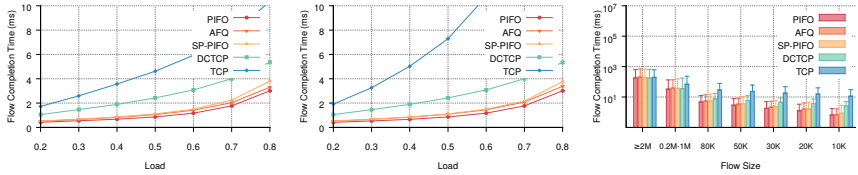
We consider two scheduling objectives: (i) minimizing Flow Completion Times (FCTs); and (ii) enforcing fairness. We consider that ranks are set at the end hosts for the former objective and computed in the switch for the latter. For both objectives, we show that SP-PIFO scheduling capabilities achieve near-optimal performance, with as little as 8 queues.

Methodology We integrated SP-PIFO in Netbench [26, 27], a packet-level simulator. Similar to [31], we use a leaf-spine topology with 144 servers connected through 9 leaf and 4 spine switches. We set the access and leaf-spine links to 1Gbps and 4Gbps, respectively. This results in a theoretical end-to-end Round-Trip-Time (RTT) of 32.12μs when crossing the spine (4 hops) and 26μs under the leaf (2 hops). We generate traffic flows following two widely-used heavy-tailed workloads: pFabric web application and data mining [31]. Flow arrivals are Poisson-distributed and we adapt their starting rates to achieve different utilization levels. We use ECMP and draw source-destination pairs uniformly at random.

Minimizing Flow Completion Times

Rank definition & benchmarks We minimize FCTs by implementing the pFabric algorithm [31] which sets the packet ranks according to their remaining flow sizes. Specifically, we compare pFabric performance when run on top of PIFO and SP-PIFO. We also analyze TCP NewReno with traditional drop-tail queues and DCTCP with ECN-marking drop-tail queues. Our pFabric implementation does not consider starvation prevention. As suggested in [31], we approximate pFabric rate control by using standard TCP with a retransmission time-out of 3 RTTs, balancing the difference in RTOs between schemes with the proportional queue size. That is, we use an RTO of 96μs and 8 queues$\times$10 packets for SP-PIFO (resp. 1 queue$\times$80 packets in PIFO), and an RTO of 300μs and 146KB drop-tail queues for both TCP and DCTCP, with ECN marking at 14.6KB, i.e. $\sim$10 packets.

Summary Fig. 2.9 and Fig. 2.10 depict the average and 99th percentile FCTs of large ($\geq$ 1MB) and small flows ($<$ 100KB) for both data mining and web search workloads. We see that SP-PIFO achieves close-to-PIFO performance in both distributions. When comparing performance across flow sizes, we see that SP-PIFO achieves better performance for small flows. This is



(a) (0,100KB): Average on 8 queues (b) (0,100KB): Average on 32 queues (c) FCT breakdown 70%: Average on 32 queues

FIGURE 2.11: Fairness: FCT statistics for all flows over the web search workload.

not surprising since those flows are mapped into higher-priority queues. As discussed in §2.5.3, strict-priority schemes provide higher unpifoness protection for packets mapped into higher-priority queues.

When comparing the two traffic distributions, we see that SP-PIFO performs better under the data mining workload. This is again expected. While both distributions are heavy-tailed, the data mining one is more skewed [31] and therefore easier to handle for SP-PIFO. Indeed, the probability of having large flows simultaneously sharing the same port (potentially blocking smaller flows) is lower for the data mining workload.

Data mining (Fig. 2.9) The average FCTs achieved by PIFO and SP-PIFO are similar for small flows, i.e. within ∼0.4–11%. Concretely, SP-PIFO outperforms DCTCP and TCP by a factor of ∼2–5× and ∼8–30×, respectively. When considering the 99th percentile, the gap between PIFO and SP-PIFO slightly accentuates to ∼9.6–26.6%. Still, SP-PIFO outperforms DCTCP and TCP by a factor of ∼1.5–4.7× and ∼12.5–22×, respectively. The largest performance gap between PIFO and SP-PIFO occurs at low utilization. In this regime, the number of packets scheduled is low and the transient adaptation of SP-PIFO is more visible. Whenever the utilization is >40%, the difference is consistently below 20%. Finally, SP-PIFO and PIFO still perform similarly among large flows: within ∼1.9–9%, with improvements of ∼1.4–2.7× and ∼1.5–2.8× respect to TCP and DCTCP.

Web search (Fig. 2.10) The results are similar to the data mining one, with slightly worse performance for SP-PIFO, especially amongst big flows. Indeed, since the distribution is less skewed, bigger flows have higher chances to reach higher-priority queues, blocking transmissions of smaller flows. Still, we see that the performance of SP-PIFO is within ∼16.54–32.5% of PIFO for small flows, and between ∼1.3–4.4× and ∼4.7–16.7× better

than DCTCP and TCP. Even at the 99th percentile, the difference between SP-PIFO and PIFO stays within ∼20.7–32%. Note that, while the percentages might seem high, the values we are looking at are very small.

Enforcing fairness across flows

Rank definition & benchmarks We enforce fairness across flows by implementing the Start-Time Fair Queueing (STFQ) rank design [32] on top of PIFO and SP-PIFO. We benchmark our solution with AFQ [33] (§2.9). We analyze the performance for different flow sizes and number of queues. Specifically, we use 8 queues×10 packets in SP-schemes (resp. 1 queue×80 packets for single-queue schemes) and 32 queues×10 packets in SP-schemes (resp. 1 queue×320 packets for single-queue schemes). For AFQ, we select the bytes-per-round parameter which gives the best performance. In our testbed, this is 320 and 80 BpR for the 8-queue and 32-queue scenario, respectively. As in [33], we use DCTCP as transport layer for AFQ, PIFO and SP-PIFO (with an RTO of 300μs). We set ECN marking to 48KB, i.e. ∼32 packets. We generate traffic following the pFabric web search distribution.

Summary Fig. 2.11a and Fig. 2.11b depict the average FCTs of small flows across different levels of utilization, when 8 queues and 32 queues are used. Fig. 2.11c depicts the FCTs across flow sizes at 70% utilization and for 32 queues. In all cases SP-PIFO achieves near-PIFO behavior and is on-par performance with AFQ (current state-of-the-art).

Impact of the utilization (Fig. 2.11a & Fig. 2.11b) SP-PIFO stays within ∼23–28% (resp. ∼21–28%) of ideal PIFO across all levels of utilization when 8 queues (resp. 32) are used. Even in the highest utilizations, it is consistently below ∼26% (resp. ∼25%). SP-PIFO performance is at the level of AFQ, within ∼3–10% (resp. ∼0.5–11%), generating improvements of ∼1.4–2.3× and ∼2.7–4.2× (resp. ∼1.4–2.3× and ∼3.7–7.4×) over DCTCP and TCP. The fact that SP-PIFO performance is equivalent with 8 and 32 queues is not surprising: as the bandwidth-delay product is low, only a reduced queue size is required for efficient link utilization.

Impact of flow sizes (Fig. 2.11c) At 70% utilization, we see that SP-PIFO lies within ∼10–30% of PIFO performance for all flow sizes and is on-par with AFQ. The only exception is for very small flows (<10K) in which AFQ performs 20% better. SP-PIFO improves DCTCP and TCP behaviors for small flows, within ∼1.5–3X and ∼2–13X, respectively. Considering the 99th

percentile, we see that SP-PIFO stays within ∼8–35% of PIFO and improves between ∼12–78% and ∼1.5–10.76× with respect to DCTCP and TCP.

Impact of the number of queues (Fig. 2.12) We analyze the impact of the number of queues on average FCTs for both AFQ and SP-PIFO. We set the BpR at MSS for all queue configurations, as in [33], avoiding AFQ dropping packets too often for cases of fewer queues. We see that while AFQ has a higher sensitivity with respect to the number of queues, SP-PIFO achieves a similar performance, without any configuration or prior traffic knowledge.

2.7.2 *Hardware testbed*

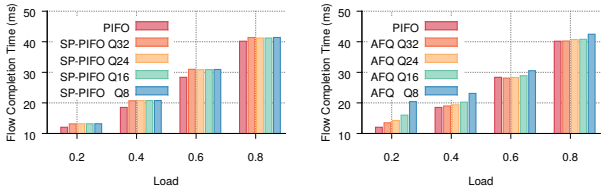
We finally evaluate our hardware-based implementation of SP-PIFO on the Intel Tofino Wedge 100BF-32X platform [21]. We perform two experiments. First, we analyze the bandwidth allocated by SP-PIFO to flows with different ranks when scheduled over a bottleneck link. Second, we measure the impact on the FCT when SP-PIFO runs pFabric. We show that SP-PIFO efficiently schedules traffic at potentially Tbps.

Bandwidth shares We transmit 8 UDP flows of 20Gbps between two servers. We generate the flows progressively, in increasing order of priority (decreasing rank). We use 4 priority queues and schedule the flows over a 10Gbps interface. We generate the flows using Moongen [34] and use an intermediate switch to amplify them to the required throughput.

Fig. 2.13 depicts the flows' bandwidth and how SP-PIFO manages to virtually extend the number of queues. As expected, the first 4 flows receive the complete bandwidth, since they are mapped to dedicated queues. As the number of flows exceeds the number of queues, flows start to share queue space and see a reduced bandwidth.

Flow completion times We simultaneously generate 1000 TCP flows of different sizes, going from 1GB to 100GB in steps of 100MB, and schedule them over a bottleneck link of 7Gbps. We set the rank of each flow to the absolute flow size, following [31]. We compare the FCTs achieved by SP-PIFO scheduling and the ones achieved by a FIFO queue.

Fig. 2.14 shows the resulting FCTs. As expected, the FIFO queue leads to increased FCTs by not considering flow size. In contrast, SP-PIFO prioritizes short flows over long ones, minimizing their FCTs and the overall transmission time.



(a) All: Average (b) All: Average

FIGURE 2.12: Fairness: FCT statistics for all flows at different loads, when the number of queues is modified.

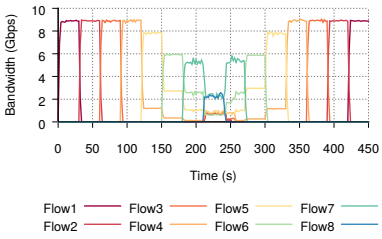


FIGURE 2.13: Tofino: Bandwidth allocation under progressive flow generation with increasing priorities.

2.8 DISCUSSION

We now discuss the limitations of SP-PIFO and how we can mitigate them. We first discuss intrinsic limitations that come from using PIFO as a scheduling scheme. We then discuss specific limitations of SP-PIFO together with the problem of adversarial workloads. Finally, we suggest potential hardware primitives that could facilitate PIFO implementations in the future.

PIFO-inherited limitations Individual PIFO queues suffer from two main limitations. First, they cannot rate-limit their egress throughput preventing them from implementing non-work-conserving scheduling algorithms. SP-PIFO also shares the same limitation. Second, PIFO queues cannot directly implement hierarchical scheduling algorithms. Yet, as proposed by [20], multiple SP-PIFO schemes (i.e., using different set of priority queues) can be grouped as a tree to approximate hierarchical scheduling algorithms. The key challenge consists in figuring out how to allow access of multiple queues with existing traffic manager capabilities. While this is orthogonal

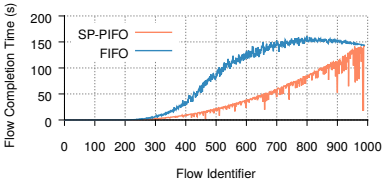


FIGURE 2.14: Tofino: FCT statistics across different flow sizes with pFabric ranks.

to this chapter, one option would be to recirculate packets, enabling access to the traffic manager (and therefore the queues) multiple times in the same pipeline. Doing so, while limiting the impact on performance, is an interesting open question.

SP-PIFO-specific limitations The main limitation of SP-PIFO is that, as an approximation scheme, it cannot guarantee to perfectly emulate the behavior of a theoretical PIFO queue for all ranks. We note two things. First, our evaluation (§2.7) shows that, for realistic workloads, SP-PIFO performance is often on-par with PIFO performances. Second, we note that SP-PIFO *can* provide strong PIFO-like guarantees *for some ranks* by dedicating some queues to them at the price of reduced performance for the other ranks. We believe this is an interesting tradeoff as current switches can support up to 32 queues per port [33].

Adversarial workloads We have argued that, on average, SP-PIFO can adapt to any kind of rank distribution. This has certain limitations. First, we assume that all queues are drained at some point. Nonetheless, a malicious host could send a large number of high-priority packets and, as a result, packets in lower-priority queues would never be drained. Such "starvation" attacks are common to any type of priority scheme. For instance, a malicious host could try to grab a bigger slice of the network resources by setting ranks to 0 in slack-based algorithms [17, 29, 31] or resetting flow identifiers in fair-queuing schemes [20]. The problem of starvation in strict-priority scheduling is well-known in the context of QoS and is typically addressed by policing high-priority traffic at the network's edge [35].

Aside from starvation attacks we also assume that, for a given rank distribution, the particular order of ranks is random. In practice, this is reasonable. While the ranks for individual flows might have some structure (e.g., monotonically-increasing ranks), when various flows are scheduled together the ordering of their packet ranks is randomized. Yet, attackers

could try to coordinate large numbers of flows to create adversarial orderings, which "outplay" the adaptation mechanisms (§2.5.2). Nevertheless, any non-malicious flow which is active at the same time can thwart such strategies by randomly breaking the adversarial order. Aside from that, the network could be monitored to detect such adversarial attacks.

Facilitating PIFO in the future On a forward-looking perspective, we note some improvements in hardware primitives that would facilitate PIFO implementations in the future. As we already discussed in §2.6, a higher number of stages would facilitate per-queue state storage and a higher number of queues would directly increase PIFO performance. Further than that, multiple and dynamic memory access between the ingress and egress pipelines would allow state updates after inversions in the highest-priority queue without having to rely on resubmission techniques. In the same direction, access to queue information from the ingress pipeline or an enhanced flexibility in the management of strict-priority queues directly from the data plane would enable more accurate unpifoness prediction at enqueue, opening the doors to higher-performance SP-PIFO algorithms.

2.9 RELATED WORK

Programmable packet scheduling While scheduling has been extensively studied over the years, the idea of making it programmable is recent [18]. Sivaraman et. al. introduced it by proving that the best scheduling algorithm to use depends on the desired performance goal, and proposed the PIFO abstraction as an enabler [18, 19]. In [17], Mittal et. al. also showed that a universal packet scheduling outperforming in all scenarios does not exist, and that least-slack-time first (LSTF) algorithm can be configured to approximate a wide range of objectives. Eiffel [36] presents an alternative queue structure that approximates fine priorities by exploiting the characteristics that define packet ranks in most scenarios. In contrast to [18, 36], which rely on new hardware designs, SP-PIFO shows that efficient programmable packet scheduling can be achieved today, at scale, and on existing devices.

Exploiting priority queues Other schemes leverage multiple priority queues for specific performance objectives. They highlight the need for programmable scheduling in existing devices [37], and illustrate how rank designs producing near-optimal results can already be implemented in existing data planes. For enforcing fairness, FDPA [38] simplifies the computational cost

of per-flow virtual counters or individual user queues in traditional-fair-queuing schemes by using arrival-rate information at a user level. AFQ [33], instead, simulates ideal fair queuing by implementing per-flow counters on a count-min sketch and dynamically rotating priorities in a strict-priority scheme to imitate the round-robin behavior. In contrast, SP-PIFO fixes queue priorities and dynamically adapts the mapping of packets to queues. This makes SP-PIFO implementable *at line rate* in existing data planes.

pFabric [31] and PIAS [39] use priority queues to minimize flow completion times. While pFabric relies, in general, on a PIFO-queue design, [31] includes experiments in which flows are mapped to priority queues based on their size. PIAS [39] addresses the case of unknown flow sizes by using Multi-level Feedback Queues (MLFQ) [40] to achieve the desired Shortest Job First (SJF) behavior. It gradually switches flows from higher to lower-priority queues as their transmitted byte count increases.

In contrast to these proposals, SP-PIFO supports a much wider range of performance objectives. SP-PIFO (like PIFO [18]) can implement *any* scheduling algorithm in which the relative scheduling order does not change with future packet arrivals. As illustrated in the evaluation section (§2.7), the algorithms presented in AFQ [33], FDPA [38], pFabric [31] and PIAS [39] can be used as ranking designs (i.e., setting packet ranks to scheduling virtual rounds, estimated arrival rates, shortest remaining processing time of flows, or number of packets transmitted) to be run on top of SP-PIFO.

2.10 CONCLUSIONS

In this chapter, we introduced SP-PIFO, a programmable packet scheduler which closely approximates the scheduling behavior of PIFO queues, today, on programmable data planes. The key insight behind SP-PIFO is to dynamically adapt the mapping between the packet ranks and a set of strict-priority queues. Our evaluation on realistic workloads shows that SP-PIFO is practical: it closely approximates PIFO behaviors and, in some cases, perfectly matches them. Our evaluation also confirms that SP-PIFO runs on programmable hardware. Overall, this work shows that the benefits of programmable packet scheduling—experimenting with new scheduling algorithms—can be fulfilled today, on existing programmable data planes.

3

PACKS

3.1 INTRODUCTION

In the previous chapter, we introduced SP-PIFO, a programmable packet scheduler designed to run on existing programmable data planes. After we presented SP-PIFO in 2020, multiple works followed its path, exploring the possibilities of approximating PIFO behaviors on existing switches [1, 41–44]. We quickly realized that both, SP-PIFO, and its follow-up works, shared a significant limitation: they were only approximating one of the two key necessary behaviors of PIFO queues, either their *scheduling behavior* or their *admission control* (Fig. 3.1). In order to approximate PIFO queues fully, we needed to *simultaneously* capture both of its behaviors.

PIFO queues sort packets at line rate, thanks to two key functions: (i) they always *admit* packets with the lowest ranks; and (ii) they *schedule* packets in perfect order of rank. For example, PIFO queues can "push" incoming low-rank packets before higher-rank packets that are already in the queue, even dropping the higher-rank packets (if needed) to accommodate the newly arrived low-rank ones.

SP-PIFO [1], QCluster [43], AFQ [33], PCQ [42], and Gearbox [44], only approximate PIFO's *scheduling behavior*. They map incoming packets to priority queues to minimize the rank inversions at the output. However, they do not actively control packet drops, which they leave as a byproduct effect of the schedulers' design. As such, even though these schedulers can support a broad variety of scheduling algorithms, their behavior can have a negative impact for loss-sensitive applications (cf. §3.2).

FIGURE 3.1: PACKS navigates the space between SP-PIFO [1] and AIFO [41], optimizing for both rank ordering and drops.

AIFO [41] only approximates PIFO's *admission behavior*: it executes a rank-aware admission-control policy on top of a FIFO queue that drops incoming packets imitating a PIFO queue. But because it runs on a single FIFO queue, AIFO cannot prioritize packets based on their ranks, which limits the scheduling algorithms that it can accurately approximate.

Our work In this chapter, we introduce PACKS, a programmable PACKet Scheduler that approximates *both* the admission and scheduling behaviors of a PIFO queue on programmable hardware. PACKS runs on top of a set of strict-priority queues and combines an admission-control mechanism with a queue-mapping mechanism. Since priority queues cannot drop nor modify the order of enqueued packets, PACKS emulates the behaviors that a PIFO queue follows and executes them at enqueue.

Key insights PACKS derives its admission and queuing decisions from two key sources: the rank distribution of the last packets received (monitored via a sliding window) and the real-time buffer occupancy of each queue. PACKS integrates this data into a quantile-based admission and queue-mapping process that prioritizes packets with lowest expected ranks.

PACKS's *rank-aware* approach allows it to minimize rank inversions and outperforms existing queue-mappers that assume no prior rank knowledge and rely on per-packet heuristics. PACKS's *queue-occupancy-aware* approach ensures efficient resource utilization and reduces packet drops.

Evaluation We implement PACKS in P4 and evaluate it on real workloads and in hardware. Our results under mixed flow scenarios show that, PACKS consistently outperforms in approximating PIFO's admission behavior and reduces the rank inversions by up to 7× and 15× with respect to SP-PIFO, and AIFO, and the number of packet drops with respect to SP-PIFO by up to 60%. Under pFabric ranks, PACKS reduces the average FCT across small flows by up to 33% and 2.6× with respect to SP-PIFO and AIFO.

Contributions The main contributions of this chapter are:

- PACKS, a programmable scheduler that emulates PIFO queues on top of a set of strict-priority queues (§3.3).
- An admission-control algorithm and a queue-mapping technique that approximate all PIFO behaviors (§3.4).
- A theoretical analysis of PACKS's optimality (§3.5).

- A performance analysis of PACKS on MetaOpt [45] to study its performance gaps and adversarial inputs (§3.6).
- An implementation of PACKS in Java and P4 (§3.7).
- An evaluation showing PACKS's effectiveness in approximating PIFO using simulations and hardware (§3.8).

3.2 BACKGROUND

We first revisit the core insights of SP-PIFO [1] (§3.2.1) and introduce AIFO [41] (§3.2.2). These two programmable packet schedulers serve as examples of how prior research approximate PIFO's scheduling and admission control behavior, respectively (Fig. 3.1). We then motivate PACKS based on where these schedulers fall short (§3.2.3).

3.2.1 *SP-PIFO*

As introduced in Chapter 2, SP-PIFO [1] approximates PIFO's *scheduling behavior* (i.e., forwarding the earliest-arrived lowest-rank packet first) on a set of strict-priority queues. It adapts the mapping between packet ranks and priority queues dynamically to minimize the number of rank inversions (i.e., the times a higher-rank packet is scheduled before a lower-rank one).

Mapping SP-PIFO maps incoming packets to queues based on the queue *bounds*, which define the lowest rank that the scheduler can admit into each queue. Whenever SP-PIFO receives a packet, it scans the queue bounds from lowest to highest priority, and maps the packet to the first queue with a bound lower or equal to the packet rank.

Adaptation SP-PIFO uses two mechanisms to adapt queue bounds dynamically: a *push-up* stage where it pushes future low-rank (i.e., high-priority) packets to higher-priority queues; and a *push-down* stage where it pushes future high-rank (i.e., low-priority) packets to lower-priority queues. The push-up stage occurs whenever a packet is admitted into a queue. Then, SP-PIFO updates the queue's bound to the rank of the new packet. In the push-down stage, SP-PIFO decreases the queue bounds of *all* queues when it detects a rank inversion in the highest priority queue. With these two mechanisms, SP-PIFO spreads packet ranks across queues, reduces rank inversions, and approximates PIFO's scheduling behavior.

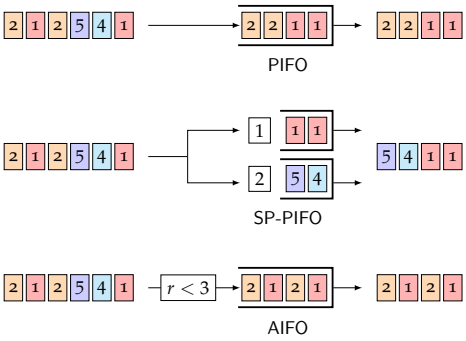


FIGURE 3.2: SP-PIFO and AIFO cannot fully approximate PIFO.

3.2.2 *AIFO*

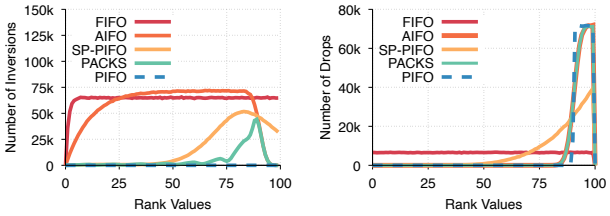
AIFO [41] approximates PIFO's *admission behavior* (i.e., only admitting the earliest-arrived lowest-rank packets) on a FIFO queue. It maintains a sliding window of the most recent ranks and it decides whether to admit each incoming packet based on the packet's rank and the buffer-occupancy level.

Admission AIFO uses the distance of the packet rank to the rank of the packets already in the queue and the time-discrepancy between the incoming and outgoing rate of the FIFO queue to admit packets. It increases the probability of dropping a packet as the distance between it's rank and that of the recently-admitted packets increases; and it increases the probability of dropping packets as the space available in the FIFO queue decreases (the reduction of space in the queue indicates the incoming rate is higher than the outgoing rate).

3.2.3 *Limitations*

We analyze the limitations of existing schedulers and motivate the need for PACKS with an example and a simple experiment.

Example Fig. 3.2 shows how PIFO, SP-PIFO and AIFO serve the packet sequence 2 1 2 5 4 1 (first packet on the right). All schedulers have capacity for 4 packets. SP-PIFO has two priority queues of two packets each, and fixed bounds with values of 1 and 2 for the highest- (resp. lowest-) priority queue. AIFO has a fixed admission control that admits ranks $r < 3$.



(a) Scheduling inversions (b) Packet drops

FIGURE 3.3: Scheduling performance, uniform rank distribution.

PIFO "pushes" the first four packets into the queue according to their rank order: 5 4 2 1. When the fifth packet arrives (1), PIFO "pushes" it into the queue between packets with ranks 1 and 2 and drops the highest-rank packet in the queue (5). When the last packet arrives (2), PIFO "pushes" it between packets of rank 2 and 4 and drops packet 4. PIFO's outgoing sequence is therefore 2 2 1 1.

SP-PIFO maps packets 1 1 to the highest-priority queue, and packets 2 2 5 4 to the lowest-priority queue (c.f., §3.2.1). Since the lowest-priority queue only has room for two packets, it drops the last packets to arrive (2 2). The output sequence is 5 4 1 1, which has sorted ranks (it approximates PIFO's scheduling), but does not contain the packets with rank 2 that PIFO accepted (it fails to approximate PIFO's admission).

AIFO admits the packets with rank $r < 3$, same as PIFO. However, since it runs on top of a FIFO queue, it cannot prioritize packets, which results in an output sequence not sorted by rank (i.e., 2 1 2 1 instead of 2 2 1 1).

PACKS predicts the arrival of packets with rank 2, discards those with ranks 4 and 5, and sorts admitted packets across priority queues — it achieves the optimal output (cf. Fig. 3.5).

Experiment These limitations generalize across ranks. We implement SP-PIFO, AIFO, PACKS and FIFO in Netbench [26, 27], and schedule a stream of packets over a bottleneck link where the ranks are distributed uniformly across $[0, 100]$ (details in §3.8). We measure the priority inversions generated by each rank and the number of packet drops per rank.

PIFO never causes inversions and schedules packets in perfect order (Fig. 3.3a). SP-PIFO approximates this behavior, especially for lower-rank packets: it maps packets with lower ranks to higher-priority queues. AIFO

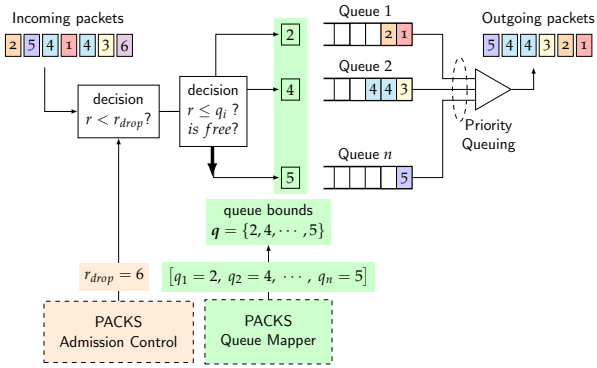


FIGURE 3.4: Overview of PACKS data-plane pipeline.

and FIFO generate a high number of inversions across most ranks because they run on a single queue and cannot prioritize lower-rank packets.

PIFO only drops packets with the highest ranks and has the best performance (Fig. 3.3b) since it prioritizes low-rank packets. AIFO closely approximates PIFO's behavior and pro-actively drops the highest-rank packets. SP-PIFO leaves drops as a by-product effect of its design (it drops higher-rank packets more often because they are mapped to lower-priority queues that drain less frequently) and performs poorly. FIFO drops packets across all ranks due to its tail-drop policy and has the worst performance.

PACKS's behavior is closest to PIFO in *both* scheduling inversions and packet drops since it combines the best of both worlds: an admission-control scheme, like AIFO, and a queue-mapping scheme, similar to SP-PIFO.

The inefficiencies of existing works in approximating PIFO behaviors ultimately lead to performance degradation. Not exempting low-priority packets from occupying buffer space when high-priority ones are present, or not sorting packets by rank, results in increased latency, reduced bandwidth for priority applications, and longer flow completion times (§3.8.2).

3.3 OVERVIEW

We now provide an overview of how PACKS approximates the behavior of a PIFO queue on existing hardware. PACKS runs on top of a set of strict-priority queues, and incorporates: (i) an *admission-control* mechanism

that decides which packets to admit, and (ii) a *queue mapper* that decides how to map admitted packets to the different priority queues (see Fig. 3.4).

PACKS uses this setup to approximate two PIFO behaviors: it *admits* packets with the lowest ranks; and *schedules* packets in order of their rank. What enables PIFO to achieve these behaviors is that it can map packets to any position in the queue, and it can drop packets (based on their ranks) even after it has admitted them into the queue. But we do not have this functionality (by default) on existing hardware.

PACKS approximates these behaviors by *predicting* the distribution of packets that will arrive in a given scheduling interval. It then uses this information to compute the admission and scheduling decisions that a PIFO would follow, and approximates these actions at each packet's arrival. With a given rank distribution, PACKS predicts the set of expected lowest-rank packets that fit into the available buffer space and it proactively drops all the arriving packets that have a higher rank than this prediction. PACKS also estimates how should it map admitted packets to each queue to approximate the correct rank order at the output and executes this mapping.

Rank-distribution estimation PACKS uses a sliding window to (dynamically) monitor the distribution of the ranks of recently arrived packets. It assumes that the distribution of latest-enqueued packets offers a good estimate of the one of incoming packets that are expected to arrive.

Admission control PACKS uses the distribution it estimates (see above) to predict which packets it should admit into the queue. Intuitively, PACKS should only admit the packets with the lowest ranks that can fit in the available buffer space (to mimic PIFO). Whenever a packet arrives, PACKS measures the available buffer space (as a percentage of the total buffer space) and computes a rank r_{drop} that represents a threshold such that all packets with rank $r \geq r_{drop}$ should be dropped. This rank is the lowest rank for which the quantile of the rank distribution exceeds the percentage of the remaining buffer space. This policy ensures that PACKS only admits the *lowest-rank* packets that it *expects* to arrive and fit in the available buffer space, which emulates PIFO's admission behavior.

Queue mapping PACKS then uses its estimate of the rank distribution to find the best mapping of expected packets to priority queues, to maximize the rank order at the output of the scheduler. Intuitively, the best mapping assigns packets with lower-ranks to the higher-priority queues (to prioritize low-rank packets) and minimizes the number of different-rank packets

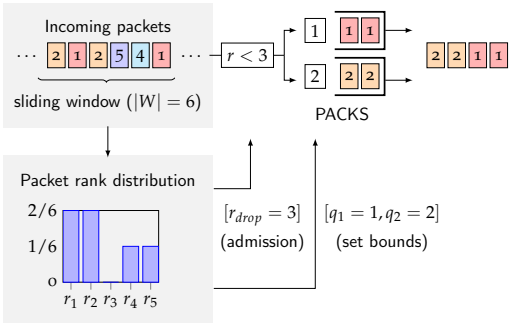


FIGURE 3.5: PACKS closely approximates PIFO's behavior.

assigned to the same queue (to reduce the probability of higher-rank packets arriving before lower-rank ones, thereby generating a rank inversion).

PACKS defines a set of rank values $\boldsymbol{q} = (q_1, ..., q_n)$ that drive how it maps packets to priority queues (in the same way that r_{drop} drives admission control). The queue bound q_i for each queue describes the highest rank that the scheduler can admit to the queue such that these packets (i.e. those with rank $r < q_i$) are the lowest rank packets that fit in the available queue space. PACKS scans queue bounds top-down (i.e. from highest- to lowest-priority) and maps each incoming packet to the first queue where the packet rank is lower or equal to the queue bound — in this way it maps the low-rank packets to the high-priority queues: PACKS prioritizes *expected* packets of low rank over higher-rank ones (similar to PIFO).

Example Fig. 3.5 shows how PACKS schedules the sequence 2 1 2 5 4 1. We assume the sequence repeats, and configure PACKS with two priority queues of two packets each and a sliding window of size $|W| = 6$. After receiving the 6-th packet, PACKS has estimated the rank distribution, where the probability of receiving a packet of ranks 1 to 5 are $p(1) = 2/6$, $p(2) = 2/6$, $p(3) = 0$, $p(4) = 1/6$, $p(5) = 1/6$. Given the available buffer space (i.e., 4 packets), and based on the monitored rank distribution, PACKS sets r_{drop} to 3, since the expected 4 packets with lowest rank are those with rank 1 and 2. Then, PACKS sets q_i based on the available buffer space at each queue (i.e., 2 packets each). As such, it sets $q_1 = 1$ to map the two expected packets with lowest rank to the highest-priority queue, and $q_2 = 2$, to map the two expected admitted packets with highest rank to the lowest-priority queue. As a result, the output sequence of PACKS is 2 2 1 1, the exact same one as in the PIFO queue (see Fig. 3.2).

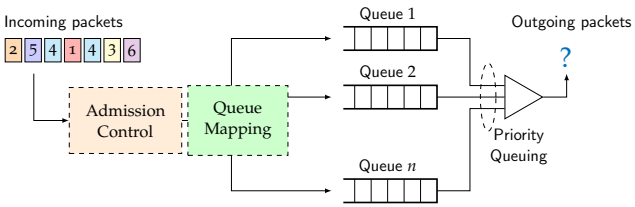


FIGURE 3.6: PACKS's design space.

3.4 PACKS DESIGN

We now describe the theoretical basis supporting the design of PACKS. First, we frame the problem and introduce the design space (§3.4.1). Second, we provide the high-level intuition behind PACKS's design by studying the case in which it schedules a batch of packets (§3.4.2). Third, we generalize the algorithm to the online setup (§3.4.3). Finally, we formalize the PACKS's algorithm (§3.4.4), and analyze it both theoretically and with MetaOpt [45], an heuristic analyzer (§3.4.5).

3.4.1 *Design space*

Let us consider the scheduling design space in Fig. 3.6, which represents the available resources in existing data planes [21, 46]. Packets arriving at the scheduler are already tagged with ranks, either specified by the end hosts or at prior stages of the switch. The scheduler is composed by a set of strict-priority queues of fixed sizes, an *admission-control* mechanism that decides which packets to admit, and a *queue mapper* that decides how to map admitted packets to the priority queues. [1] After a packet is mapped to a queue, it is enqueued *only* if the queue has sufficient buffer space; otherwise, the packet is dropped. The scheduler continuously drains queues in decreasing order of priority, scheduling packets from low-priority queues only when higher-priority queues are empty, and schedules packets within each priority queue in a first-in first-out fashion.

1 Some devices allow extra functionalities such as flexible priority-queue configuration, round-robin scheduling, or buffer management. We use Fig. 3.6's abstraction for generality and to guarantee line-rate processing for *all* packets.

Problem *How can we best approximate the behavior of a PIFO queue on top of the PACKS abstraction in Fig. 3.6?*

The PACKS abstraction only allows for two design decisions: an admission-control and a queue-mapping algorithm. Our objective is to design such two mechanisms in a way that their overall behavior approximates the one of a PIFO queue. This is, an admission-control mechanism that (ideally) admits the earliest-arrived lowest-rank packets, and a queue-mapping algorithm that (ideally) prioritizes packets with lower rank.

3.4.2 *High-level intuition*

We introduce the high-level intuition behind PACKS's design by analyzing the case in which a PIFO queue schedules a batch of A packets. We assume, for now, that all packets have the same size, and that the PIFO queue has a capacity of B packets. For each incoming packet, the PIFO queue decides whether to admit or drop the packet. Only after processing *all* the packets, the PIFO queue schedules the admitted packets.

Approximating PIFO's admission In this setup, the PIFO queue admits the B (earliest-arrived) lowest-rank packets to the buffer, dropping the rest. Considering the rank distribution of the packets in the batch, $\mathcal{W}$, the admitted packets are the first B packets that we find when reading the distribution from left to right (see Fig. 3.7). As such, we can define a rank r_{drop}, such that all packets with rank equal or higher than r_{drop} are dropped by PIFO. Formally, computing r_{drop} is finding the highest rank in the distribution, for which the quantile of the distribution is below the fraction of the available buffer, B/A:

$$\begin{aligned} &\text{maximize } r_{drop}, \text{ where } 0 \leq r_{drop} \leq R, \\ &\qquad s.t., \ \mathcal{W}.quantile(r_{drop} - 1) \leq B/A \end{aligned} \tag{3.1}$$

Once we know r_{drop}, approximating PIFO's admission on top of the PACKS abstraction is straightforward: we just have to configure PACKS's admission-control to drop all incoming packets from the batch with ranks higher or equal than r_{drop}.

So far, our model assumes that PIFO treats all packets with the same rank equally (i.e., either admitting or dropping them). In practice, however,

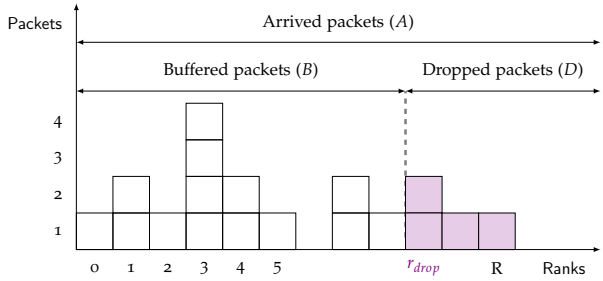


FIGURE 3.7: Admission control for a rank distribution, $\mathcal{W}$.

since the PIFO queue has limited size, PIFO may only admit the earliest-arrived subset of them. To support this behavior, we extend the model by defining a time, t_{drop}, above which PIFO drops all the packets of the highest-admitted rank (i.e., $r_{drop} - 1$). We can approximate this behavior on the PACKS abstraction by configuring its admission-control mechanism to drop packets based on both, r_{drop} and t_{drop}. Specifically, PACKS should drop packets if $r \geq r_{drop}$ or if $\{r = r_{drop} - 1$ and $t \geq t_{drop}\}$.

Approximating PIFO's scheduling Once PIFO has decided whether to admit or drop each packet, it schedules the B buffered packets in a earliest-arrived, lowest-rank-first fashion. This requires packet sorting at line rate. We can approximate this behavior in the PACKS abstraction using priority queues. When the number of queues is greater or equal than the number of ranks, we can perfectly sort packets by mapping packets of each rank to a different priority queue. When this is not the case, we can still aim for a good approximation [1].

For each priority queue, i, we define a rank, q_i, such that we only admit to the queue packets with rank lower or equal than q_i (c.f., Fig. 3.8). We call these ranks *queue bounds*. Formally, we let $\boldsymbol{q} = (q_1, \ldots, q_n) \in \mathbb{Z}^n$ be the set of bounds for queues 1 to n. We define a mapping strategy that uses queue bounds to map packets to their *highest-possible* priority queue, based on their rank. For each incoming packet with rank r, we scan queues top-down (i.e., from highest- to lowest-priority) and map the packet to the first queue, i, that satisfies $r \leq q_i$.[2] With this definition, we convert the problem of sorting packets at line rate based on their ranks to the problem of finding the optimal queue bounds that maximize rank order at the output of the

2 Note that PACKS scans queues top-down, while SP-PIFO does so bottom-up [1].

scheduler. We define a loss function $\mathcal{U}_{\mathcal{S}} : \mathrm{R}^n \times \mathrm{R}_{\geq 0} \rightarrow \mathrm{R}_{\geq 0}$, which stands for *scheduling unpifoness*, such that $\mathcal{U}_{\mathcal{S}}(\boldsymbol{q}, r)$ quantifies the approximation error of scheduling a packet with rank r based on queue bounds $\boldsymbol{q}$ compared to an ideal PIFO queue. Intuitively, it estimates the probability that a packet with rank r is scheduled after a packet with higher rank, r'. In the PIFO queue, $\mathcal{U}_{\mathcal{S}} = 0$, since packets are scheduled in perfect order. Thus, in the PACKS abstraction, a lower $\mathcal{U}_{\mathcal{S}}$ leads to a better approximation.

Our goal is to find the optimal queue bounds, $\boldsymbol{q}^*_{\mathcal{S}}$, that minimize $\mathcal{U}_{\mathcal{S}}$. Let $\mathcal{Q}$ be the space of all valid queue-bound vectors and $\mathcal{W}$ the distribution of packet ranks. Then, $\boldsymbol{q}^*_{\mathcal{S}}$ are:

$$\boldsymbol{q}^*_{\mathcal{S}} = \arg\min_{\boldsymbol{q} \in \mathcal{Q}} \mathcal{U}_{\mathcal{S}}(\boldsymbol{q}, r) \tag{3.2}$$

Given that queue bounds are *fixed* during the enqueue process, scheduling errors cannot occur between ranks mapped to different priority queues. Thus, we can compute the total scheduling unpifoness as the sum of the individual losses at each priority queue. Letting $\mathcal{U}_{\mathcal{S}}(q_i)$ be the loss function corresponding to the queue with bound q_i, this is:

$$\mathcal{U}_{\mathcal{S}}(\boldsymbol{q}, r) = \sum_{1 \leq i \leq n} \mathcal{U}_{\mathcal{S}}(q_i) \tag{3.3}$$

Finally, letting $p_{\mathcal{W}}(r)$ and $p_{\mathcal{W}}(r')$ be the probability of ranks r and r', respectively, both mapped to the queue i, we can define the scheduling unpifoness of the queue as:

$$\mathcal{U}_{\mathcal{S}}(q_i) = \sum_{\substack{q_{i-1} < r \leq q_i \\ r < r' \leq q_i}} p_{\mathcal{W}}(r) \cdot p_{\mathcal{W}}(r') \tag{3.4}$$

With this formulation, given that we know the exact rank distribution, $\mathcal{W}$, we can easily compute the optimal queue bounds, $\boldsymbol{q}^*_{\mathcal{S}}$. For instance, [47] proposes an algorithm that does so in polynomial time.

To provide a high-level intuition about the optimal queue bounds, we derive an upper-bound of $\mathcal{U}_{\mathcal{S}}(q_i)$ by setting $p_{\mathcal{W}}(r') = 1$. In doing so, we assume the worst case scenario in which, for each rank r, there is always a higher-rank packet, r' in the queue that can produce a scheduling error. As such:

$$\begin{aligned} \hat{\mathcal{U}}_{\mathcal{S}}(q_i) &= \sum_{q_{i-1} < r \leq q_i} p_{\mathcal{W}}(r) \\ &= \mathcal{W}.quantile(q_i) - \mathcal{W}.quantile(q_{i-1}) \end{aligned} \tag{3.5}$$

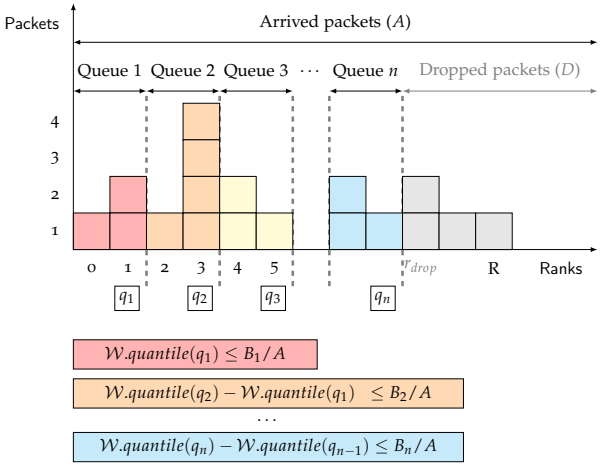


FIGURE 3.8: Queue mapping for a rank distribution, $\mathcal{W}$.

We can see how the optimal bounds are those that minimize the quantiles of the rank distribution for the set of ranks mapped to each priority queue. In other words, the optimal bounds are those that result in the least amount of different-rank packets mapped to each queue (i.e., those that minimize the colored area within each priority queue in Fig. 3.8).

Since we have to map all the admitted ranks, $0 \leq r < r_{drop}$, to some queue, removing a rank from a queue implies adding it to the adjacent queue. Thus, any reduction of unpifoness in a queue, increases the unpifoness of the adjacent queue. We can only perform such an optimization step as long as there is a queue that can absorb the cost of taking in more ranks without becoming a new, greater maximum-cost queue. The optimum is achieved when the estimated scheduling unpifoness in each queue is balanced out.

PACKS's collateral drops Unlike in PIFO, the admission-control in the PACKS abstraction (i.e., drop if $r \geq r_{drop}$) is not its only source of packet drops. Indeed, an admitted packet can still be dropped by the priority-queue's enqueue mechanism if the available buffer space in the selected queue is not sufficient to accommodate the packet. As such, in order for the PACKS abstraction to fully approximate PIFO's admission behavior, it should not only control which packets are admitted; it should also make sure that the admitted packets are not dropped at enqueue when they

are mapped to the priority queues. This brings us to the third part of the PIFO-approximation problem: approximating PIFO's efficient buffer usage.

We now compute the optimal queue bounds that minimize the drops that occur when mapping packets to priority queues, $\boldsymbol{q}_{\mathcal{D}}^{*}$, and compare them to the optimal bounds that optimize rank order at the scheduler's output, $\boldsymbol{q}_{\mathcal{S}}^{*}$.

Let B_i define the buffer capacity of the i-th priority queue in the PACKS abstraction. Let $\boldsymbol{B} = (B_1, \ldots, B_n) \in \mathbb{Z}^n$ describe the buffer allocation across queues, where the sum of the buffer space of each queue is the total buffer space: $\sum_{i=1}^{n} B_i = B$. Let $\boldsymbol{q} = (q_1, ..., q_n) \in \mathbb{Z}^n$ be the set of queue bounds defining the mapping strategy, where $0 \leq q_1 \leq q_2 \leq, \ldots, \leq q_n = r_{drop} - 1$. With this strategy, we can compute the number of packets mapped to the i-th priority queue, m_i, as:

$$\begin{aligned} m_1 &= [A \cdot \mathcal{W}.quantile(q_1)] \\ m_2 &= [A \cdot \mathcal{W}.quantile(q_2)] - m_1 \\ m_n &= [A \cdot \mathcal{W}.quantile(q_n)] - m_{n-1} \end{aligned} \tag{3.6}$$

We define a loss function $\mathcal{U}_{\mathcal{D}} : \mathrm{R}^n \times \mathrm{R}_{\geq 0} \rightarrow \mathrm{R}_{\geq 0}$, which stands for *dropping unpifoness*, such that $\mathcal{U}_{\mathcal{D}}(\boldsymbol{q})$ measures the number of packets dropped when mapping packets to queues based on queue bounds $\boldsymbol{q}$. In the PIFO queue, $\mathcal{U}_{\mathcal{D}} = 0$, since there is no queue mapping, and drops only occur at admission. In PACKS, a lower $\mathcal{U}_{\mathcal{D}}(\boldsymbol{q})$ leads to a better approximation.

Our goal is to find the optimal bounds, $\boldsymbol{q}_{\mathcal{D}}^{*}$, that minimize $\mathcal{U}_{\mathcal{D}}(\boldsymbol{q})$. Let $\mathcal{Q}$ be the space of valid queue-bound vectors and $\mathcal{W}$ the distribution of packet ranks, then the bounds $\boldsymbol{q}_{\mathcal{D}}^{*}$ are:

$$\boldsymbol{q}_{\mathcal{D}}^{*} = \underset{\boldsymbol{q} \in \mathcal{Q}}{\arg\min}\, \mathcal{U}_{\mathcal{D}}(\boldsymbol{q}) \tag{3.7}$$

Since bounds are fixed during the enqueue process, and each queue drops packets independently of the others, we can compute the total unpifoness as the sum of the individual losses at each queue, $\mathcal{U}_{\mathcal{D}}(q_i)$:

$$\mathcal{U}_{\mathcal{D}}(\boldsymbol{q}) = \sum_{1 \leq i \leq n} \mathcal{U}_{\mathcal{D}}(q_i) \tag{3.8}$$

The loss at queue i, $\mathcal{U}_{\mathcal{D}}(q_i)$, is either the difference between the number of packets mapped to the queue, m_i, and the queue space, B_i, or 0, if the packets have not filled up the queue space:

$$\mathcal{U}_{\mathcal{D}}(q_i) = \begin{cases} m_i - B_i & \text{if } m_i > B_i \\ 0 & \text{otherwise.} \end{cases} \tag{3.9}$$

As such, the optimal bounds $\boldsymbol{q}^*_{\mathcal{D}}$ are the ones that minimize the difference between the number of packets mapped to each queue and the buffer size of the queue. Since all packet drops contribute equally to the loss function, there may exist multiple queue-bound vectors, $\boldsymbol{q}^*_{\mathcal{D}}$, that result in an optimal number of drops. In fact, any set of queue bounds is optimal as long as the packets mapped to each queue is lower or equal than the buffer space allocated to that queue (i.e., $m_i \leq B_i$):

$$\forall i : A \cdot (\mathcal{W}.quantile(q_i) - \mathcal{W}.quantile(q_{i-1})) \leq B_i \tag{3.10}$$

Given that PACKS's admission control already ensures that the total number of packets admitted can fit within the total buffer space (i.e., $A \cdot \mathcal{W}.quantile(r_{drop} - 1) \leq B$), we can guarantee that there exists at least one set of queue bounds, $\boldsymbol{q}^*_{\mathcal{D}}$, that leads to zero drops at queue-mapping time. We can find such optimal bounds by computing the ranks for which the quantile of the rank distribution stays below the allocated queue sizes. This is $\forall i$: maximize q_i s.t. the eq. 3.10 is satisfied.

Same as it happened in the admission-control counterpart, there may be rank distributions for which the number of packets of a certain rank exceeds the queue capacity (even when that rank is the only one mapped to the queue). In that case, we need finer granularity than the rank-level to perform the queue mapping. Same as we did for admission control, we can overcome this limitation by introducing an enqueue-time value t_i, for each priority queue, i, such that packets are only admitted to the queue if: $r \leq q_i - 1$ or if $\{r = q_i \text{ and } t \leq t_i\}$. Packets not admitted by queue i are carried over to the next queue, $i + 1$, which has to add them to its quantile.

Sorting vs. dropping Having computed the optimal queue bounds that best approximate PIFO in optimizing rank order at the output, $\boldsymbol{q}^*_{\mathcal{S}}$, and in minimizing packet drops at queue-mapping, $\boldsymbol{q}^*_{\mathcal{D}}$, we can see that they are not always the same. Indeed, $\boldsymbol{q}^*_{\mathcal{S}}$ minimizes the quantiles of the rank distribution for the ranks mapped to each priority queue, and $\boldsymbol{q}^*_{\mathcal{D}}$ minimizes

the difference between these quantiles and their respective queue sizes. Thus, which queue bounds should we use?

In general, we could pick any of the two options based on e.g., which of the two behaviors we believe is more important. However, since our goal is to design a *programmable* scheduler, we select the option that *generalizes* the most. We realize that $\boldsymbol{q}_{\mathcal{D}}^{*}$ are not only the best bounds for minimizing packet drops at queue-mapping time, but *also* the optimal bounds for *scheduling* in case the rank distribution is not known a priori (see eq.3.5 and eq.3.10). Indeed, if the rank distribution of incoming packets is not known, the optimal queue mapping that minimizes rank reordering is the one that distributes packets across queues proportionally to the queue sizes. Thus, $\boldsymbol{q}_{\mathcal{D}}^{*}$ can be seen as a worst-case bound for $\boldsymbol{q}_{\mathcal{S}}^{*}$, leading to a good performance in both dimensions, as we show in §3.8. As a result, we leverage $\boldsymbol{q}_{\mathcal{D}}^{*}$, as the queue bounds for our design.

3.4.3 *Online adaptation*

So far, we have assumed a simplified scenario where packets arrive to the scheduler in a *batch*-basis, all packets have the same size, and we know the complete rank distribution of the batch at enqueue. In practice, however, packets arrive in a *stream*, and the scheduler needs to perform the admission and enqueue decisions *per-packet, at line rate*. In the following, we translate our high-level intuition to an algorithm design that is practical and which we can deploy to existing hardware.

Sliding window to monitor rank distribution In the online setup we do not know the rank distribution of incoming packets, $\mathcal{W}$, in advance. Instead, the best prediction that we can make is based on the rank distribution of recently-received packets. As such, same as previous approaches [41, 47], we monitor this distribution, W, using a sliding window, and use it to drive the admission and queue-mapping decisions.

Queue occupancy to estimate congestion In the online case, packets arrive in a *continuous* stream and not in a batch basis. Thus, instead of computing the quantiles over the number of packets arrived in the batch, A, we do so over the number of packets sharing the buffer in a scheduling interval, B. At the same time, while in the batch case we could assume empty queues at start, in the online case we have to consider dynamic buffers which should absorb the short-term mismatches between traffic arrival and departure

rates. We do so by measuring the buffer occupancy of the queues, and using them as an estimate of their congestion levels. [3] As a result, given b, the buffer-occupancy level at a certain packet's enqueue time, we decide to admit the packet if $W.quantile(r_{drop} - 1) \leq \left\lfloor \frac{1}{1-k} \cdot \frac{B-b}{B} \right\rfloor$, where k is an optional parameter to give room for burstiness. Similarly, given b_i, the buffer-occupancy level of queue i, we perform the queue-mapping process based on queue bounds, $\boldsymbol{q}$, satisfying:

$$
\begin{aligned}
q_1 &:= \max_{r_1 \in \mathbb{N}} \; s.t.\; W.quantile(r_1) \;\leq\; \frac{1}{1-k} \cdot \left\lfloor \frac{(B_1 - b_1)}{B} \right\rfloor \\
q_2 &:= \max_{r_2 \in \mathbb{N}} \; s.t.\; W.quantile(r_2) \leq \\
&\qquad\qquad \leq \; \frac{1}{1-k} \cdot \left\lfloor \frac{(B_1 - b_1)}{B} + \frac{(B_2 - b_2)}{B} \right\rfloor \\
&\quad \ldots \\
q_n &:= \max_{r_n \in \mathbb{N}} \; s.t.\; W.quantile(r_n) \leq \frac{1}{1-k} \left\lfloor \frac{\sum_{j=1}^{n} (B_j - b_j)}{B} \right\rfloor
\end{aligned}
\tag{3.11}
$$

Since $q_n = r_{drop} - 1$, the lowest-priority queue's mapping policy already implies the admission control at the scheduler, which simplifies the algorithm implementation (cf. alg. 3).

Minimizing collateral drops Same as in the batch case, packets of a certain rank may exceed a queue's capacity. In the batch case, we relied on t_i to map packets to a lower-priority queue if the higher-priority queues were full. In the online case, we assess queue occupancy during mapping; if the selected queue for a given packet is full, we direct the packet to the next queue with available space. This approach addresses a key limitation of queue-mappers like SP-PIFO, which excessively drop incoming packets when mapped to the same queue (e.g., during bursts of packets with the same rank or with monotonic rank increase). As such, PACKS prevents drops and ensures an efficient usage of the buffer resources. Additionally, PACKS's *top-down* scanning process ensures that PACKS preserves the scheduling order of such packet sequences, despite mapping them to different priority queues.

Algorithm 3 PACKS

Require: An incoming packet pkt with rank r

1: **procedure** INGRESS
2: Update sliding window W with r
3: $B \leftarrow$ buffer.total $\quad B_i \leftarrow$ buffer(q_i).total
4: $b_i \leftarrow$ buffer(q_i).used
5: **for** $Queues(i) : i = 1$ **to** n **do** ▷ Scan top-down
6: **if** $W.quantile(r) \leq \frac{1}{1-k} \cdot \left\lfloor \frac{\sum_{j=1}^{i}(B_j - b_j)}{B} \right\rfloor$ **then**
7: **if** $b_i < B_i$ **then** ▷ Queue i not full
8: $Queues(i).enqueue(pkt)$ ▷ Select queue
9: **return;**
10: **end if**
11: **end if**
12: **end for**
13: $Drop(pkt)$ ▷ Drop packet
14: **end procedure**

3.4.4 *PACKS algorithm*

We detail the PACKS algorithm in alg 3. For each incoming packet, PACKS decides whether to admit the packet or drop it, and how to map admitted packets to priority queues.

Admission control Whenever an incoming packet arrives, PACKS performs two main operations. First, it updates the sliding window, W, with the rank of the new packet, r. Then, it measures the current buffer occupancy, b and uses it to compute the portion of the buffer space, B, that is still free: $\frac{B-b}{B}$. PACKS admits the incoming packet if the quantile of its rank in the monitored rank distribution is lower than the fraction of available buffer space: $W.quantile(r) \leq \left\lfloor \frac{1}{1-k} \cdot \frac{B-b}{B} \right\rfloor$. Note that we weight the admission condition by an optional parameter, k, to allow for some burstiness. Also, note that in alg. 3, the admission condition is *implicit* in the queue-mapping process. Indeed, the drop action in line 13, executed when the packet has

3 This is a common approach in queue-management [13, 41, 48]. We could also have used the *sojourn-time* of packets, as proposed by CoDel [48].

not been mapped to the lowest-priority queue, already serves the purpose of admission control.

Queue mapping For the admitted packets, PACKS scans priority queues top-down (i.e., from highest- to lowest-priority) and maps the packet to the first queue with available space that satisfies the condition: $W.quantile(r) \leq \frac{1}{1-k} \cdot \left\lfloor \frac{\sum_{j=1}^{n}(B_j - b_j)}{B} \right\rfloor$. If a packet is not admitted to any of the queues, because its rank is too high, or because all queues are full, it is dropped.

3.4.5 *PACKS analysis*

Similarly to other networking algorithms [13, 41, 47–50], PACKS uses a window-based approach instead of a per-packet heuristic. As a result, PACKS outperforms under a stable rank distribution, when the window size is large. While the window-based approach generally makes PACKS less vulnerable to adversarial packet workloads (PACKS's bounds are updated more smoothly, making them harder to disrupt, cf. Fig. 3.9), it also represents PACKS's Achilles' heel.

We evaluated PACKS on MetaOpt [45]–a recent heuristic analysis tool–, to understand its performance gap relative to SP-PIFO, AIFO and PIFO, and to identify adversarial inputs (cf. §3.6). We found that PACKS is robust against adversarial sequences that make SP-PIFO drop more than 60% of high-priority packets, or make AIFO delay highest-priority packets by more than 60% of the total queue size. We also found that PACKS's adversarial inputs consist of bursts of either very high or very low rank packets, which "pollute" the monitored distribution and prevent the well functioning of the algorithm (cf. §3.6.3). In §3.8.1 we study the impact of such behaviors in depth and show how PACKS can react faster to such distribution changes by using smaller window sizes and higher burstiness allowances. Then, PACKS relaxes its admission criteria and its behavior converges to the one of per-packet heuristics such as SP-PIFO (cf. Fig 3.19).

In §3.5, we study PACKS's optimality theoretically. We prove that, for certain window and buffer-size conditions, the departure rate for all ranks in PACKS converges to that of a PIFO queue (cf. Theorem. 3.1). We also suggest an upper bound for the number of inversions that PACKS produces for a generic packet sequence, with respect to PIFO (cf. Claim. 1).

3.5 THEORETICAL ANALYSIS OF PACKS

Comparison with PIFO In the following, first, we show that, under certain conditions, the departure rate for *all* packet ranks in PACKS is the same as for a PIFO queue. Moreover, under these conditions, there is only a small difference between the sets of packets forwarded by PIFO and PACKS.

Let the set of packets forwarded (up to time t) by PIFO and PACKS be $\text{PIFO}(t)$ and $\text{PACKS}(t)$, respectively. Then, to measure the difference in drops between PACKS and PIFO, we define:

$$\Delta(t) = \frac{|\text{PIFO}(t) \setminus \text{PACKS}(t)| + |\text{PACKS}(t) \setminus \text{PIFO}(t)|}{|\text{PIFO}(t) + \text{PACKS}(t)|}.$$

We have $\Delta(t) \in [0, 1]$, where a small value of $\Delta(t)$ indicates a small difference between PACKS and PIFO. We denote the maximal and minimal rank probabilities with $\delta_+ := \max_i p(i)$ and $\delta_- := \min_i p(i)$.

Theorem 3.1. *Assume that the window size $|\mathcal{W}|$, buffer spaces $B_1, \ldots, B_n$, and the number of arrived packets, T, tend to infinity. Furthermore, assume that the maximal and minimal rank probabilities δ_+ and δ_- are bounded between two positive constants. We denote the ratio of the outgoing and incoming packet rate by v, and suppose $v < 1$ (otherwise, both PIFO and PACKS behave like a FIFO). We claim that the difference between the drops of PIFO and PACKS is at most δ_+, i.e., $\Delta(T)_{T\to\infty} \leq \delta_+$. Moreover, for each packet rank, the admission rate of PACKS is identical to the one of PIFO.*

Proof: Since the window size, $|\mathcal{W}|$, is considered very large, the empirical rank distribution in $\mathcal{W}$ tends to the real packet rank distribution. In other words, after waiting a long time, we can know the rank probabilities with high precision, that is $|p(i) - p_{\mathcal{W}}(i)| \xrightarrow[T \geq |\mathcal{W}|]{|\mathcal{W}|\to\infty} 0$. Thus, empirical quantiles, $\mathcal{W}.quantile(i)$, tend to the quantiles according to the real distribution, i.e., $\mathcal{W}.quantile(i) \to \sum_{i=1}^{r} p_i$.

Intuitively, since the buffer space B is very large, the relative queue occupancy b/B changes smoothly over time. More precisely, let $b(t)$ denote the buffer occupancy after the arrival of the t^{th} packet (or, for short, 'at time t'), and let $q_n(t) = \frac{1}{1-k}\frac{B-b(t)}{B}$ denote the highest queue bound at time t. At time T, we have queue bound $q_n(T)$ as the admission bound. Let r_T be the maximum rank such that $\mathcal{W}.quantile(r_T) \leq q_n(T)$. This means that the

ratio of the admitted packets is $\sum_{i=0}^{r_T} p(i)$ Thus, after the arrival of the next packet, $\mathbb{E}(b(T+1) - b(T)) = \sum_{i=0}^{r_T} p(i) - v$ (recall that, for every incoming packet, the number of drained packets is v on average). This means the following.

1. If $\sum_{i=0}^{r_T} p(i) > v$, the queue occupancy likely increases, ultimately triggering a drop in q_n and in the rate of admitted packets.
2. If $\sum_{i=0}^{r_T} p(i) < v$, the occupancy likely decreases, triggering a rise in q_n and in the rate of admitted packets.
3. Finally, in the event of $\sum_{i=0}^{r_T} p(i) = v$, the queue occupancy makes a motion very similar to the one-dimensional random walk, eventually, after a while likely triggering q_n to either drop or rise for a short time period, before bouncing back to r_T.

We note that, since the buffer spaces are considered to be very large, and the minimum rank probability δ_- is lower bounded by a positive constant, these events happen with probability 1 based on the law of large numbers. Furthermore, in case 3, $q_n(T+t) = r_T$ for any $t \geq 0$ with probability 1.

This also means that, in case 3, $\Delta(T) \xrightarrow{T\to\infty} 0$, since after a while PIFO and PACKS forward the same packets with probability 1. In cases 1 and 2, there is a single rank 'on the border' that either gets forwarded or dropped by chance both in PIFO and PACKS; thus, in these cases, $\Delta(T)_{T\to\infty} \leq \delta_+$. Note that the overall forwarding rate of this rank (and thus of all ranks) is the same for both PIFO and PACKS.

An alternative *intuitive* reasoning supporting the statement that the forwarding rates coincide for PIFO and PACKS is the following. In both cases, there are three classes of ranks: (i) small ranks that are always forwarded; (ii) large ranks that are always rejected; and (iii) a borderline rank r^* that is either forwarded or rejected by chance. Since draining is continuous both for PIFO and PACKS, the leftover bandwidth after the small-ranked packets is given to the borderline rank, r^*, as it is the only choice. □

Next, we present an asymptotically tight upper bound for the number of inversions that PACKS produces for a packet sequence compared to PIFO.

Claim 1. *On a sequence of S packets, given a buffer size of B, PACKS cannot cause more than $\Theta(B \cdot S)$ inversions with respect to PIFO.*

Proof: A bad sequence for PACKS: Take a sufficiently large S, with the packet ranks in the sequence being $S, S-1, \ldots, 1$. Then, PACKS will enqueue all the

packets to the highest priority queue $Queues(1)$. In this setting, the behavior of PACKS basically transforms to being a FIFO using $Queues(1)$. We assume that after a brief period, when the rate of packet arrivals exceeds the departure rate, $Queues(1)$ gets full. Oversimplified, e.g., while enqueuing the first B_1 packets, none is dequeued. Then, packets are enqueued and dequeued in an alternate fashion. In this simplified example, the PIFO output rank sequence will be (first dequeued on the left): $O_P = [S - B_1, S - B_1 - 1, \ldots, 1, S - B_1 + 1, S - B_1 + 2, \ldots, S]$. In the meantime, the output of PACKS is the same sequence as the input was: $[S, \ldots, 1]$. We can observe that PIFO forwarded a number of $S - B_1$ packets B_1 time slots faster than PACKS. If $S \gg B_1$ and $B_1 \geq c \cdot B$ for some constant $c > 0$, then the number of inversions produced by PACKS compared to the PIFO output is $\Omega(SB)$. We can see that the same asymptotic bound hold in the more realistic scenario when some packets are dequeued in the initial phase when the queue gets full.

Upper bound if the same packets are admitted as PIFO: obviously, one packet cannot get ahead more packets than the buffer size B, hence in the output sequence of PACKS there could be not more than $O(BS)$ more inversions than in the output of PIFO. □

We note that, after a short initialization period, an ever-decreasing sequence of ranks (like in the proof of Claim 1) makes AIFO and SP-PIFO suffer similarly as PACKS in terms of the number of rank inversions.

Comparison with AIFO The next theorem states that PACKS admits the same packets as AIFO. This notable since AIFO was designed to mimic the admission behavior of PIFO.

Theorem 3.2. *Given the same window size, buffer size, and burstiness allowance, PACKS drops the same packets as AIFO.*

Proof: Following the notations of the AIFO paper [41], we denote the total buffer size of AIFO by C, and its queue occupancy by c. AIFO admits a packet r if $W.quantile(r) \leq \frac{1}{1-k} \cdot \frac{C-c}{C}$. Assume indirectly that there exists an $t \in \mathbb{N}$, for which the t^{th} arriving packet is enqueued in exactly one of PACKS and AIFO. We choose t as the minimum of such values. We denote the rank of this packet as r_t.

Case 1: PACKS enqueued r_t, while AIFO did not. Here we have the following inequalities explained below, yielding a contradiction:

$$W.quantile(r_t) \overset{(a)}{>} \frac{1}{1-k} \cdot \frac{C-c}{C} \overset{(b)}{=} \frac{1}{1-k} \cdot \frac{\sum_{j=1}^{n} B_j - b_j}{B} \overset{(c)}{\geq}$$

$$\overset{(c)}{\geq} W.quantile(r_t).$$

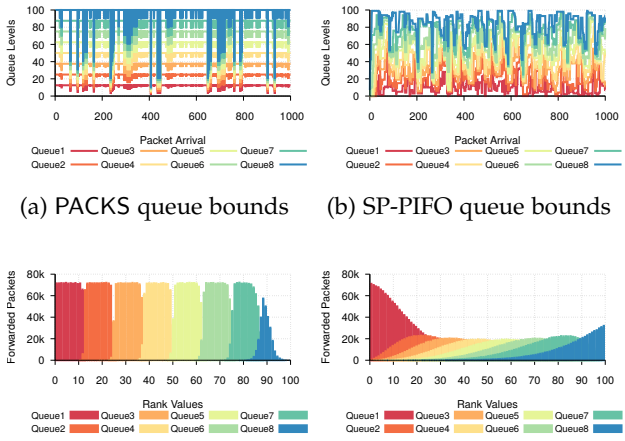
Here, we get (a) from the fact that AIFO does not enqueue r_x. For (b), we just match the notations of AIFO and PACKS. Finally, (c) holds because PACKS enqueues r_t. Combined, (a), (b), and (c) clearly yield a contradiction.

Case 2: AIFO enqueued r_t, while PACKS did not. Since AIFO enqueued r_t, we have $W.quantile(r_t) \leq \frac{1}{1-k} \cdot \frac{C-c}{C} = \frac{1}{1-k} \cdot \frac{\sum_{j=1}^{n} B_j - b_j}{B}$. Let $i \in \{1, \ldots, n\}$ be the minimal number such that $W.quantile(r_t) \leq \frac{1}{1-k} \cdot \frac{\sum_{j=i}^{n} B_j - b_j}{B}$. We know that such an i exists. Since PACKS did not enqueue r_t at all, we can deduce it did not enqueue r_t in the i^{th} queue either. This is possible only if $b_i = B_i$. If $i \geq 2$, this contradicts the minimality of i, since this means $W.quantile(r_t) \leq \frac{1}{1-k} \cdot \frac{\sum_{j=i-1}^{n} B_j - b_j}{B}$. If $i = 1$, and $n \geq 2$, then PACKS will enqueue r_t to the first queue having free space; note that such a queue exists, since before the arrival of r_t, AIFO had spare buffer space. Finally, The case of $i = 1$, and $n = 1$ also yields contradiction, since then, the AIFO would not have enqueued r_t either. The proof follows. □

Finally, we argue that, for the highest priority packets, PACKS causes no more rank inversions than AIFO.

Theorem 3.3. *For any packet sequence, given the same window size, total buffer size, and burstiness allowance, PACKS causes no more priority inversions than AIFO for the highest priority packets.*

Proof: The theorem follows from two statements: (a) AIFO and PACKS admit the same set of packets (under the same configuration, see Theorem. 3.2), and (b) The quantile estimate of the highest priority packet is always the smallest (equalling 0). Let t denote the index of the packet in the input sequence. Let I_{PACKS} and I_{AIFO} denote the number of higher-ranked packets that t follows in the output sequence of PACKS and AIFO, respectively. From (b), we can show that for a given packet with the highest priority and index t, there is no packet that arrives after t (having an index greater than t), and is going to be dequeued before packet t. Rephrased, this means that



(a) PACKS queue bounds (b) SP-PIFO queue bounds

(c) PACKS queue mapping (d) SP-PIFO queue mapping

FIGURE 3.9: Queue-bounds evolution and rank mapping for PACKS and SP-PIFO under a uniform distribution (8 queues).

$I_{\text{PACKS}} \leq I_{AIFO}$. This is true for each packet of highest priority. Thus, PACKS always has at most the same total number of priority inversions as AIFO for the highest priority packets. □

3.6 PERFORMANCE ANALYSIS USING METAOPT

MetaOpt [45, 51] is a tool to compare the performance of two competing heuristics or a heuristic and an optimal solution. It identifies adversarial workloads that cause the maximum difference between the performance of the two algorithms specified as input.

We model PACKS in MetaOpt and compare it to AIFO [41], SP-PIFO [1], and PIFO [20]. Our goal is to understand *when and under what inputs* PACKS performs substantially better or worse than them. We focus on two performance metrics. The first metric is the number of packets dropped weighted by the packet's priority (where the priority is defined as the difference between the maximum rank in the distribution and the packet rank: max. rank - packet rank). The second metric is the number of priority inversions weighted by the packet's priority. These metrics help us identify the

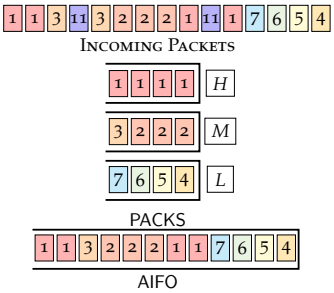


FIGURE 3.10: Adversarial input that maximizes the gap between weighted priority inversion of AIFO compared to PACKS. AIFO can delay the highest priority packet by more than 60% of the queue length. (Starting window = [1, 1, 1, 1]).

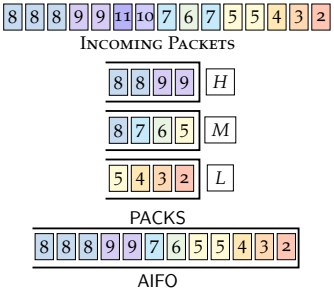


FIGURE 3.11: Adversarial input that maximizes the gap between weighted priority inversion of PACKS compared AIFO. (Starting window = [1, 1, 1, 1]).

adversarial inputs that cause the heuristics to disrupt the performance of lower-rank packets (which are most important in the PIFO context).

Experiment setup We let packets take ranks between 1 and 11. We consider *all* the 15-packet traces possible with these ranks. We set the buffer size to 12 packets, and assume it empty at start. We configure PACKS and AIFO with a window size $|\mathcal{W}| = 4$ and a burstiness allowance $k = 0$. We configure SP-PIFO and PACKS with 3 priority queues of 4 packets each.

3.6.1 *Comparison with AIFO*

Packet drops We find that PACKS and AIFO always admit the same set of packets when they use the same configuration. This is expected, as we prove in Theorem. 3.2.

Rank inversions Fig. 3.10 and Fig. 3.11 illustrate the adversarial inputs that MetaOpt discovered for AIFO with respect to PACKS and vice versa. We find that AIFO can delay the highest priority packets by more than 60% of the total queue size compared to PACKS. AIFO only has an admission policy and suffers when the input sequence is not sorted. In Fig. 3.10, we show an input sequence where AIFO causes 24 priority inversions for lowest-rank packets, whereas PACKS is able to fully sort the packets.

Adversarial inputs to AIFO consist of lower ranked packets compared to the adversarial inputs to PACKS. PACKS underperforms when a distribution shift happens, and packets in the window are not a good estimate of the newer incoming packets. The worst case of PACKS compared to AIFO is on a packet sequence that is approximately sorted (Fig. 3.11). Due to the distribution shift, PACKS ends up mapping higher-priority packets to lower-priority queues and lower-priority packets to higher-priority queues. In practice, the impact of scheduling such packet sequence with PACKS would not be significantly detrimental. Since queues are empty at start, PACKS would start sending the lower-rank packets while higher-rank ones arrive.

Further, PACKS's adversarial sequence (Fig. 3.11) consists of packets with higher ranks than the adversarial input to AIFO (Fig. 3.10), which have lower importance. This indicates that AIFO can cause higher average delays for important sensitive packets compared to PACKS. Our results show that PACKS never causes more priority inversion for the lowest ranked packets than AIFO (as we prove in Theorem. 3.3).

3.6.2 *Comparison with SP-PIFO*

Packet drops Fig. 3.12 and Fig. 3.13 show the adversarial inputs that MetaOpt found for SP-PIFO respect to PACKS and vice versa. We observe that SP-PIFO can drop more than 60% of high-priority packets while leaving 66% of the total queue size empty. SP-PIFO lacks an admission policy and underperforms when we have a stream of packets with the highest priority

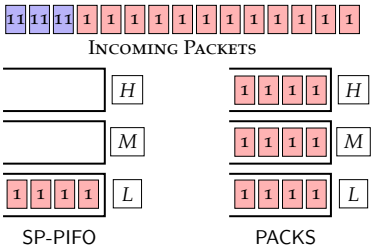


FIGURE 3.12: Adversarial input that maximizes the gap between weighted packet drop of SP-PIFO compared to PACKS. (Starting window = [1, 1, 1, 1]).

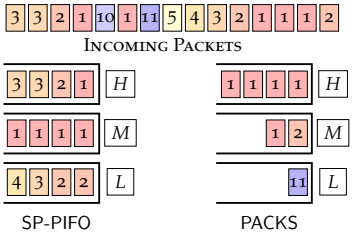


FIGURE 3.13: Adversarial input that maximizes the gap between weighted packet drop of PACKS compared to SP-PIFO. (Starting window = [1, 2, 1, 1]).

(all with rank 1). In this case, SP-PIFO maps all the packets to its lowest-priority queue, dropping many of them while the other queues are empty. PACKS, however, fills the queues one by one from highest to lowest priority, efficiently using the buffer resources and preventing packet drops.

PACKS drops at most 3 high-priority packets whereas SP-PIFO can drop up to 8 high-priority packets (2.33× more). PACKS underperforms when the input sequence meets two conditions: (i) the rank of most of the packets increases, and (ii) a few of the packets in the middle of the trace have a higher rank than the ones received before or after them. Condition (i) describes an adversarial case for both SP-PIFO and PACKS, but condition (ii) helps SP-PIFO mitigate this by moving to a higher priority queue. Even with this, PACKS drops at most 3 high-priority packets more than SP-PIFO (2.33× less than the packet drop of SP-PIFO on its adversarial input).

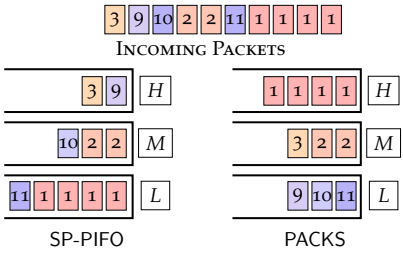


Figure 3.14: Adversarial input that maximizes the gap between weighted priority inversion of SP-PIFO compared to PACKS. (Starting window = [1, 1, 1, 1]).

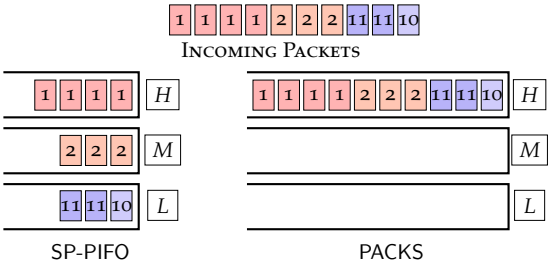


Figure 3.15: Adversarial input that maximizes the gap between weighted priority inversion of PACKS compared to SP-PIFO. (Starting window = [1, 1, 11, 11]).

Rank inversions To capture only the impact of rank inversions, we set the queue sizes long enough so that packet drops do not occur. Fig. 3.14 and Fig. 3.15 show the adversarial inputs that MetaOpt discovered. We note that the adversarial input to PACKS is only slightly worse than the adversarial input to SP-PIFO. The worst-case input for SP-PIFO with respect to PACKS causes 20 priority inversions for the highest priority packet, while the worst-case input for PACKS only causes 24 of them.

SP-PIFO performs poorly when the rank of most of the packets is sorted, but there are a few packets in between with higher ranks than the ones received before or after them. These higher ranks cause SP-PIFO to push the rest of packets to higher-priority queues, leading to priority inversions. This pattern is the same as the one that causes PACKS to drop numerous packets compared to SP-PIFO.

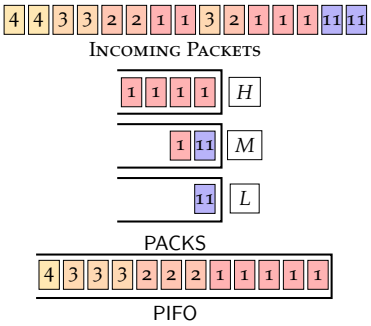


FIGURE 3.16: Adversarial input that maximizes the gap between weighted packet drop of PACKS compared to PIFO. (Starting window = [1, 1, 1, 1]).

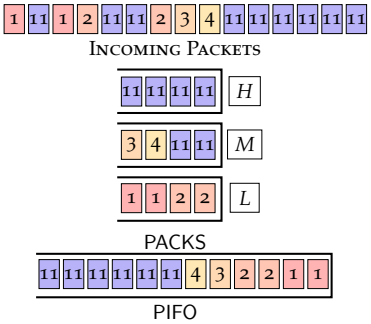


FIGURE 3.17: Adversarial input that maximizes the gap between weighted priority inversion of PACKS compared to PIFO.
(Starting window = [1, 11, 1, 11]).

PACKS underperforms when we can split the packets into multiple batches that meet two conditions: (i) the packets in each batch are in the non-decreasing order of their ranks, and (ii) all the packets in a given batch have higher rank than the packets in a subsequent batch. SP-PIFO would put each batch in one of its queues, achieving perfect sorting, whereas PACKS does not perform any sorting across batches of packets.

3.6.3 *Comparison with PIFO*

Fig. 3.17 and Fig. 3.16 show the adversarial inputs that MetaOpt discovered. We see that the worst-case input to PACKS (with respect to PIFO) is the same as the worst-case input to AIFO (with respect to PIFO).

Packet drops The worst-case input is an increasing sequence of packet ranks. PACKS (similar to AIFO) computes the quantile using a sliding window. In this sequence, the quantile estimate of every packet is large, so PACKS (similar to AIFO) will drop the packets. The fact that both worst-case inputs to PACKS and AIFO, with respect to PIFO are the same, is expected, since PACKS and AIFO drop the same packets, as proved in Theorem. 3.2.

Rank inversions The worst-case input is a decreasing sequence of packet ranks. In that case, PACKS does not do any sorting and performs the same as AIFO (putting every packet in the highest priority queue with available space). This is also expected (cf. Claim 1 and the insight after its proof).

3.7 IMPLEMENTATION

We implemented PACKS in $P4_{16}$ for Intel Tofino 2 [21] using 439 lines of code. Our implementation uses 12 stages and the resources outlined in Table 3.2. For each incoming packet, PACKS: (i) monitors the distribution of recent ranks; (ii) computes the quantile of the packet's rank on this distribution; (iii) measures queue occupancies; and (iv) uses this data to decide the packet's admission, dropping, and queue mapping.

Rank-distribution monitoring We track the rank distribution of the packets received by implementing a sliding window over a set of $|W|$ registers. Each register stores the rank of one packet, and we use a circular packet counter, from 0 to $|W| - 1$, to track the position of the oldest update. Upon the arrival of a new packet, we check the counter's value and update the register pointed to by the counter with the value of the new packet's rank. In our prototype, the sliding window has a size of 16 (which can be extended by using sampling [41]). It uses 4 stages and accesses 4 registers in parallel at each stage.

Quantile computation We compute the quantile of each incoming packet's rank based on the monitored distribution. Specifically, we count how many

times the packet's rank is lower than a rank in the sliding window and then divide the result by the window size. We perform the count by accessing each register of the sliding window and comparing the packet's rank with the register's value using the register's stateful ALU. We output the result of each comparison into a binary metadata field, $output_j$, which is set to 1 if the packet's rank is smaller than the register value and 0 otherwise. We aggregate the output values by progressively summing pairs of them at each stage using non-stateful ALUs, requiring $\log_2 |W|$ stages. Finally, we divide the sum of the output values by the window size, $|W|$: $W.quantile(r) = (\sum_j output_j)/|W|$. We select the window size to be a power of 2, and implement the division through bit-shift operations.

Queue-occupancy monitoring We use a *ghost thread* [52], available in Tofino 2, to monitor queue occupancy levels at enqueue. Normally, this information is only available in the *egress* pipeline, since packets need to traverse the traffic manager to access it. We address this limitation by setting up a ghost thread that periodically writes the queues' occupancy levels (from the egress) to a register accessible from the ingress pipeline. The ghost thread takes two clock cycles to update the state of each queue and handles one queue per invocation. This results in 8 clock cycles to update the state of 4 queues. To scale PACKS across a larger set of queues and ports, we approximate the admission and queue-mapping conditions by considering the overall buffer occupancy instead of individual per-queue occupancies (i.e., $W.quantile(r) \leq \frac{1}{1-k} \cdot \frac{B-b}{B} \cdot \frac{i}{n}$). Alternatively, we could use traditional packet recirculation to convey queue-occupancy information to the ingress pipeline (as done in [41]). The first option sacrifices accuracy, while the second, throughput.

Admission and queue mapping After obtaining the quantile of the packet's rank based on the monitored rank distribution, $W.quantile(r)$, and the available buffer, b, we combine them to derive the admission and mapping conditions. We rewrite them as: $B \cdot (1-k) \cdot n \cdot W.quantile(r) \leq (B-b) \cdot i$. We compute the right side of the equation by using the math unit and bit-shift operations. Simultaneously, we compute the left side of the equation by picking a k value strategically such that the operation can be performed by a bit shift on the quantile. Finally, we compute the comparison between the two terms using the *minimum* operation of the math unit, and execute the corresponding drop or enqueue action based on its result.

Resource Type	Usage (Average per stage)
Exact Match Crossbar	3.4%
Gateway	3.4%
Hash Bit	1.3%
Hash Dist. Unit	4.2%
Logical Table ID	10.9%
SRAM	2.4%
TCAM	0%
Stateful ALU	23.8%

TABLE 3.2: Resource requirements of PACKS on Intel Tofino 2.

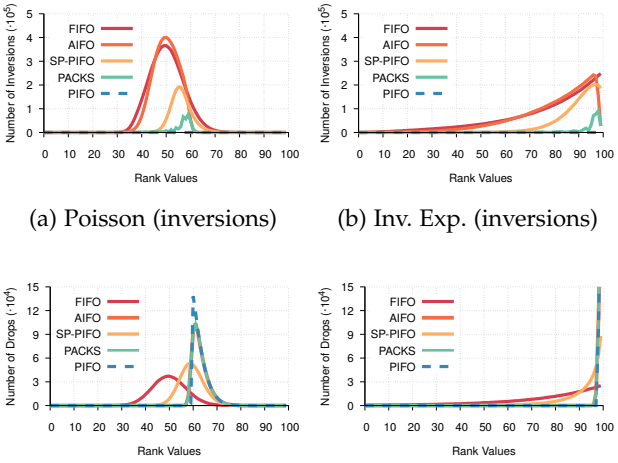
3.8 EVALUATION

We evaluate PACKS's performance in three steps. First, we study its performance in approximating PIFO's scheduling and admission behaviors for various rank distributions, and analyze its sensitivity to the configuration parameters (§3.8.1). Second, we study its practicality even under complex traffic workloads (§3.8.2). Finally, we evaluate PACKS's reaction and bandwidth allocation when deployed on hardware (§3.8.3).

3.8.1 *Performance analysis*

First, we analyze PACKS's behavior across different rank distributions to assess its performance in approximating PIFO's admission and scheduling.

Methodology We implement PACKS, PIFO, FIFO, SP-PIFO and AIFO in Netbench [26], a packet-level simulator. We study the performance of a switch scheduling a constant bit-rate flow of 11Gbps over a 10Gbps bottleneck link for one second. We assign each packet a rank within [0-100), drawn from an exponential, Poisson, convex, or inverse-exponential distribution. We set up PACKS and SP-PIFO with 8 priority queues of 10 packets, and AIFO and FIFO with a queue of 80 packets. We set PACKS's and AIFO's window size to 1000 packets and the burstiness allowance, k,



(a) Poisson (inversions) (b) Inv. Exp. (inversions)

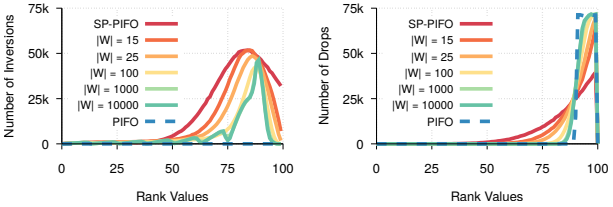
(c) Poisson (drops) (d) Inv. Exponential (drops)

FIGURE 3.18: PIFO approximations for various rank distributions.

to 0. We measure the number of scheduling inversions produced by each rank (i.e., how often a packet with the rank is scheduled *before* a lower-rank packet in the queue) and the number of dropped packets per rank.

Uniform case In §3.2.3, we have seen how PACKS outperforms existing schemes under a uniform rank distribution (cf. Fig.3.3) in both the number of scheduling inversions and the drop distribution across packet ranks. Indeed, PACKS reduces the number of inversions by more than $3\times$, $10\times$ and $12\times$ with respect to SP-PIFO, AIFO and FIFO. While all schemes drop a similar percentage of packets (within $\pm 0.03\%$), PACKS achieves the closest-to-PIFO drop distribution across packet ranks. PIFO only drops packets with ranks larger than 90. FIFO deviates furthest from PIFO by dropping packets across *all* ranks. It is followed by SP-PIFO, which drops packets with ranks as low as 20. AIFO and PACKS perform best, only dropping packets with ranks above 77 and 79, respectively.

Alternative distributions (inversions) We obtain similar results for non-uniform rank distributions. Fig. 3.18 shows the scheduling inversions and the packet drops across ranks for the Poisson and inverse-exponential rank distributions (we see similar results for the convex and exponential distributions). In all cases, PACKS outperforms SP-PIFO and AIFO, and gets



(a) Window size (inversions) (b) Window size (drops)

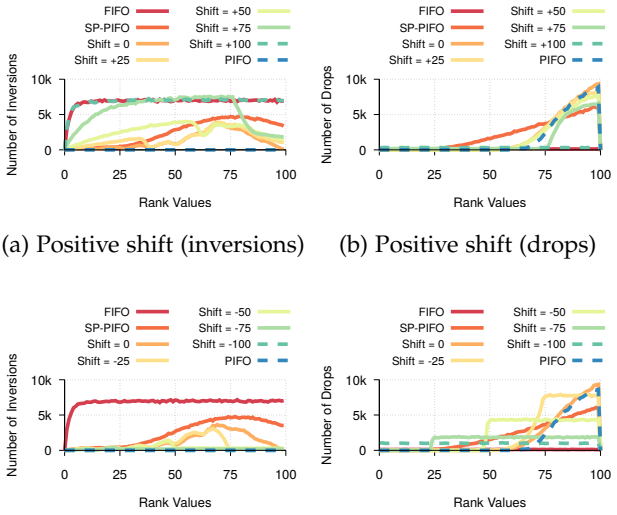
FIGURE 3.19: PACKS's window-size sensitivity (UDP, uniform).

closest to PIFO in inversions and packet drops. For the Poisson distribution, PACKS reduces the number of inversions by 5× and more than 15× and 17× compared to SP-PIFO, AIFO and FIFO, respectively. Similarly, for the inverse-exponential distribution, PACKS prevents over 7×, 14× and 15× more inversions than SP-PIFO, AIFO and FIFO, respectively. Notably, PACKS predominantly prevents inversions among lowest-ranked packets, which have higher priority.

Alternative distributions (drops) Under the Poisson distribution, all schemes drop overall a similar number of packets (within ±0.04%), being SP-PIFO the one with the highest drop rate. When considering the distribution of dropped packets across ranks, PACKS and AIFO are the schemes most closely approximating PIFO. Specifically, the lowest rank dropped by PIFO is 59, while PACKS and AIFO drop packets starting at rank 56 [4]. Conversely, SP-PIFO and FIFO show notably worse performance, dropping packets with ranks as low as 36 and 20, respectively. We observe similar results for the inverse-exponential distribution. In this case, however, while the total number of packets dropped by PACKS and AIFO is similar to the one of PIFO (+0.1% and +0.4%, respectively), SP-PIFO drops 42% more packets than them. This is due to the highly skewed nature of the distribution, which is hard for SP-PIFO to manage without admission control (cf. §3.4.4).

Sensitivity to window size Fig. 3.19 illustrates the impact of window size on PACKS's performance. Given that the rank distribution ranges from 0 to 100 and the overall buffer space is of 80 packets, PACKS performs best with window sizes above $|W| = 100$, which capture the entire distribution. Since the rank distribution is stable, a higher window size consistently leads to

4 Note that PACKS's and AIFO's drop distribution significantly overlap.



(a) Positive shift (inversions) (b) Positive shift (drops)

(c) Negative shift (inversions) (d) Negative shift (drops)

FIGURE 3.20: Rank-distribution sensitivity (TCP uniform).

more stable queue bounds and better performance, as indicated by the bumps in the distribution reflecting the behavior of priority queues. For example, with $|W| = 1000$, PACKS performs very close to optimal, reducing inversions by 22% compared to $|W| = 100$ and increasing the lowest-dropped rank from 69 to 78. Further increasing the window to $|W| = 10000$ doesn't improve performance significantly, only reducing inversions by 1% and raising the lowest-dropped rank from 78 to 80, compared to $|W| = 1000$.

Window sizes below $|W| = 100$ lead to worse performance since the window cannot capture the entire distribution. Interestingly, as we reduce the window size, PACKS's behavior approaches that of SP-PIFO. Nevertheless, even with very small window sizes, PACKS still outperforms. For instance, with $|W| = 15$, which barely captures a 15% of the distribution, PACKS still produces 30% less inversions compared to SP-PIFO and starts dropping packets at rank 34 instead of 18.

Sensitivity to distribution shifts We assess how PACKS performs when the monitored rank distribution differs from that of incoming packets. To do so, we modify PACKS's algorithm to consistently shift *all* ranks *in the sliding window* by a factor. This approach does not reflect a real-world scenario

since, even under a drastic distribution shift in practice (e.g., a microburst), packets from the "new distribution" would arrive in a continuous stream, allowing the sliding window to adapt gradually as each packet arrives. Still, it helps us understand PACKS's performance boundaries. We run TCP flows at 80% load, with packet ranked uniformly at random from 0 to 100.

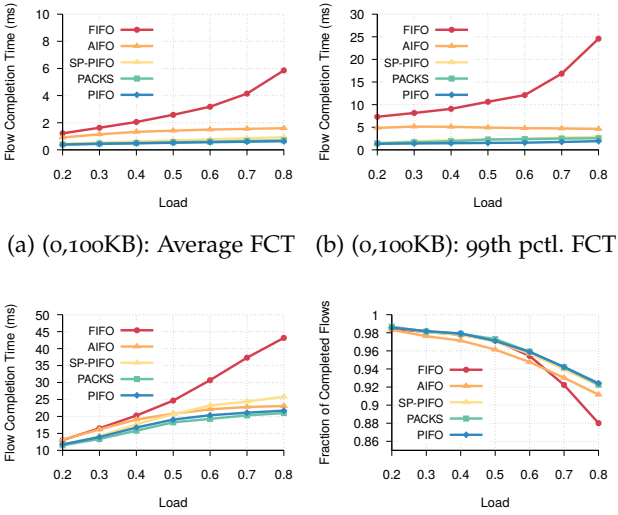
Fig. 3.20a and Fig. 3.20b show the impact of shifting the ranks of the sliding window by *positive* factors. This leads to more permissive admission and queue-mapping decisions, as if we increased the priority of incoming packets. When the shift reaches 100, all arriving packets have higher priority than the ones in the sliding window, causing PACKS to admit all packets and behave like a FIFO queue. Despite the extreme scenarios with shifts $\geq$75, PACKS exhibits significant robustness to positive distribution shifts. For instance, with a shift of +25, PACKS vastly outperforms SP-PIFO by reducing inversions by 34% and with a lowest-rank dropped of 46, as opposed to 12 in SP-PIFO. Even with a shift of +50, PACKS performs comparably to SP-PIFO in terms of total inversions while dropping 162$\times$ fewer packets below the rank of 58.

Fig. 3.20c and Fig. 3.20d show the impact of shifting the ranks of the sliding window by a *negative* factor. This is equivalent to decreasing the priority of incoming packets, which has a more detrimental impact on performance than positive shifts, and affects packet drops. Indeed, admission control drops a percentage of packets equal to the magnitude of the shift. With a -100 shift, PACKS drops all incoming packets. Similarly, a -75, -50 and -25 shift lead to dropping 75%, 50% and 25% of packets with the lowest priority, respectively. For the subset of admitted packets, PACKS maintains ideal behavior in terms of scheduling inversions. We can counteract the effect of negative distribution shifts by increasing the burstiness allowance, k, or decreasing the window size to speed up reaction time.

3.8.2 *Performance in typical use cases*

We now study PACKS's performance under two common scheduling objectives: minimizing flow completion times and enforcing fairness [1, 31, 41, 53]. These scenarios are challenging for PACKS because they involve large and non-stationary rank distributions, which are difficult to monitor.

Methodology We use a leaf-spine topology with 144 servers connected through 9 leaf and 4 spine switches, and set the access and leaf-spine links



(a) (0,100KB): Average FCT (b) (0,100KB): 99th pctl. FCT

(c) [All flows]: Average FCT (d) Completed flows

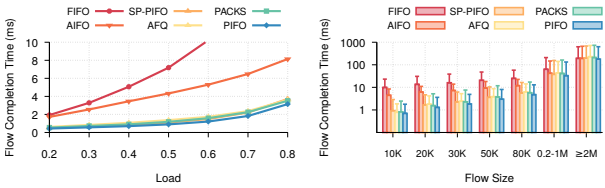
FIGURE 3.21: pFabric: FCT statistics across different flow sizes.

to 1Gbps and 4Gbps, respectively. We generate traffic flows following the pFabric web-search workload [31]. Flow arrivals are Poisson-distributed and we adapt their starting rates for different loads. We use ECMP and draw source-destination pairs uniformly at random.

Setup pFabric We run pFabric [31] (without starvation prevention [5]) on top of PIFO, AIFO, SP-PIFO and PACKS, and assess their efficacy in minimizing flow completion times. pFabric assigns ranks to packets based on their remaining flow sizes. As suggested in [31], we approximate pFabric's rate control using standard TCP with an RTO of 3 RTTs. We configure PACKS and SP-PIFO with 4 queues$\times$10 packets and PIFO, AIFO and FIFO with 1 queue$\times$40 packets. For PACKS and AIFO, we set $|W| = 20$ and $k = 0.1$.

Results pFabric Fig. 3.21 depicts the mean and 99th percentile FCT of small flows ($<$ 100KB), the average FCT across all flows, and the fraction of completed flows. Across all loads, PACKS consistently achieves lower FCTs compared to AIFO and SP-PIFO, and gets the closest to PIFO's performance.

5 Starvation [31] is a limitation inherent to PIFO, and therefore of all its approximations. Previous works have proposed solutions (e.g., PDA [43]) which can also run on PACKS.



(a) (0,100KB): Average FCT (b) FCT split, 70% load

FIGURE 3.22: Fairness: FCT statistics at different loads.

In terms of average FCTs for small flows, PACKS achieves FCTs just 5% to 9% longer than PIFO, which is remarkable given its use of just 4 queues. In turn, PACKS outperforms SP-PIFO by 11% to 33%, AIFO by a factor of 2.25× to 2.6×, and FIFO by 3.2× to 9.2× (most benefit under heavy loads).

At the 99th pctl., PACKS achieves FCTs 8% to 49% longer than PIFO, but remains better than SP-PIFO (from 2.2% to 12% better), AIFO (1.8× to 3.3×), and FIFO (5× to 10×).

Regarding the mean FCT across all flows, PACKS achieves on-par performance with PIFO (the mean FCT of PACKS is even a bit lower–from 2% to 5%–due to long flows not completing transmission, cf. Fig. 3.21d). Once again, PACKS consistently outperforms SP-PIFO across all loads (with improvements ranging from 0.2% at the lowest load to 23% at the highest load), AIFO (9% to 21%) and FIFO (13% to 2×).

Finally, PACKS's fraction of completed flows closely matches that of PIFO (± 0.01% to 0.2%). Moreover, PACKS achieves higher completion rates than SP-PIFO, with improvements from 0.06% to 0.2%, and AIFO (resp. 0.3% to 1.2%). At the highest load, the completion rates of FIFO, AIFO, SP-PIFO, PACKS and PIFO are 88%, 91.15%, 92.16%, 92.26% and 92.41%, respectively.

Setup fair queuing We run the Start-Time Fair Queueing rank design [32] on top of the schedulers and evaluate their performance at enforcing fairness across flows. We compare to FIFO and AFQ [33] for reference. We set the bytes-per-round of AFQ to 80 packets. We use 32 queues×10 packets in SP-schemes and 1 queue×320 packets for single-queue schemes. Same as [1, 41], we generate traffic from the pFabric web-search distribution, and assess

fairness by measuring the flow completion time of short flows. For PACKS and AIFO, we set the window size to 10 and the burstiness margin to 0.2.

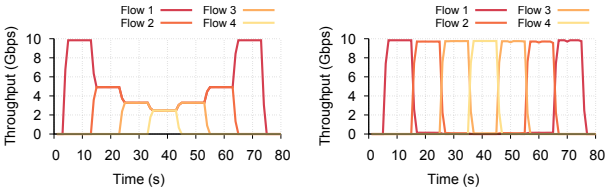
Results fair queuing Fig. 3.22a depicts the average flow completion time for small flows across loads from 20% to 80%. PACKS stays within 10–24% of the ideal PIFO, and consistently outperforms FIFO, AIFO and AFQ across all loads. Specifically, PACKS reduces the average FCTs for short flows by 2.5–5.5×, 1.12×–2.4×, 9–27% with respect to FIFO, AIFO and AFQ, respectively. PACKS performs similarly to SP-PIFO (within ±6%), underperforming at lower loads, but outperforming by 6% at the highest load (80% utilization).

Fig. 3.22b illustrates the average and 99th percentile flow completion times across flow sizes at 70% utilization. PACKS's performance consistently stays within 17–26% of the ideal PIFO in terms of average FCT and within 15–54% for the 99th percentile across all flow sizes. It shows comparable performance to SP-PIFO (within ±10% for average FCTs and ±20% for the 99th percentile) and AFQ for average FCTs (within ±15%). However, AFQ outperforms at the 99th percentile, by up to 31%. For the smallest flows, PACKS achieves the lowest average FCT, closely trailing AFQ by 5%.

3.8.3 *Hardware testbed*

We show that PACKS performs at line rate on actual hardware by running it on the Edgecore Networks DCS810 (AS9516-32D) Intel Tofino2 Switch [21]. Same as previous works [1, 41, 53], we measure the bandwidth that PACKS allocates to different priority flows over a bottleneck link. We generate traffic between two servers, connected by a Tofino2 switch, using interfaces of 100 Gbps (sender→switch) and 10 Gbps (switch→receiver). We run four UDP flows of 20 Gbps each using MoonGen [34, 54]. We start flows sequentially (one flow at a time), in increasing order of priority with a time gap of 10 seconds between starts. We stop them sequentially in decreasing order of priority, with 10 seconds between stops.

Fig. 3.23 depicts the flows' bandwidth and how PACKS manages to effectively prioritize traffic from lower ranks. While the FIFO queue distributes the bandwidth uniformly across flows (failing at prioritizing traffic), PACKS successfully allocates the available bandwidth to the highest-priority flow.



(a) FIFO Bandwidth-split (b) PACKS Bandwidth-split

FIGURE 3.23: Bandwidth allocation for increasing-priority flows.

3.9 RELATED WORK

Packet scheduling has been extensively studied for decades [17, 29, 31–33, 39, 55–58]. The concept of programmable scheduling was introduced by [19, 20], which proposed the PIFO queue as an enabling abstraction. While promising, implementing PIFO queues in hardware proved challenging. Hence, a subset of follow-up works have suggested new hardware designs such as PIEO [59], BMW-Tree [60], BBQ [61], and Sifter [62]. Other works have focused on approximating PIFO behaviors on existing programmable data planes: SP-PIFO [1], QCluster [43], PCQ [42], AIFO [41], Spring [47], and Gearbox [44]. PACKS falls into the latter category.

3.10 CONCLUSIONS

In this chapter, we presented PACKS, the first programmable packet scheduler that emulates PIFO queues on existing data planes in *both* rank ordering *and* packet drops. PACKS runs on top of a set of priority queues and uses packet-rank information and queue-occupancy levels during enqueue to schedule packets in order of priority, determining whether to admit each incoming packet and to which queue it should be mapped.

We fully implement PACKS in P4 and evaluate it on real workloads. We show that PACKS better-approximates PIFO than state-of-the-art approaches. Specifically, PACKS reduces the rank inversions by up to 7× and 15× with respect to SP-PIFO and AIFO, and the number of packet drops by up to 60% compared to SP-PIFO. Under pFabric ranks, PACKS reduces the mean FCT across small flows by up to 33% and 2.6×, compared to SP-PIFO and AIFO. We also show that PACKS runs at line rate on existing hardware.

4

QVISOR

4.1 INTRODUCTION

In the previous chapters we have proposed two programmable schedulers, SP-PIFO and PACKS, allowing operators to specify (new) scheduling policies on high-level abstractions that can be deployed on existing switches.

These works have one major limitation: they only cover the case of *single-tenant scheduling*, where *all the traffic* needs to be scheduled following *one single scheduling policy*. For example, they may schedule traffic following the Shortest-Remaining Processing Time policy to minimize FCTs [31, 39, 58]; *or* the FIFO+ policy to minimize tail latency [17, 29]; *or* a Fair Queuing scheme to enforce fairness across pre-defined traffic classes [32, 55, 56].

In practice, however, most networks today (e.g., cloud, data-center networks, and wide area networks) are shared by *multiple* tenants running various applications [63]. Each of these tenants may need to schedule their traffic using different policies in order to achieve their performance goals.

In this chapter, we introduce the vision for QVISOR, a *scheduling hypervisor* that aims at extending the existing programmable schedulers to enable *multi-tenant programmable scheduling* on *existing switches*. With QVISOR, tenants program the scheduling policies for their traffic flows; operators define how tenants should share the available resources; and QVISOR does the rest: deploying the scheduling policies into the underlying hardware.

In this chapter, we try to address the following research challenge:
"Can we simultaneously deploy multiple scheduling algorithms on the scheduling resources of existing programmable data planes?"

Enabling multi-tenant scheduling on commodity switches requires solving two main challenges: (i) providing tenants a substrate where they can *specify* their scheduling policies, and (ii) finding mechanisms to *merge* and *deploy* the specified policies on top of the underlying hardware resources.

In other domains, this problem has long been solved thanks to *virtualization*, which abstracts the hardware resources and allows multiple tenants

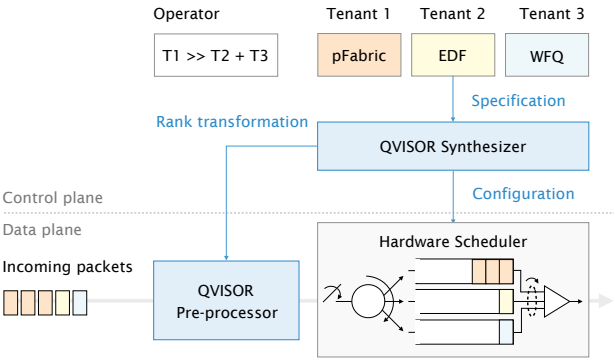


FIGURE 4.1: QVISOR's high-level architecture.

to coexist on the same infrastructure. The key enabler is the *hypervisor*, an interface between the tenants and the hardware, that deploys the applications of the tenants and orchestrates the hardware resources across them. Similarly, to enable multi-tenant scheduling on commodity switches, we need, essentially, a scheduling hypervisor.

A scheduling hypervisor Existing hypervisors typically virtualize compute resources at the end-hosts (CPU, memory, and I/O), allowing multiple tenants to access them while running their applications on virtual machines. Tenants only need to worry about programming their application, without having to mind about what other tenants do, nor about the details of the underlying infrastructure. Operators just specify tenants' access to resources (e.g., offering isolation and/or certain guarantees) and how resource conflicts should be resolved (e.g., by treating some tenants preferentially).

A scheduling hypervisor should follow the same intuition, but virtualizing the scheduling resources of the switch (i.e., the buffer and compute resources to execute the scheduling policies). Tenants should specify the scheduling policies to be used for their traffic on a high-level abstraction, as if they were to run in isolation on dedicated hardware. The operator should define how the scheduling resources should be shared across tenants (e.g., prioritizing traffic from certain tenants). The hypervisor should take care of the rest: combining and deploying the specified policies into hardware.

QVISOR We envision QVISOR, a scheduling hypervisor to virtualize the scheduling resources of commodity switches, allowing them to be shared by multiple tenants (see Fig. 4.1). QVISOR takes as input a specification on

how tenants wish to schedule their traffic, along with a high-level policy by the operator on how the scheduling resources should be shared. QVISOR then comes up with a joint scheduling strategy that follows the per-tenant policies while satisfying the operator's constraints, and deploys it into the underlying hardware.

Key challenges Realizing the vision for QVISOR, requires solving two challenges. First, finding flexible and easy ways for tenants and operators to *specify* their policies. Second, *merging* and *deploying* these policies into the available infrastructure, while being able to *reason* about their performance. The solutions to both challenges have to be co-designed: we can not offer tenants a more-expressive abstraction than what can be later deployed, since it would lead to non-compilable applications, nor a more-limiting one, since it would lead to inefficient resource usage.

Insights We introduce a preliminary design of QVISOR that builds upon the following insights:

- Tenants have the illusion that their traffic is scheduled by a PIFO queue, which they can program using ranks.
- Operators define their policy with a composition language that enables resource sharing and prioritization.
- With these definitions, the problem is reduced to generating a single *joint* scheduling function, that combines the scheduling policies of the tenants and the operator.
- The specification is a 2-layer scheduler where the leafs (resp. root) are the intra- (resp. inter-) tenant policies.
- Multi-tenant policies have higher expressivity than single-tenant ones, but this only affects the worst case. *In practice*, workloads are not always active and do not always overlap, allowing us to multiplex the scheduling resources over time. When workloads do overlap, we resort to the high-level policy of the operator.
- Once we have a synthesized *joint* scheduling function, we can seamlessly deploy it to commodity schedulers.
- Overall, we essentially "trick" single-tenant schedulers to work as multi-tenant programmable schedulers.

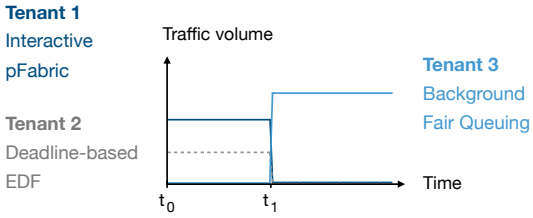


FIGURE 4.2: Example of a data-center workload.

4.2 MOTIVATION

Put yourself in the shoes of an operator managing a network with three tenants, T_1, T_2, and T_3 (see Fig. 4.2). Each tenant runs a different application. Your task is to handle the congestion in a switch, where the workloads of the three tenants coexist. The switch supports a PIFO queue, which you can configure to distribute the available bandwidth across the tenants.

All tenants have specified the scheduling policies that they wish to use for their traffic. T_1 runs an interactive application, sensitive to delay. Thus, T_1 would like to use pFabric's scheduling policy, which prioritizes flows with shorter remaining time, to minimize FCTs [31]. T_2 runs deadline-constrained flows, where meeting deadlines is crucial but delay sensitivity is low. Therefore, T_2 picks the earliest-deadline-first (EDF) algorithm to maximize meeting deadlines [40]. T_3 runs background applications and opts for a Fair Queuing scheme [55].

Based on the tenants' contracts, you know that tenants T_1 and T_2 should share the resources fairly, and should have priority over T_3. Your task is challenging for various reasons:

Problem 1: Scheduling policies clash with each other Within $t_0 < t < t_1$, traffic from tenant T_1 should be scheduled using pFabric, and traffic from tenant T_2, using EDF. How can we achieve this behavior on a conventional scheduler? Both scheduling policies have different objectives, and prioritize packets based on different metrics: EDF prioritizes packets based on their flows' deadlines and pFabric does it based on the remaining flow size [31, 40]. If we naively execute the two scheduling policies simultaneously, they clash. Indeed, most packets from the deadline-constrained flows end up taking the link resources, since the priorities defined by the EDF scheduling

policy are higher than the ones set by pFabric [43]. Thus, to reason about how to combine policies together, we need a way to compare them fairly.

Idea 1: Homogenize scheduling policies We can "normalize" and "quantize" policies into a common scale and granularity. This constrains their behavior and helps us reason about their interaction. Once polices are homogenized, we can compare them fairly, and think about how to merge them into a joint scheduling policy.

Problem 2: The way policies clash changes over time Even if we have a means to homogenize scheduling policies, their behavior is not constant: it depends on traffic, and traffic always changes. Traffic shifts become more drastic as tenants enter or leave the network. For example, at t_1, tenants T_1 and T_2 become inactive and tenant T_3 starts transmitting. T_3 has different performance requirements than T_1 and T_2 and, thus, requires a different scheduling strategy. At the same time, traffic from T_3 should be scheduled with the lowest priority. How can we seamlessly *switch* between policies?

Idea 2: Predict traffic shifts or react upon them We can develop analysis techniques to evaluate how different scheduling policies may work together, either theoretically, offline (e.g., based on worst-case analysis from the given specification) or online at runtime (e.g., based on the latest packets received). We can use the results to predict and actuate upon potential traffic shifts.

If we have an upfront specification of each tenant's policies, we can develop static analysis techniques to evaluate the worst-case scenario for the combined workloads. This allows us to smartly design combined scheduling functions that enforce the desired behaviors, even under adverse conditions. For example, if tenant T_3 should be scheduled with lower priority than T_1 and T_2, we can *shift* all the priorities in T_3's scheduling policy so that, even in the worst case, it does not impact the performance of the other tenants.

If we do not have the specification of all tenants in advance, we can design reaction methods to adapt the scheduling policy at runtime, as tenants enter or leave the network. Similarly to how we deploy forwarding rules when a packet from a new flow arrives to an SDN switch, we can adapt the switch's scheduling policy under certain events. For example, an event-driven controller could synthesize a new scheduling policy after the first packets of a new workload arrived, and deploy it into the data plane. While this may come with challenges, such as emptying the buffers (e.g., if an incoming tenant has priority), or resetting the state of stateful scheduling

functions, we believe that recently-proposed runtime programmable data planes can help lighting the path [64].

These analysis techniques would also be valuable to prevent adversarial workloads from potentially malicious tenants. For example, they could help us develop monitoring techniques to identify such adversarial workloads in the network and automatically stop them in case they ever occurred.

Problem 3: We don't have a standard scheduler PIFO queues provide us with a comfortable abstraction that eases the solution to the different challenges. Indeed, PIFO queues schedule packets with the guarantee that high-priority packets will always be scheduled before low-priority ones. This allows us to reduce the problem of designing a scheduling hypervisor into the one of analyzing how prioritization functions interact with each other and how they can be combined effectively. Existing switches today, however, do not support PIFO queues. Instead, they run, at most, PIFO approximations on top of strict-priority queues or FIFO queues [1, 33, 41, 42]. These PIFO approximations support different operations, and they offer various types of performance guarantees. How can we deploy multi-tenant policies on top of these schedulers?

Idea 3: Develop standard APIs or build synthesizers We can build standard APIs that allow the deployment of combined scheduling policies into existing schedulers. Alternatively, we can leverage program synthesis to transform high-level scheduling specifications down to the available hardware resources of existing switches.

To run on top of an existing scheduler, a scheduling hypervisor would need a set of APIs to interact with the various configuration parameters of the scheduler (e.g., to dedicate a set of priority queues to a certain tenant). As such, we would need to abstract the operations that the scheduler can support, and provide them to the hypervisor as a design space. With them, the hypervisor would be able to come up with a possible configuration, that satisfied the specification and reason about its guarantees. Further, given the set of operations supported by the scheduler, the hypervisor could leverage program synthesis to come up with a scheduling policy that fit into the available resource constraints, even if it could only satisfy a part of the input specification [65].

4.3 QVISOR OVERVIEW

We present a preliminary design for QVISOR, that defines its architecture and grounds the foundation for future research.

At the high-level, we envision QVISOR to work as follows (cf. Fig. 4.1). First, every tenant must specify the scheduling algorithms that should schedule the traffic within their flows. Second, the network operator must define a policy on how the scheduling resources should be distributed across tenants. Based on these inputs, QVISOR should: (i) synthesize a joint scheduling policy that follows the policies specified by the tenants, while respecting the constraints given by the operator; and (ii) deploy this policy into the underlying hardware to apply it to incoming packets.

Accordingly, we propose a design with two parts: a *synthesizer* that runs in the control plane and generates the joint scheduling policy from the inputs, and a *pre-processor* that runs in the data plane and "prepares" packets such that they can be scheduled with the synthesized policy.

In the following, we describe QVISOR's inputs (§4.3.1), synthesizer (§4.3.2) and pre-processor (§4.3.3). We also discuss how to deploy QVISOR on existing hardware schedulers (§4.3.4).

4.3.1 *QVISOR inputs*

Per-tenant scheduling policies Tenants define the scheduling policy that they want to use for their traffic as a tuple composed of a traffic subset and a scheduling algorithm. For example, tenant $T_1 = \{\mathcal{P}_1, pFabric\}$ defines a set of packets $\mathcal{P}_1$, to be scheduled with the *pFabric* policy. Another tenant, $T_2 = \{\mathcal{P}_2, STFQ\}$, comprises the set of packets $\mathcal{P}_2$ which have to be scheduled using the $STFQ$ algorithm. Note that a *tenant* refers to a traffic segment (e.g., from a application), not necessarily a physical tenant.

Packet labels For QVISOR to process incoming packets, it requires tenants to identify their packets with two labels: the *tenant identifier*, and the *packet rank*. Intuitively, the tenant identifier allows QVISOR to decide how to schedule the packet with respect to packets from other tenants, and the *packet rank* describes how to schedule the packet with respect to other packets from the same tenant. Packet ranks define the priority with which packets should be scheduled based on the rank function picked by the

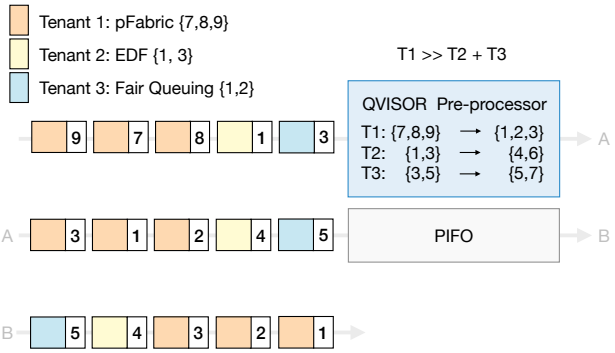


FIGURE 4.3: QVISOR's transformations.

tenant. Ranks can be computed at the end-host or at the same switch, but always have to be specified before reaching QVISOR's pre-processor.

Operator's specification The operator defines a high-level inter-tenant policy using a simple string with the tenant identifiers, separated by three possible operators: $\{>>\}$, $\{>\}$, and $\{+\}$. The first operator indicates that the preceding tenant(s) should have strictly-higher priority than the following one(s), mandating isolation. The second, states that the preceding tenant(s) should be preferentially treated with respect to the following tenant(s). In this case, the priority is applied in a best-effort manner. The third operator indicates that preceding and following tenant(s), should share the resources. For example, the specification $\{T_1 >> T_2 > T_3 + T_4 >> T_5\}$ indicates that tenant T_1 should have strictly-higher priority than tenants T_2, T_3, and T_4, which in turn should have strictly-higher priority than T_5. It also defines that tenant T_2 should have higher priority, whenever possible, than T_3 and T_4, and that tenants T_3 and T_4 should share their resources.

4.3.2 *QVISOR synthesizer*

Given the input policy specifications of the tenants and the operator, the goal of the synthesizer is to generate a joint scheduling function that combines the algorithms of the different tenants, such that they can be simultaneously used, while satisfying the high-level requirements of the operator.

We express the joint scheduling function as a set of rank transformation functions to be applied to the ranks of the incoming packets. These functions

are then deployed into the pre-processor and applied at line rate to incoming packets. QVISOR's synthesizer supports two types of transformation functions: rank-shift and rank-normalization operations.

Rank shift It allows the prioritization of traffic from certain tenants over others. For instance, to prioritize traffic from T_1 over T_2, we can *shift* the ranks of all packets of T_1 such that they have lower ranks than all packets from T_2. Alternatively, we can *shift* all ranks of T_2 such that they are higher than the ones of T_1. If the rank distributions are bounded and known in advance, we can implement most priority operations by just applying shifts to the ranks of the different tenants.

Rank normalization As shown in §4.2, naively scheduling packets with various rank functions simultaneously on the same scheduler can be detrimental. The normalization function consists in bounding the ranges of a given rank function, and quantizing its ranks into discrete levels such that they can be fairly compared with the (normalized) ranks of other tenants. With this function, different tenants can be scheduled simultaneously with a higher fairness and without loosing their intra-tenant scheduling behavior.

4.3.3 *QVISOR pre-processor*

For each incoming packet, QVISOR's pre-processor parses the packet headers and extracts the *tenant identifier* and the *packet rank*. It uses these tags to query the transformation functions that should be applied to the packet, based on the results of the synthesizer. It executes them, updates the rank, and forwards the packet to the hardware scheduler.

Example Fig. 4.3 shows QVISOR's pre-processor handling a sequence of packets from tenants T_1, T_2, and T_3, ranked using pFabric, EDF, and FQ algorithms, respectively. The operator has specified that traffic from T_1 should have higher priority than traffic from T_2 and T_3, which should share the resources. Given these inputs, the synthesizer has generated three transformation functions: packets from T_1 carrying ranks $\{7,8,9\}$, have to be re-labeled with ranks $\{1,2,3\}$, respectively; packets from T_2 with ranks $\{1,3\}$ have to be transformed into $\{4,6\}$; and packets from T_3 with ranks $\{3,5\}$, into $\{5,7\}$. The pre-processor applies these transformations and forwards the traffic to the scheduler (a PIFO queue), which sorts the packets based on ranks. As a result, the output sequence satisfies the input

specifications: all packets from T_1 have the highest priority, and the traffic from tenants T_2 and T_3 share the resources evenly. At the same time, traffic within each tenant is scheduled in order of rank, following the specified ranking algorithm and achieving their desired performance.

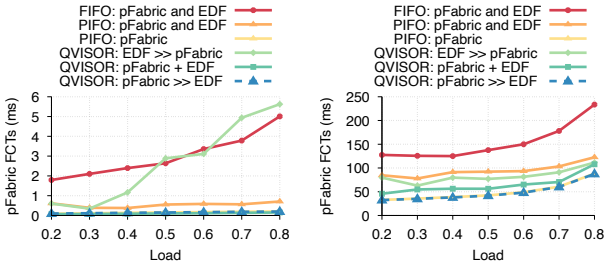
4.3.4 *QVISOR on existing schedulers*

To run QVISOR on top of an existing scheduler, we have to consider two aspects. First, such schedulers consist of the baseline hardware (e.g., a set of FIFO queues), and some pre-processing function (e.g., to map packets to queues, or to decide which packets to admit). To virtualize them, we need to virtualize both, the hardware and the functions on top. Second, differently from PIFO queues, these schedulers do not always guarantee perfect packet sorting based on ranks. Thus, QVISOR may need additional configurations to fulfill the specification (e.g., dropping packets above a certain rank or dedicating queues to some tenants). As such, in order for QVISOR to run on existing schedulers, it should know what packet-processing operations they support and what guarantees they provide. With this information, QVISOR should be able to synthesize a set of operations to satisfy the specification while staying within the available resources.

For example, in Fig. 4.3 we need to prioritize traffic from T_1. In most existing schedulers [1, 42, 43], we can only guarantee such prioritization by allocating dedicated queues. This is, if we have a scheduler with five queues, we can map traffic from T_1 to the three highest-priority queues, and traffic from T_2 and T_3 to the two lowest-priority queues. Then, we need to run two mapping functions in parallel: one to map traffic from T_1 to the first three queues based on the pFabric policy, and another to map traffic from T_2 and T_3 to the last two queues, based on the EDF and FQ algorithms fairly combined.

4.4 PRELIMINARY EVALUATION

In this section, we show the potential of QVISOR and prove that it is possible to simultaneously run multiple scheduling algorithms on top of a single-tenant scheduler. We implement a proof-of-concept version of QVISOR on Netbench [26], a packet-level simulator. We evaluate it when scheduling the traffic from two tenants on a data-center network. We use a leaf-spine



(a) (0,100KB): Mean FCTs (b) [1MB,∞): Mean FCTs

FIGURE 4.4: QVISOR's performance.

topology with 144 servers connected through 9 leaf and 4 spine switches, and set the access and leaf-spine links to 1Gbps and 4Gbps, respectively. The first tenant runs a data-mining workload that needs to be scheduled with the pFabric algorithm. The second tenant runs 100 flows that transmit at a constant bit-rate of 0.5Gbps between pairs of servers picked uniformly at random, which have to be scheduled following the EDF algorithm.

We measure the flow completion time of the pFabric traffic, under various loads, when the workloads are scheduled by: a FIFO queue, a PIFO queue, and QVISOR on top of a PIFO queue under three different policies: when the pFabric traffic is prioritized, when the EDF traffic is prioritized, and when both tenants share the resources. We also analyze the ideal case in which there is only pFabric traffic in the network.

Fig. 4.4 illustrates the resulting flow completion times for both, small and big flows. We see how, FIFO and QVISOR with a policy that prioritizes EDF traffic, are the most detrimental cases for pFabric. This is expected: for the first, the FIFO scheduler can not prioritze traffic, and thus the pFabric policy becomes useless; and for the second, pFabric traffic gets blocked behind EDF traffic. We also see how naively executing the two scheduling policies on a PIFO queue is detrimental for pFabric, since most of the pFabric packets get deprioritized after the ones of EDF. Instead, when we use QVISOR with a policy that either prioritizes pFabric, or lets both tenants share the resources fairly, pFabric's traffic achieves a performance that is either ideal (i.e., equivalent to when running in isolation), or very close.

4.5 LOOKING FORWARD

The path towards scheduling virtualization is still plenty of open problems. In the following, we discuss a subset of them.

Increasing specification expressivity In our preliminary QVISOR design, tenants specify their scheduling algorithms using ranks, and operators define their policies using three basic operators (§4.3.1). We expect future research to explore novel QVISOR designs which can support more expressive specifications. For example, recent research has proposed more complex abstractions such as PIFO trees or Directed Acyclic Graphs (DAGs) [17, 19, 20, 36, 66, 67], which offer a higher degree of expressivity for both tenants and operators. With them, tenants can specify hierarchical and non-work-conserving scheduling algorithms, and operators can specify complex relations across tenants (e.g., multiple tiers). How to support these abstractions on QVISOR is an open question.

Compiling scheduling policies into hardware We also expect future research to focus on the interaction between QVISOR and existing schedulers. As discussed in §4.3.4, to work on existing infrastructures, QVISOR needs to be aware of the operations that they support. These operations have to be abstracted and provided to QVISOR as a domain-specific language, which it can then use to synthesize scheduling configurations. QVISOR also needs to know the guarantees that these operations offer, to be able to reason about whether the synthesized configurations can satisfy the specifications.

We could frame QVISOR's goal as a *compilation* problem where, given a high-level specification of the scheduling policies, and the design space of the operations supported by hardware, the objective is to come up with a set of operations that satisfies the policies. In a second step, we could take a *synthesis* approach, where QVISOR would not just fail if the desired policy could not be compiled, but would propose *partial specifications* implementable on the available resources. QVISOR would output the proposed configuration, with the supported specifications and the offered guarantees.

Optimizing configurations at runtime We expect future designs of QVISOR to optimize at runtime both, the joint scheduling policy (e.g. computing transformation functions at line rate, based on the distribution of the lat-

est packets), and the hardware-scheduler configurations (e.g., reallocating queues mapped to a tenant if the tenant is not transmitting).

Multi-objective scheduling algorithms In QVISOR, we have asked ourselves whether it is possible to run multiple scheduling algorithms on a single infrastructure. We have considered the case of having multiple tenants with different objectives. Another perspective would be to analyze whether we can achieve multiple objectives simultaneously *on the same traffic*. For example, Fair Queuing schemes enforce fairness, but also help in reducing FCTs, since they implicitly prioritize short flows. Multi-objective scheduling algorithms add another dimension to the QVISOR problem, offering new opportunities to combine traffic with similar requirements.

Synthesizing scheduling algorithms With the advent of programmable scheduling, we have more abstractions than ever to represent scheduling algorithms [17, 20] as well as algorithms for various objectives. Could we abstract and generalize this knowledge to create scheduling algorithms for arbitrary performance objectives? This is, given a workload and a performance goal (e.g., as a utility function or an SLA), could we synthesize the optimal scheduling algorithm?

Cross-device virtualization Recent works have managed to implement multi-tenant scheduling policies *at the end-hosts*, allowing the virtualization of the scheduling resources across tenants at the NICs [36, 66]. QVISOR enables multi-tenant scheduling *in the network* at commodity switches. As such, similarly to works in other domains [68, 69], we expect future research to propose mechanisms to orchestrate the scheduling virtualization from a *network-wide* perspective.

4.6 RELATED WORK

Packet scheduling Historically, most research on packet scheduling has focused on the design of algorithms with *a single performance objective*, such as enforcing fairness across traffic classes [29, 32, 55–57], minimizing FCTs [31, 39, 58], or minimizing jitter [17, 29]. Recently, researchers have explored the possibility of a universal scheduler capable of replicating any given algorithm [17, 18]. The lack of such a silver bullet [18] has prompted the development of programmable scheduling [19] and the emergence of new abstractions to enable it [17, 20, 36, 59, 66], some of which are compatible

with commodity hardware [1, 33, 41, 42, 44, 59]. As a result, while operators today cannot satisfy *all* scheduling objectives simultaneously, they can approximate most *individual* policies on existing devices.

In QVISOR, we bring up a new research question: is it possible to simultaneously implement *multiple* scheduling algorithms on a shared hardware scheduling infrastructure? We answer positively by proposing a scheduling hypervisor that navigates the space between programmable scheduling, and the ideal concept of a universal packet scheduler.

Virtualizing programmable networks Previous works have focused on virtualizing programmable networks, from their end-host counterparts to the in-network resources [64, 70–81]. While most solutions target software switches, SmartNICs or NetFPGAs [74, 77, 79, 80], a few focus on hardware switches [64, 72–74, 82]. Among them, [73, 74, 82] optimize resource sharing at compile time by combining multiple applications into a single program, [64] enables runtime reprogramming of switch data planes, and [72] facilitates the dynamic sharing of switch resources across applications. Only [80, 81] address the virtualization of scheduling infrastructure in programmable switches, but they do not virtualize programmable scheduling policies.

4.7 CONCLUSION

We introduced a vision for QVISOR, a scheduling hypervisor to virtualize the scheduling resources in commodity switches, enabling efficient resource sharing among multiple tenants. QVISOR acts as an intermediary between tenants and the underlying hardware. It allows tenants to program their own policies for their traffic, and operators to define a high-level policy on how resources should be shared. With these inputs, QVISOR creates a joint scheduling strategy that combines the tenant policies, while satisfying the operator's conditions, and deploys it to hardware. As a result, QVISOR brings multi-tenant programmable scheduling to conventional switches.

5

ACC-TURBO

5.1 INTRODUCTION

Pulse-wave DDoS attacks have recently managed to take down critical network infrastructure while causing enormous financial and reputational damages [83–86]. In contrast to conventional DDoS attacks, which grow steadily and persist longer in time, pulse-wave DDoS attacks consist of high-rate short-lived bursts. Each burst typically uses a different attack vector (e.g., NTP, DNS, Memcached) and can reach hundreds of Gbps [87–89].

The threat in pulse-wave attacks resides in that they target the Achilles' heel of existing DDoS mitigation systems: their reaction time. Most in-network DDoS defenses today (both in research and production) rely on some sort of offline facility that can either directly scrub traffic [90–95], orchestrate a routing-based defense [96, 97], or deploy an in-network pre-configured mitigation [98, 99]. The time required for an in-network defense to reach this external facility and deploy the corresponding defense can be in the order of seconds to minutes [88]. Pulse-wave attacks exploit this vulnerability by sending traffic pulses that force the DDoS defense to repeat this control loop over and over. By keeping the defenses in a constant transient state, pulse-wave attacks manage to make them ineffective. If the defenses rely on traffic redirection, pulse-wave attacks may even produce route flapping in the network [100].

Designing a pulse-wave defense is intricate. Like conventional-DDoS defenses, an ideal pulse-wave defense needs to be, first, *generic*, to identify a wide variety of attack vectors at different granularity [101, 102]. Generic techniques require unsupervision and incur the risk of misclassifying traffic. Thus, an ideal defense also needs to be *measured* in responding to attacks [103]. Most DDoS defenses fail at the first condition. For example, signature-based defenses [91, 98, 99] are not generic. They only cover a small set of vectors and can not keep up with the constantly-growing list of new attacks [101, 102]. Similarly, most congestion-management tools (e.g., heavy-hitter detectors or active queue management) lack granularity: they only work at the per-flow level or the whole-traffic level. Other

defenses fail at satisfying the second condition. For example, drop- or routing-based defenses [104, 105] strongly degrade performance in case of misclassification.

Aggregate-Based Congestion Control (ACC) was proposed two decades ago, *satisfying the two* design conditions [106]. ACC is a canonical mechanism to reduce the impact of congestion caused by *generic* traffic aggregates, with a *measured* bandwidth control. At a high level, ACC is a feedback loop that iteratively: *(i)* infers the aggregates causing the congestion; before *(ii)* reducing their throughput to a reasonable level. To infer the aggregates, ACC clusters the headers of packets dropped by a Random Early Detection (RED) queuing discipline. ACC then rate-limits the inferred aggregates to keep the total traffic throughput below the link capacity.

While ACC is effective at inferring and controlling conventional DDoS attacks, it fails at mitigating pulse-wave attacks, as it cannot keep up with the required fast reaction times. The reason is two-fold: (i) ACC relies on *offline* inference and control mechanisms, which run on either a separate server or a control plane, and (ii) ACC leverages a threshold-based defense activation. By running *offline*, ACC suffers from slow reaction time, which opens the door to pulse-wave DDoS attacks. By relying on threshold-based activation, ACC either further slows down reaction time or increases the probability of false positives, which negatively impacts its performance. In our experiments, even with the best configuration, ACC drops $\approx 20\%$ of benign traffic in the event of a pulse-wave attack (§5.2.2).

	Activation	Inference	Control
ACC [106]	Threshold-based	Offline clustering on RED drops	Uniform rate limiting using estimated rates
ACC-Turbo	Always-on	Online clustering on all traffic	Programmable scheduling using exact rates

TABLE 5.1: *ACC-Turbo* techniques vs. ACC.

Our work In this chapter, we redesign ACC for pulse-wave DDoS defense. We propose *ACC-Turbo*: the *first* sub-second-reaction-time aggregate-based congestion control mechanism that mitigates pulse-wave DDoS attacks by running at line-rate on commodity hardware. *ACC-Turbo* strategically

combines two key techniques: *online clustering* directly in the data plane to infer attacks, and *programmable scheduling* to mitigate them (cf. Table 5.1).

First, *ACC-Turbo* offloads the clustering process to the data plane and analyzes *all* the traffic at line rate (instead of just a sample). This allows *ACC-Turbo* to speed up reaction time substantially. Further, since data-plane processing does not impact traffic latency, *ACC-Turbo* runs the clustering algorithm *continuously*. This eliminates the need for a threshold-based activation, which can be vulnerable to pulse-wave attacks and opens the door to potential false positives. The *always-on* design enables *ACC-Turbo* to anticipate the congestion events, achieving yet-faster reaction time.

Second, *ACC-Turbo* uses programmable scheduling to deprioritize malicious traffic instead of dropping or rate-limiting it. Doing so has three advantages: (i) it accommodates fine-grained assessments which can adjust to the requirements of individual aggregates, increasing fairness; (ii) it adapts at the per-packet level, being able to react to traffic variations rapidly, achieving faster and more-accurate bandwidth allocations; and (iii) it only leads to hard drops under congestion, being transparent otherwise.

Evaluation We implement *ACC-Turbo* in P4 [28] and run it on programmable switches (Intel Tofino). We show that *ACC-Turbo* effectively mitigates pulse-wave DDoS attacks in real-time (i.e., less than 1s reaction time), rapidly adapting to attack variations. We also compare *ACC-Turbo* with Jaqen [98], a state-of-the-art DDoS defense. We show that *ACC-Turbo* is *at least* 10$\times$ faster than Jaqen in mitigating attacks while also being safer.

Contributions The main contributions of this chapter are:

- An online-clustering approach to infer pulse-wave DDoS attacks at scale directly from the network (§5.4).
- A scheduling algorithm to mitigate pulse-wave attacks and minimize their impact on background traffic (§5.5).
- An implementation of *ACC-Turbo* in Python and P4 (§5.6).
- A comprehensive evaluation showing *ACC-Turbo*'s practicality and its ability to run on hardware (§5.7, §5.8).

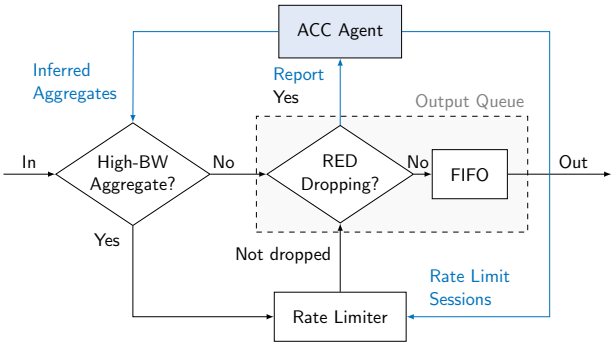


FIGURE 5.1: ACC architecture [106].

5.2 BACKGROUND

In this section, we review the design of ACC (§5.2.1), as well as its limitations in mitigating pulse-wave DDoS attacks (§5.2.2).

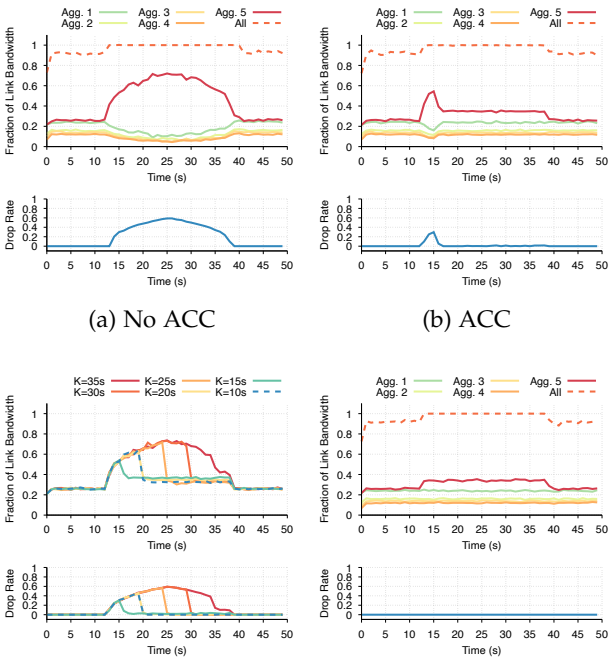
5.2.1 *Aggregate-based Congestion Control*

Aggregate-based Congestion Control is a mechanism for inferring and controlling high-bandwidth aggregates that persistently overload a link in the network. In this chapter, we focus on its local version, which runs on the switch that gives access to the congested link.[1]

Fig. 5.1 shows the architecture of an ACC-enabled switch. The core of ACC is a RED module, implemented on top of a FIFO queue. This module monitors the average queue size of the FIFO queue and drops packets probabilistically depending on its size. If the FIFO queue is almost empty, all incoming packets are accepted. As the queue grows, the probability of dropping an incoming packet increases. When the queue is full, the probability is at its maximum, and all incoming packets are dropped.

Whenever the RED module decides to drop a packet, it reports the header of the dropped packet to an ACC Agent. The ACC Agent periodically analyzes the packet headers of all dropped packets and tries to infer the traffic aggregates responsible for the congestion. When the ACC Agent identifies

1 The original work also includes a "pushback" mechanism to extend the rate-limiting policies to upstream switches. This part is out of our scope.



(a) No ACC (b) ACC

(c) ACC: Impact of K (d) *ACC-Turbo*

FIGURE 5.2: Performance of ACC and *ACC-Turbo* in the original experiment [106].

an aggregate, it creates a rate-limiting session to police its throughput. From that point onwards, all upcoming packets belonging to the aggregate are rate-limited before being processed by the RED module.

Identifying congestion The ACC Agent is activated when the output queue experiences sustained high congestion. This occurs when the drop rate in the output queue exceeds a pre-defined value, p_{high}, during a certain time period, K, which is also pre-defined.

Inferring aggregates The ACC Agent infers traffic aggregates based *solely* on IP prefixes. At the high level, it extracts a list of either source or destination IP addresses that account for more than twice the mean number of packet drops and clusters them into 24-bit prefixes. To minimize collateral

damage, ACC then walks down the prefix subtree for each of the aggregates, trying to obtain longer prefixes that still contain most of the packet drops.

Rate-limiting aggregates For each aggregate inferred, the ACC Agent then computes the bandwidth to which it should be rate-limited. This limit is computed such that the drop rate at the output queue, gets below a pre-defined value, p_{target}. To that end, the ACC Agent, first, sorts the list of inferred aggregates by their number of drops (highest, first). Then, it computes the excess arrival rate at the output queue, R_{excess}, defined as the amount of traffic that should be dropped in order for the drop rate to get below p_{target}. Finally, the ACC Agent determines the minimum number of aggregates that should be rate-limited, $|\mathcal{A}|$, and the rate to which they should be limited, L, such that the total rate is reduced by R_{excess}:

$$\sum_{i=1}^{|\mathcal{A}|}(Aggregate[i].rate - L) = R_{excess}$$

With this design, ACC manages to be generic (inferring attacks agnostically to their characteristics) and measured (rate-limiting inferred attacks instead of just dropping them).

Experiment We illustrate ACC's performance with packet-level simulations (cf., §5.8 for setup details). We reproduce ACC's original experiment [106]. It consists in scheduling five aggregates (1-5) over a bottleneck link using FIFO, and ACC, respectively. Aggregates 1-4 are constant-bit-rate flows. Aggregate 5 is a variable-rate flow which represents an attack, and starts increasing (resp. decreasing) its rate at $t = 13s$ (resp. $t = 25s$).

Fig. 5.2a and Fig. 5.2b show the bandwidth share across the five aggregates (top) and the drop rate at the output queue (bottom) when no protection (i.e., FIFO) and ACC are used, respectively. ACC is configured with $p_{high} = 0.1$, $K = 2s$, and $p_{target} = 0.05$. Table 5.2 details the rest of parameters. Without ACC, we see how the attack traffic captures most of the link bandwidth, degrading the performance of the other aggregates. With ACC, the attack is efficiently mitigated. Indeed, when the drop rate at the output queue exceeds p_{high}, the ACC Agent infers the attack and rate-limits it sufficiently to reduce its impact on background traffic.

Finally, we also see how the reaction time of ACC is $\approx 4s$. This is the time since the first attack packets arrive (at $t = 13s$), until the defense is deployed (at $t = 17s$). This time is mostly driven by the monitoring-window size, K. Indeed, for smaller K values, ACC checks more often whether p_{high}

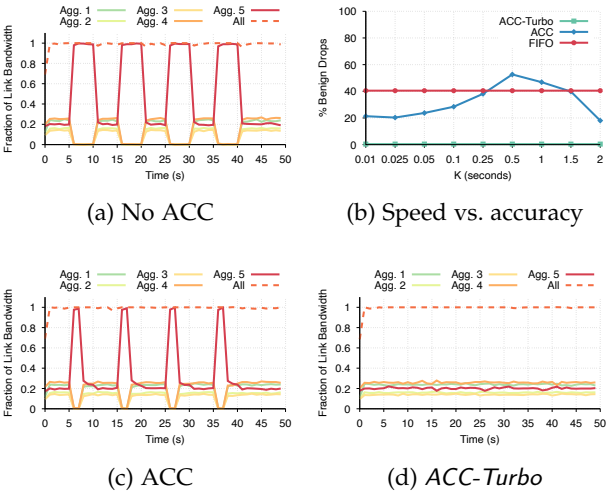
Name	Definition	Value
K	Sustained-congestion period	$2s$
p_{high}	Sustained-congestion droprate	0.1
p_{target}	Target droprate	0.05
k	Exponential moving average interval	$0.1s$
Sessions	Maximum number of rate-limiting sessions	5
Release Time	Minimum time required for an aggregate to be released after rate-limiting starts	$10s$
Free Time	Minimum time required for an aggregate to be released after it is detected to "behave"	$20s$
Cyc. Time	Time to revisit the aggregate	$5s$
Init. Time	Time to revisit the aggregate in initial phase	$0.5s$

TABLE 5.2: List of the ACC parameters used in the simulations.

is exceeded, being able to identify faster whenever that occurs. A lower K, however, does not always imply a faster reaction time (cf. Fig. 5.2c). For example, $K = 10s$ achieves a slower reaction time than $K = 15s$. The reason is that when $K = 10s$, even though the threshold is checked first at $t = 10s$, it is not triggered until $t = 20s$, when p_{high} is reached.

5.2.2 *Limitations of ACC*

While ACC manages to defend against conventional DDoS attacks successfully, it is vulnerable to pulse-wave DDoS attacks. The reason is two-fold. First, ACC uses *offline* inference and control mechanisms, which run on either a separate server or a control plane [106]. Pulse-wave DDoS attacks are specially crafted to target the time required by such control planes to compute and deploy the right DDoS defense. They do so by sending short high-rate traffic pulses which morph over time. These pulses congest the link resources while the inference and control processes are running. By the time the inference and control processes manage to converge and ACC mitigates the attack, another pulse comes in, forcing the control loop to start the mitigation process all over again.



(a) No ACC (b) Speed vs. accuracy

(c) ACC (d) *ACC-Turbo*

FIGURE 5.3: Performance under morphing attack.

The second reason is that ACC relies on a threshold-based defense activation, which introduces a second vulnerability to pulse-wave attacks: If we configure a too small threshold (to speed up reaction time), the probability of false positives (i.e., benign traffic bursts identified as attacks) increases, as we prove in §5.8. In contrast, if we configure a too large threshold (to maximize accuracy), reaction time gets slower, opening the door to pulse-wave attacks. In ACC, false positives are especially concerning under attack, given that (i) ACC rate-limits all aggregates to the same amount, and (ii) its rate-limiting policies have long-lasting effects (cf. §5.2.1).

Example We evaluate ACC's mitigation efficiency in the case of a pulse-wave attack composed of four vectors (starting at 5*s*, 15*s*, 25*s*, and 35*s*). For simplicity, we represent the four pulses as a single "attack" aggregate (i.e., aggregate 5). We leverage four constant-bit-rate flows as background traffic (i.e., aggregates 1-4), which transmit at $\approx$ the link capacity.

Fig. 5.3a and Fig. 5.3c illustrate the bandwidth share at the output link when FIFO and ACC (configured as in §5.2.1) are used. We see how ACC fails at mitigating the attack, only managing to defend the second half of attack pulses. By the time the monitoring window K is triggered, and p_{high} is reached, the pulse-wave attack has already managed to throttle benign traffic. Fig. 5.3b shows how reducing the size of the monitoring window,

K, does not help. In fact, due to false positives, the performance also gets bounded for the smallest K values, with $\approx 20\%$ of benign traffic dropped.

Given the incapacity of ACC to mitigate pulse-wave DDoS attacks, we propose *ACC-Turbo*. With an under-second reaction time, and not using a threshold-based activation, *ACC-Turbo* successfully mitigates *both* conventional *and* pulse-wave DDoS attacks (cf., Fig. 5.2d, resp. Fig. 5.3d).

5.3 OVERVIEW

In this section, we present the threat model (§5.3.1), and introduce an overview of *ACC-Turbo*'s design (§5.3.2).

5.3.1 *Threat model*

Attacker The attacker's objective is to generate a pulse-wave DDoS attack (i.e., a series of short-duration high-rate traffic pulses) towards a critical link of the network in order to exhaust its capacity and prevent legitimate flows from using it. To that end, the attacker is free to generate *any* kind of attack traffic. For instance, the attacker can rely on: (i) a botnet of infected devices which directly floods traffic towards the link; (ii) reflection and amplification techniques, which send spoofed requests to open servers such that their responses cross the link; or even (iii) complex link-flooding attacks which exchange low-rate flows from numerous sources to numerous destinations such that they also cross the link [107].

System model Same as ACC, *ACC-Turbo* runs on the switch that gives access to the critical link. This switch can be, e.g., an edge-router at an ISP or IXP, with a bigger input capacity than the output link's bandwidth. *ACC-Turbo* analyzes all traffic entering the switch (including attack traffic) and processes each packet individually. We consider that *ACC-Turbo* only looks at the packet headers, and it performs limited computations. (This is imposed by the fact that we want *ACC-Turbo* to run at line rate on programmable switches [28]).

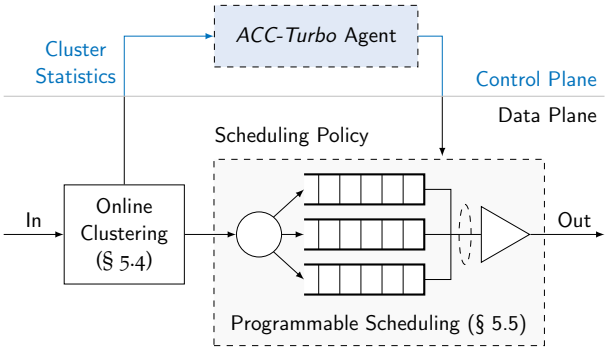


FIGURE 5.4: *ACC-Turbo* architecture.

5.3.2 *ACC-Turbo design*

ACC-Turbo is a switch-native aggregate-based congestion control mechanism that mitigate pulse-wave DDoS attacks on programmable switches. It consists of an *online-clustering* module that runs entirely in the data plane and a *programmable-scheduling* module that runs in both the control plane and the data plane (see Fig. 5.4). With this hybrid design, *ACC-Turbo* balances the need for fast reaction (by running the inference process in the data plane) and accuracy (by dedicating most data-plane resources to inference while offloading the least time-sensitive parts to the control plane).

The online-clustering module extracts a set of features from the headers of arriving packets and uses them to cluster similar packets together. It runs in the data plane, processing *all* packets at line rate. It also runs *continuously*, regardless of whether there is congestion, eliminating the need for a threshold-based activation. Running continuously also allows *ACC-Turbo* to anticipate its inference decisions, speeding up reaction time.

The programmable-scheduling module involves both the control and the data plane. The control plane periodically polls information about the extracted clusters, including statistics about the clusters and their exact arrival rates. Then, it uses this information to assess the probability that each cluster contains attack traffic, and derives a scheduling policy for each cluster. The scheduling policies aim at deprioritizing clusters with a higher probability of belonging to an attack. *ACC-Turbo* finally deploys these policies to the data plane and applies them to incoming packets.

By using programmable scheduling, *ACC-Turbo* adapts to traffic variations at a per-packet granularity and rapidly reacts to attack changes. Further, programmable scheduling enables *ACC-Turbo* to run continuously, even in the case of no attack. This is because programmable scheduling does not hurt traffic: it only drops packets in case of severe congestion (being transparent otherwise) and starts by those with higher chances of being part of an attack.

5.4 TRAFFIC-AGGREGATE INFERENCE

We now describe the theoretical basis behind *ACC-Turbo*'s inference component. First, we phrase the problem formally and propose a practical solution based on online clustering (§5.4.1). Second, we introduce the design decisions that make *ACC-Turbo* implementable in existing programmable switches (§5.4.2). The resulting clustering algorithm can be found in alg. 4.

5.4.1 *Problem definition*

Let us define a packet p as a set of features $\mathcal{F}$, where each feature corresponds to a field from the packet header (e.g., *sport, dport, ip.ttl, ip.proto*). For each feature $f \in \mathcal{F}$, packet p has a specific value associated: p_f. We distinguish two types of features: *ordinal* features, for which closer proximity between their values implies stronger similarity (e.g., *ip.src, ip.dst, ip.len, ip.ttl*), and *nominal* features, for which closer values do not necessarily imply similarity (e.g., *sport, dport, ip.proto*).

Let us define an aggregate a as the same set of features $\mathcal{F}$. For each *ordinal* feature f, the aggregate a has a *range* of values associated: $f(a) = [min_f(a), max_f(a)]$. For each *nominal* feature f, aggregate a has a *set* of discrete values associated $f(a) = \{min_f(a), ..., max_f(a)\}$. With this definition, aggregate a represents *all* the packets with feature values included in its ranges and sets.

Objective At the high-level, given a set of incoming packets, our goal is to infer a set of aggregates that represent *all* the observed packets as *precisely* as possible. We need to limit the number of aggregates to infer by a parameter, $|\mathcal{A}|$. Otherwise, one could list each packet as a separate aggregate and obtain perfect precision.

Definition 5.1 (Aggregate-inference problem)**.** *Let $\delta_f(a)$ be the cost of an aggregate a, for feature f, which measures the number of values that it represents. For ordinal features, $\delta_f(a) = max_f(a) - min_f(a)$. For nominal features, $\delta_f(a) = |f(a)|$ (i.e., the number of values in the set). Let $\delta(a) = \prod_{f \in \mathcal{F}} \delta_f(a)$ be the cost of the entire aggregate, as the product of each individual feature cost. For a given feature f, a set of packets $\mathcal{P}$, and a limit on the number of inferred aggregates ($|\mathcal{A}|$), find the set of aggregates $\mathcal{A}^* = a_1, ...a_{k'} (k' \leq |\mathcal{A}|)$ that represent $\mathcal{P}$ and minimize $\delta_f(\mathcal{A}^*) = \sum_{a \in \mathcal{A}^*} \delta_f(a)$.*

The presented problem is equivalent to the intent-inference problem in [108], which is NP-hard. As such, we propose a heuristic solution based on online clustering. Our solution aims at approximating the optimal result while enabling per-packet processing. The cost function in Def. 5.1 estimates the number of different packets that *a* represents, but it is also a measure of similarity of the packets in *a*. Indeed, packets that can be represented in a narrow aggregate have higher similarity than those which require a broader aggregate. With this intuition, we build an online-clustering algorithm that groups similar packets by minimizing the cost in Def. 5.1.

Definition 5.2 (Online-clustering framework [109])**.** *For a sequence of points p in $\mathcal{P}$, maintain a collection of $|\mathcal{C}|$ clusters such that, when each input point p is presented, either it is assigned to one of the current clusters or it starts off a new cluster while two existing clusters are merged into one.*

The online-clustering framework is characterized by an endless stream of data, where each data sample (i.e., packet) is seen only once: it arrives, is processed, and it departs, never to return. As such, the algorithm must take an irrevocable action after the arrival of each point [110]. Note the difference to the *streaming* case, where the job is finite, and the algorithm can pass again through the data to fine-tune the clustering result [111, 112].

5.4.2 *Design decisions*

The proposed framework allows three design decisions: the number of clustering possibilities that should be evaluated at each iteration, the type of information that should be stored about each cluster, and the distance that should be used to assess the clustering decisions. We make these decisions with the goal of maximizing performance while staying within the resource constraints of existing data planes (to achieve an in-network design).

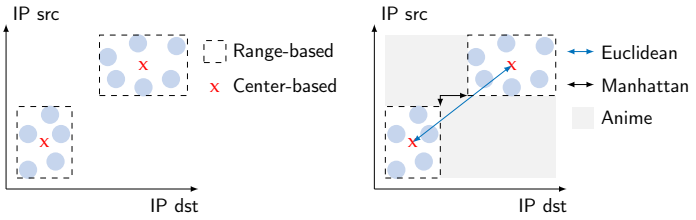


FIGURE 5.5: Cluster representations and distances.

Clustering search (fast vs. exhaustive)

When a new packet arrives, the online-clustering algorithm can either: (i) merge the new packet to its closest cluster or (ii) merge two existing clusters and create a new cluster for the new packet.

*[✓] **Fast search*** If the clustering algorithm only supports step (i), we call it *fast*. This approach follows a linear search and requires only $|\mathcal{C}|$ distance computations: one for each existing cluster. *ACC-Turbo* relies on this type of search because it can be implemented on programmable data planes.

*[X] **Exhaustive search*** If the clustering algorithm supports steps (i) and (ii), we call it *exhaustive*. This approach follows a quadratic search and requires $\binom{|\mathcal{C}|}{2}$ additional distance computations. For a given clustering decision, the exhaustive approach outperforms since its search space includes (but is not limited to) the search space of the fast approach. However, the exhaustive approach is not implementable *at line-rate* in existing programmable data planes: it requires accessing multiple times each cluster's information, while registers in today's pipelines can only be accessed once per packet.

Cluster representation (ranges vs. center)

A naive way to represent a cluster is as a mere collection of packets. This is, keeping track of *all* the packets and their respective feature values. Such representation is clearly not practical since it does not scale. Therefore, we study two alternative cluster representations.

*[X] **Center-based representation*** We can represent clusters by just a single point (e.g., the center of the cluster). The advantage of this representation is that the distance computation is simple, and centers can be easily updated

following some pre-defined learning rate [113]. However, we lose a lot of information, such as how big the cluster is or which packets does it contain, which is useful for cluster assessment and traceability (cf. §5.10).

[✓] Range-based representation Following the problem formulation in §5.4.1, *ACC-Turbo* represents each cluster c with a range of values for each ordinal feature $[min_f(c), max_f(c)]$, and a set of discrete unique values for each nominal feature $\{min_f(c), ..., max_f(c)\}$. Ranges (resp. sets) represent the feature values of packets in a cluster (Fig. 5.5). This representation preserves information about the cluster sizes, and simplifies interpretability by providing the exact mapping of packets to clusters. Further, ranges and sets are easy to compute and update, which facilitates its implementation on programmable data planes. For instance, the ranges of a new cluster that merges two existing clusters c_i and c_j for feature f are: $[\min(min_f(c_i), min_f(c_j)), \max(max_f(c_i), max_f(c_j))]$. Sets can be implemented as admission lists, using bloom-filters (cf. §5.6).

The distance function

We first introduce two distance functions for reference, and then derive the distance function that we use in *ACC-Turbo*. For the sake of simplicity, we illustrate the case in which we only consider ordinal features.

[X] Anime distance [108] The cost function in Def.5.1 can be translated to the online-clustering framework as:

$$\delta_{Anime}(\mathcal{C}) = \sum_{c_i \in \mathcal{C}} \delta(c_i) = \sum_{c_i \in \mathcal{C}} \left(\prod_{f \in \mathcal{F}} max_f(c_i) - min_f(c_i) \right) \tag{5.1}$$

This cost function sums the cost of each individual cluster, which accounts for the number of packets that the cluster represents, and estimates the similarity across packets in the cluster. We now derive a distance function that can be used to assess clustering decisions while trying to minimize this cost. Assuming a range-based cluster representation, we define the distance between clusters c_i and c_j, $\delta(c_i, c_j)$, as the amount of increase in cost produced by merging the two clusters, compared to the total cost of the two clusters if they are not merged: $\delta(c_i, c_j) = \delta(c_i \cup c_j) - (\delta(c_i) + \delta(c_j))$.

The main drawback of the Anime distance is the size of its output space. We can measure it as the product of the maximum ranges for each feature. For instance, the maximum cost for the following features and sizes {*ip.len (16b), ip.id (16b), ip.f_offset (13b), ip.ttl (8b), ip.proto (8b), ip.src (32b), ip.dst*

(32b), sport (16b), dport (16b)} is 2^{157}. Data structures in existing data planes can store blocks of maximum 64 bits, which cannot represent such cost.

[X] Euclidean distance Alternatively, we can derive a cost function from the broadly-used Euclidean distance [113]:

$$\delta_{\ Euclid.}(\mathcal{C}) = \sum_{c_i \in \mathcal{C}} \delta'(c_i) = \sum_{c_i \in \mathcal{C}} \left(\sum_{f \in \mathcal{F}} \sum_{p \in c_i} \| p_f - r_f(c_i) \|^2 \right), \tag{5.2}$$

where $r_f(c_i)$ is the representative of cluster c_i for feature f. In this case, the output-space size is much smaller. For the same example, the maximum cost can be represented with less than 20 bits. However, computing the Euclidean distance involves square and root operations, which are not straightforward for existing data planes.

[✓] Manhattan distance In *ACC-Turbo*, we use an alternative distance metric that gathers the benefits of both. Starting from the Anime distance and trying to reduce the size of its output space, we substitute the product of all the individual feature distances by a summation. With this modification, the overall cost function to be minimized can be written as:

$$\delta_{\ Manh.}(\mathcal{C}) = \sum_{c_i \in \mathcal{C}} \delta''(c_i) = \sum_{c_i \in \mathcal{C}} \left(\sum_{f \in \mathcal{F}} max_f(c_i) - min_f(c_i) \right) \tag{5.3}$$

The resulting distance function to compare a new packet p with one of the clusters, c_i, can be written as: $\delta(p, c_i) = \delta(p \cup c_i) - (\delta(p) + \delta(c_i))$. We know that $\delta(p) = 1$ will have a fixed value, so it will not impact the comparison. First, $\delta(p \cup c_i) = \sum_{f \in \mathcal{F}} \delta_f(p \cup c_i)$, where:

$$\delta_f(p \cup c_i) = \begin{cases} max_f(c_i) - p_f, & \text{if } p_f < min_f(c_i), \\ p_f - min_f(c_i), & \text{if } p_f > max_f(c_i), \\ max_f(c_i) - min_f(c_i), & \text{otherwise.} \end{cases} \tag{5.4}$$

Second, we can write $\delta(c_i) = \sum_{f \in \mathcal{F}} \delta_f(c_i)$, where $\delta_f(c_i) = [max_f(c_i) - min_f(c_i)]$. Finally, $\delta(p, c_i) \approx \delta(p \cup c_i) - \delta(c_i) = \sum_{f \in \mathcal{F}} [\delta_f(p \cup c_i) - \delta_f(c_i)] = \sum_{f \in \mathcal{F}} \delta_f(p, c_i)$, where:

$$\delta_f(p, c_i) = \begin{cases} min_f(c_i) - p_f, & \text{if } p_f < min_f(c_i), \\ p_f - max_f(c_i), & \text{if } p_f > max_f(c_i), \\ 0, & \text{otherwise.} \end{cases} \tag{5.5}$$

What results from simplifying the Anime distance into a one-dimensional space is the Manhattan distance from the packet to its closest point of the cluster. For a single dimension (i.e., feature), the Manhattan distance and the Anime distance are equivalent. For higher dimensions, the Manhattan distance compresses the output space. As such, it loses information with respect to the original Anime distance. On the other hand, it is easier to compute. Further, it generates outputs in the linear space (as we have sums instead of products), becoming implementable in existing data planes.

Algorithm 4 ACC-Turbo Clustering Algorithm

Require: p: New packet, $\boldsymbol{min}$, $\boldsymbol{max}$: Initial ranges

procedure CLUSTERING
  for all p: incoming packet **do**
    for all $c_i \in \mathcal{C}$ **do**
      $d(p, c_i) \leftarrow$ COMPUTEDISTANCE(p, c_i)
    end for
  end for
  $c_{selected} \leftarrow c_0$ ▷ Initialize selected cluster
  $d_{min} \leftarrow d(p, c_0)$ ▷ Initialize min. distance
  for all $c_i \in \mathcal{C},\ i \neq 0$ **do**
    if $d(p, c_i) < d_{min}$ **then**
      $d_{min} \leftarrow d(p, c_i)$
      $c_{selected} \leftarrow c_i$ ▷ Select cluster
    end if
  end for
  if $d(p, c_{selected}) > 0$ **then**
    $\boldsymbol{min}, \boldsymbol{max} \leftarrow$ UPDATECLUSTER$(p, c_{selected})$
  end if
end procedure

```
1: function ComputeDistance(p, c_i)
2:     d(p, c_i) ← 0                                   ▷ Initialize distance
3:     for f ∈ F do                                    ▷ Iterate over all features
4:         d_f(p, c_i) ← 0
5:         if f is ordinal then
6:             if p_f < min_f(c_i) then
7:                 d_f(p, c_i) ← min_f(c_i) − p_f
8:             end if
9:             if p_f > max_f(c_i) then
10:                d_f(p, c_i) ← p_f − max_f(c_i)
11:            end if
12:            d(p, c_i) += d_f(p, c_i)                ▷ Aggregate distances
13:        else
14:            if p_f ∉ f(c_i) then
15:                d_f(p, c_i) ← 1
16:            end if
17:        end if
18:    end for
19:    return d(p, c_i)
20: end function
21:
22: function UpdateCluster(p, c_selected)
23:    for f ∈ F do                                    ▷ Iterate over all features
24:        if f is ordinal then                        ▷ Update ranges
25:            if p_f < min_f(c_selected) then
26:                min_f(c_selected) ← p_f
27:            end if
28:            if p_f > max_f(c_selected) then
29:                max_f(c_selected) ← p_f
30:            end if
31:        else                                        ▷ Update feature-value set
32:            if p_f ∉ f(c_selected) then
33:                f(c_selected) = f(c_selected) ∪ p_f
34:            end if
35:        end if
36:    end for
37:    return min, max
38: end function
```

5.5 CONTROLLING AGGREGATES

We now describe how *ACC-Turbo* uses programmable scheduling to mitigate attacks. First, we introduce the programmable-scheduling design space (§5.5.1) and then, *ACC-Turbo*'s scheduler (§5.5.2).

5.5.1 *Programmable-scheduling design space*

Programming a scheduler consists in defining the order in which it should drain packets from a given buffer. This is done by tagging each packet with a *rank* that indicates the priority with which it should be drained.[2] These ranks are then processed by a (programmable) scheduler that dequeues the packets trying to follow the order specified [1, 20, 41].

Ranking algorithms for pulse-wave attacks From our definition in §5.4, a ranking algorithm to mitigate pulse-wave attacks should deprioritize aggregates of high bandwidth and high packet similarity. A number of ranking algorithms can be proposed with this criteria. For example, $rank(p) = th.(c_i)$, $rank(p) = num.packets(c_i)$, and $rank(p) = th.(c_i)/size(c_i)$ deprioritize packets by throughput, packet rate, and a combination between throughput and cluster size, respectively. These algorithms can be easily computed with the available data-plane resources. Aggregate rates can be obtained from packet counters, and packet similarity, from cluster sizes (cf. §5.4).

5.5.2 *Scheduling algorithm*

To use the proposed ranking algorithms for pulse-wave DDoS defense *today*, we need to make them fit into the limited resources of existing data planes. This is hard for three reasons. First, these resources need to be shared with *ACC-Turbo*'s online-clustering module, which is resource exhaustive (e.g., 12 stages for 4 clusters and 4 features (§5.6)). Second, existing programmable switches do not support schedulers able to process ranks, forcing us to "build" our own. While it is possible to approximate this scheduling logic using priority queues [1, 41], doing so requires additional resources (e.g., one stage per priority queue [1]). Third, the clustering and scheduling

2 Generally, a lower rank indicates a higher priority.

modules must operate sequentially since the ranking algorithm requires the clustering results.

Design We build a programmable scheduler on top of priority queues and offload the rank computation and the queue mapping to the control plane. Specifically, the control plane periodically (i) polls information about the extracted clusters from the data plane, (ii) assesses clusters' maliciousness and maps them to a priority queue, and (iii) deploys this mapping into the data plane such that *future packets* of each cluster can be scheduled accordingly. This design dedicates all the data-plane resources to the inference process, maximizing its accuracy and preserving line-rate processing.

5.6 IMPLEMENTATION

We implement *ACC-Turbo* in $P4_{16}$ [28] on Intel Tofino Wedge 100BF-32X [21], with 2484 lines of code. Our prototype uses 12 stages and supports 4 features and 4 clusters. For each incoming packet, *ACC-Turbo* computes its distance to each cluster, selects the closest cluster, and enqueues the packet with the selected cluster's priority.

Cluster selection For each cluster c, we store the minimum and maximum values of its ordinal-feature's ranges $[min_f(c), max_f(c)]$ using registers. We store its nominal-feature's sets using an admission list, implemented as a bloom filter (cf. §5.4).

For each arriving packet, we compute its distance to each cluster by computing all the per-cluster per-feature distances, and aggregating them per cluster. For *ordinal* features, we compute the per-cluster per-feature distances by, first, accessing the register containing $min_f(c)$, and checking if the packet feature, p_f, is below the stored value. If that is the case, we set the distance to $d_f(p, c) = min_f(c) - p_f$ within the register's ALU. If not, we access the register containing $max_f(c)$ and set the distance to $d_f(p, c) = p_f - max_f(c)$ if $p_f > max_f(c)$. Since the two registers are accessed sequentially, this takes two stages. However, computation for different cluster-feature pairs can be parallelized. For *nominal* features, we set the distance to 1 if the bloom-filter entry matched by p_f is empty. This takes one stage.

We aggregate the per-cluster per-feature distances into per-cluster distances by progressively summing two distances at each stage using non-stateful ALUs. This requires $log_2|\mathcal{F}|$ stages, being $|\mathcal{F}|$ the number of features. Finally, we find the minimum distance by progressively comparing

two distances at each stage also using non-stateful ALUs. This requires $log_2|\mathcal{C}|$ stages, being $|\mathcal{C}|$ the number of clusters. The distance-computation and distance-aggregation operations can also be parallelized to reduce the number of stages.

Cluster update When the minimum distance is not zero, we know that the packet has fallen outside of the selected cluster's coverage. In that case, we need to update the cluster's ranges and sets to accommodate the new packet. We do so by using resubmission.

Queue selection We use a match-action table, populated by the controller, to map the packet into a priority queue based on the packet's selected cluster. The controller defines the cluster priorities from cluster statistics polled from the data plane (i.e., register entries and packet counters) following the scheduling policy in §5.5.

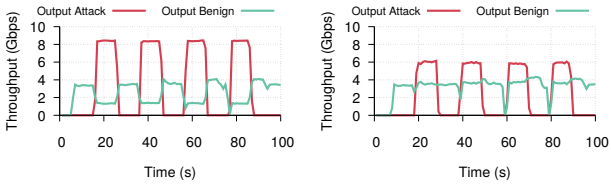
Resource requirements Our implementation (in Tofino 1 [21]) supports a limited number of clusters and features due to Tofino's limited number of stages. Newer programmable switches (e.g., Tofino 2 and 3 [114, 115]) have a higher number of stages, allowing more-performant implementations with more clusters and features.

5.7 HARDWARE-BASED EVALUATION

We evaluate our hardware implementation of *ACC-Turbo* on Tofino. First, we evaluate *ACC-Turbo*'s performance under a pulse-wave DDoS attack (§5.7.1). Second, we compare *ACC-Turbo*'s performance to the one of Jaqen [98], a state-of-the-art DDoS defense (§5.7.2).

5.7.1 *ACC-Turbo's performance*

We generate traffic between two servers, connected by a Tofino switch, using interfaces of 100 Gbps (sender→Tofino) and 10 Gbps (Tofino→receiver). As in previous work [91, 98], we replay CAIDA traces as background traffic [116] and add attack traffic on top using MoonGen [34]. The pulse-wave DDoS attack that we generate is composed of four UDP-flood pulses, which have a duration of 10 seconds each and are followed by a 10-second interleave. Each pulse targets a different IP address within a common subnet and a different port. At its peak, the traffic reaches 40.789 Gbps. We



(a) FIFO (b) *ACC-Turbo*

FIGURE 5.6: Mitigation of a pulse-wave DDoS attack.

configure *ACC-Turbo* to use 4 clusters and to use the last two bytes of the IP destination address, the source port, and the destination port as clustering features. We use a throughput-based ranking algorithm and update the cluster priorities at the controller's maximum speed.

Recovery rate of background traffic Fig. 5.6a shows the traffic throughput evolution under no protection. The attack severely impacts the background traffic, with a throughput reduction of $\approx 61\%$. Note that we are replaying traffic traces and do not see the impact of end-host congestion control. With the effect of congestion control, performance would worsen even further. When we use *ACC-Turbo*, as soon as it infers the attack and deprioritizes its traffic, background traffic fully recovers its original throughput (Fig. 5.6b).

Reaction time Fig. 5.6b illustrates the range of possible reaction times in *ACC-Turbo*. While *ACC-Turbo* reacts to the first pulse almost immediately, it takes up to $\approx$ 1s to react to the last two pulses. This reaction time is the time it takes for the control plane to poll the throughput of each cluster, update their priorities, and deploy them to the data plane. In our testbed, with a non-optimized Python controller, this takes several milliseconds.

5.7.2 *Comparison to the state-of-the-art DDoS-mitigation technique*

We compare *ACC-Turbo* to Jaqen [98], the state-of-the-art DDoS defense. Jaqen uses sketch-based signatures to detect attacks and rate-limiting/dropping to mitigate them (cf. Table 5.3). With respect to Jaqen, we show that *ACC-Turbo* ...

1. ... is more generic (§5.7.2), since it infers attacks agnostically, and not relying on pre-configured signatures.

	Detection	Reaction	Mitigation
ACC-Turbo	Clustering	Always-on	Programmable scheduling
Jaqen [98]	Signature (sketches)	Threshold-based	Rate limit / Drop

TABLE 5.3: *ACC-Turbo*'s techniques vs. Jaqen's.

2. ... achieves faster reaction time (§5.7.2), since it can mitigate attacks without reprogramming the switch.
3. ... does not suffer from the "threshold-based activation" vulnerability (§5.7.2), since it runs continuously on all traffic.

We use the same setup as in the previous section but with the four bytes of the destination IP address as features. We replay CAIDA traces as background traffic at twice their speed (reaching ≈ 7 *Gbps*) during 100 seconds and run attacks on top using MoonGen [34]. We generate attacks at maximum capacity, reaching up to ≈ 99 *Gbps*.

Genericity

We evaluate genericity by analyzing Jaqen's and *ACC-Turbo*'s robustness to attack-traffic variations. Specifically, we consider a UDP-flood attack, which initially consists of a single UDP flow (all the packets share the 5-tuple). We then modify the attack traffic by using: (i) UDP carpet bombing [117–119] (i.e., the attack targets a /24 destination prefix instead of a single IP); and (ii) UDP source spoofing. We configure Jaqen with a sketch that detects heavy hitters either by monitoring the 5-tuple (Jaqen†) or the source IP (Jaqen‡). We measure the percentage of benign packets dropped and show the results in Table 5.4.

We observe that Jaqen is only effective when the right defense is deployed, but its performance drastically decreases as soon as the attack pattern varies. Instead, *ACC-Turbo* performs well generally, being more robust to attack variations. This is expected: while *ACC-Turbo* infers attacks without any initial assumption of the attack characteristics, Jaqen is signature-based and relies on pre-configured defenses for a fixed subset of attacks.

Benign packet drops (%)	FIFO	Jaqen†	Jaqen‡	*ACC-Turbo*
No Attack	0.00	2.50	3.68	0.00
Single Flow	89.85	2.67	3.95	14.79
Carpet Bombing	89.88	73.19	3.95	19.98
Source Spoofing	89.87	88.16	88.51	14.80

TABLE 5.4: Mitigation efficiency under attack variations.

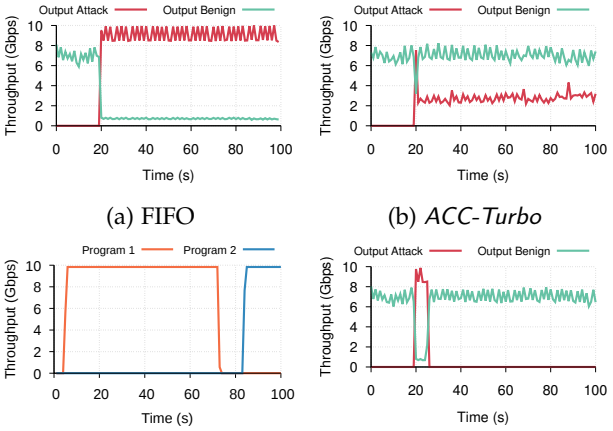
Reaction time

We evaluate *ACC-Turbo*'s and Jaqen's reaction time, defined as the time since they see the first attack packet until they start mitigating the attack. For *ACC-Turbo*, the reaction time is the time it takes for the control plane to poll the cluster statistics, update the cluster priorities, and deploy them to the data plane. For Jaqen, it is the time it needs to: detect the attack, compute the right mitigation, orchestrate the network to reroute legitimate traffic, replicate the switch state to the controller, and reprogram the switch with the right mitigation and replicated state.

We measure *ACC-Turbo*'s and Jaqen's reaction times. For Jaqen, we consider two cases: (i) when it needs to deploy a new mitigation (worst case); and (ii) when the right mitigation is already in the switch (best case). Our results show that *ACC-Turbo* reacts at least 11× (resp. 10×) faster than Jaqen in the first (resp. second) case.

ACC-Turbo's reaction time We generate a simple UDP-flood attack (all packets sharing the 5-tuple) on top of the CAIDA trace (Fig. 5.7a), and measure the time it takes for *ACC-Turbo* to react. As depicted in Fig. 5.7b, *ACC-Turbo* takes $\approx 1s$ to react. This is the time required by the (here, unoptimized) control plane to poll the cluster statistics, and deploy the updated priorities to the data plane.

Jaqen's reaction time (defense not deployed) We measure how fast can Jaqen deploy a mitigation which is not already in the switch. To do so, we measure how long it takes for a hardware switch to swap between two (trivial) programs, which simply rewrite the source IP of all packets. We execute the first program for one minute before instructing the switch to swap to the second program, which has been pre-compiled and cached. We send traffic continuously and measure its downtime. We repeat the



(a) FIFO (b) *ACC-Turbo*

(c) Reprogramming time (d) Jaqen

FIGURE 5.7: Reaction-time evaluation.

experiment 10 times. On average, it takes 11.5 seconds for Jaqen to deploy a new mitigation. This is 11× slower than *ACC-Turbo*'s reaction time. Fig. 5.7c shows the result for one of the iterations.

Jaqen's reaction time (defense already deployed) If the mitigation module is already loaded, Jaqen's reaction time is the time to detect the attack from the control plane and to deploy a dropping rule to the data plane. We measure this reaction time when Jaqen is configured with the simplest defense (to ensure it is as fast as possible): a sketch that monitors the number of attack packets and a controller that reads it periodically, activating a dropping action when an attack is detected. We read the sketch entries at maximum speed and optimize the threshold value for performance.

As shown in Fig. 5.7d, Jaqen's reaction time is ≈ 10 seconds. This is *still* 10× slower than *ACC-Turbo*. This is the time required by the controller to read the sketch and deploy the drop action and the time it takes for the threshold to be reached–not once, but twice–since Jaqen's only considers attacks when detected in two consecutive time windows.

Jaqen's slow reaction time, of ≈ 10 seconds, even in the ideal case, makes it vulnerable to pulse-wave DDoS attacks.

Threshold-configuration sensitivity

We analyze how sensitive Jaqen is to the threshold-based defense activation vulnerability introduced in §5.2.2. To that end, we take Jaqen's simplest possible defense: the 5-tuple heavy hitter in §5.7.2 (Jaqen†). Such defense relies on two parameters: the *threshold* over which traffic is considered to be an attack and the *periodicity* at which this threshold is checked. We analyze the mitigation efficiency when the two parameters are modified, in the case of a simple UDP-flood attack on top of a CAIDA trace. We measure the percentage of benign traffic dropped and compare the results to the case in which no-defense (i.e., FIFO), and *ACC-Turbo*, are used.

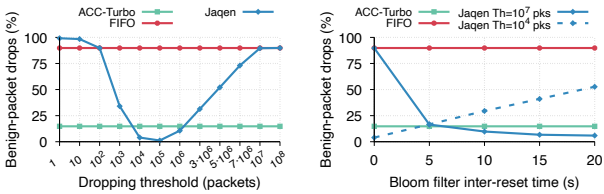
Fig. 5.8a illustrates Jaqen's high sensitivity to threshold configuration. Slight variations in the threshold value (e.g., from $1M$ packets to $7M$ packets) can result in a drastic increase of benign-packet drops (from $\approx 10\%$ to $\approx 75\%$). This is expected and aligned with our arguments in §5.2.2. Indeed, the best threshold value depends on the dynamics of both attack and benign traffic, which continuously change over time. As also expected, too-low thresholds can result even worse than not having any defense. This is because they may drop benign traffic even in case of no congestion.

Fig. 5.8b shows how the periodicity at which the threshold is checked can further impact a certain threshold's performance. For example, a threshold of 10^4 *packets*, which outperforms at the controller's maximum periodicity, performs very poorly if the periodicity decreases. At the same time, a bad-performing threshold at maximum periodicity (e.g., 10^7 *packets*) can perform well if the controller's periodicity decreases.

ACC-Turbo avoids the threshold-based vulnerability by running continuously on all traffic. Further, *ACC-Turbo* does not perform *binary* assessments on whether a traffic aggregate is malicious or not based on its absolute traffic rate. Instead, it performs *finer-grained* assessments that deprioritize traffic aggregates based on their relative rates (with respect to other aggregates) and their cluster statistics. As a result, even though *ACC-Turbo*'s performance is not as good as Jaqen's when Jaqen is configured optimally, it manages to outperform when Jaqen is not perfectly tuned.

5.8 SIMULATION-BASED EVALUATION

Given the limitations of our Tofino prototype (§5.6), we extend the evaluation of *ACC-Turbo* by using packet-level simulations in Netbench [26,



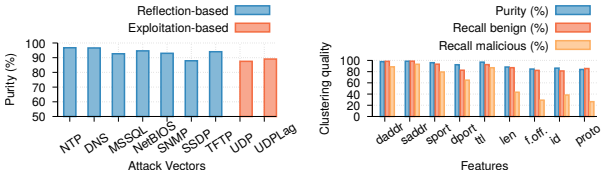
(a) Threshold setting (b) Speed

FIGURE 5.8: Threshold-configuration sensitivity.

27]. We evaluate the performance of *ACC-Turbo* when it schedules a mix of benign traffic and DDoS attacks. We analyze the impact of its different design decisions (§5.4) and study the performance of more-complete implementations of *ACC-Turbo* which are not implementable in Tofino1 but we expect to be implementable in the near future (e.g., by using Tofino2 or Tofino3 [114, 115]). We first characterize the clustering strategy (§5.8.1) and then, the scheduling scheme (§5.8.2).

Methodology We simulate an *ACC-Turbo*-enabled switch that processes one day of traffic in which a series of DDoS attacks (or a single morphing attack) is received. We do so by feeding the CICDDoS-2019 trace [120], which contains a sequence of DDoS attacks, into a simulated switch running *ACC-Turbo*. By default, we configure *ACC-Turbo* with 10 clusters, and use each byte of the *ip.src* and *ip.dst*, *sport*, *dport*, *ip.ttl*, and *ip.len* as features. We adjust the capacity of the output link to various congestion levels.

Overall performance Even though *ACC-Turbo*'s performance depends on the characteristics of benign and attack traffic, we now illustrate its practicality by evaluating it on a realistic dataset covering a wide range of attack vectors. For all attacks, *ACC-Turbo*'s online-clustering algorithm manages to distinguish packets from attack and benign distributions (achieving cluster's purity of $\approx$ 90%), with as few as 10 clusters. *ACC-Turbo*'s performance is better for well-defined traffic aggregates. For reflection attacks with high packet similarity, *ACC-Turbo* manages to save up to 29% more of benign traffic than FIFO queues, just 5.13% away from an ideal scheduler with full attack knowledge (cf. Fig. 5.11b, at 50Mbps).



(a) Attack split (b) Feature split

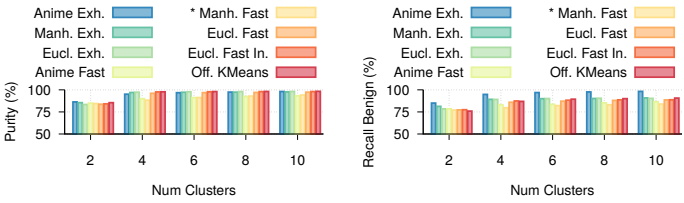
FIGURE 5.9: Performance by attack type and features.

5.8.1 *Characterizing the clustering strategy*

We evaluate *ACC-Turbo*'s inference by measuring the *purity* and *recall* of the extracted clusters. These metrics measure the accuracy of the clustering algorithm in mapping packets from different distributions into distinct clusters. We compute purity by (i) labeling each cluster as either majority-benign or majority-malicious, based on the number of clustered packets of each type, (ii) counting the number of packets that match the cluster's label, and (iii) dividing it by the total number of packets [121]. We compute *recall* of benign (resp. malicious) packets as the percentage of benign (resp. malicious) packets mapped into majority-benign (resp. -malicious) clusters. We compute the metrics every one minute and average the result. We only count periods with both attack and benign traffic.

Feature selection and attack vectors Fig. 5.9a shows the clustering performance across attack vectors. In all cases, purity is above 87%. The clustering performance is strongly related to the variance of the attack features. For example, reflection-based attacks achieve, on average, 5.4% better purity than exploitation-based attacks. Within reflection-based attacks, MSSQL and SSDP, which have higher feature-value variance, perform worst (e.g., MSSQL uses multiple source ports, while NTP or DNS use a single port).

Fig. 5.9b shows the performance of clustering on individual features. For this particular dataset, IP addresses and source port are good identifiers of malicious traffic. In contrast, fields like IP protocol are less useful as attacks use both UDP and TCP. While the absolute values are tied to the trace characteristics, the split illustrates the different performance levels that *ACC-Turbo* can achieve. While "narrow" attacks achieve good performance if we look at the right features, their performance drastically drops as soon as attacks become diverse or features do not provide a clear signature.



(a) Purity (%) (b) Recall benign (%)

FIGURE 5.10: Performance of clustering strategies.

Number of clusters Fig. 5.10 shows the performance of different clustering strategies when the number of clusters varies from 2 to 10. First, as expected (cf. §5.4), a higher number of clusters provides better purity and recall.[3] While selecting the optimal number of clusters is typically a challenge, in *ACC-Turbo* it is imposed by the hardware constraints (§5.6). Second, increasing the number of clusters is more beneficial for fewer clusters (e.g., the purity in *ACC-Turbo* improves by 4% when we move from 2 clusters to 4, while it only improves 1% when we move from 8 to 10). Since *ACC-Turbo* is designed to run in an environment where the number of clusters is limited, it builds upon this insight and dedicates all data plane resources to maximize the number of clusters (running the non-inference operations in the control plane).

Clustering search: fast vs. exhaustive As expected (cf. §5.4.2), we observe that *exhaustive* approaches generally outperform their *fast* versions, even though the difference gets smaller as the number of clusters increases. This is especially clear in the Anime and Manhattan approaches, which use range-based representations and have more information to assess clustering decisions (e.g., 93.24% to 98.09% purity increase for Anime with 10 clusters).

Cluster representations and distances Overall, center-based distances "suffer" less when downgraded from exhaustive to fast (e.g., just 0.79% purity decrease with 10 clusters). There are two reasons for that. First, since they carry little information about clusters, the potential improvement of checking more combinations is limited. Second, range-based approaches are

3 A naive way to achieve perfect purity is to have as many clusters as packets, mapping each packet to its own cluster. From a scheduling perspective, however, this approach has no value since clusters do not give any information about the maliciousness of the packets contained.

more sensitive to updates since they directly include the new points as they are analyzed. In center-based approaches, we just move the center in the direction of the new data point using some learning rate.

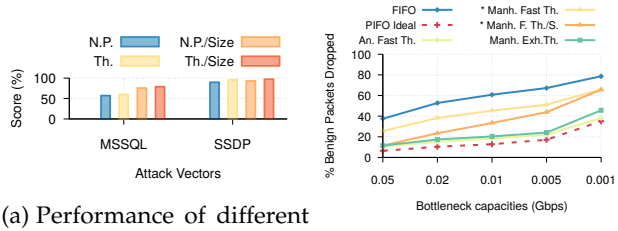
Fast vs. offline and baselines We compare the performance of *ACC-Turbo* to the one of *offline* k-means with unlimited resources. In all cases, *ACC-Turbo* is very close to the *offline* case (e.g., 4.19% of difference in purity for 10 clusters, cf. Fig. 5.10). We also study the performance of a hybrid approach, which periodically computes the cluster centers offline and updates them online with the new packets. While the hybrid approach outperforms, the improvement is not substantial enough to justify its added complexity.

5.8.2 *Characterizing the scheduling scheme*

Finally, we study *ACC-Turbo*'s scheduling performance. First, we evaluate the schedulers in §5.5, analyzing how often they prioritize benign traffic over malicious traffic. We measure a *score*, defined as the percentage of one-second intervals in the simulation where the average priority given to benign traffic is higher than the one given to malicious traffic. Second, we measure the number of benign packets dropped when the trace is scheduled by a FIFO queue, *ACC-Turbo*, and an ideal scheduler, which prioritizes benign traffic. We use 10 clusters and the 10 most representative features for the trace.

Ranking algorithms Fig. 5.11a shows the performance of the scheduling algorithms under the two most complex reflection attacks (cf. Fig.5.9a). While the absolute values are specific to the dataset, we see how adding the similarity factor to the rank definition (i.e., the cluster sizes) improves performance. This result strongly supports *ACC-Turbo*'s design decision of using a range-based representation.

Bottleneck Fig. 5.11b analyzes the number of dropped packets for various bottleneck capacities. First, we see that the *ACC-Turbo* version that is implementable today (i.e., Manhattan distance, fast approach) performs almost on par with the ideal case for smaller bottlenecks. Indeed, at 50 Mbps it saves 29% more benign traffic than FIFO queues, being just 5.13% worse than an ideal PIFO with the ground truth. Second, we compare its performance to the one of Manhattan-exhaustive, and Anime-fast approaches. We see how the loss in performance of today's *ACC-Turbo* comes from the two



(a) Performance of different ranking algorithms

(b) Overall performance

FIGURE 5.11: Impact of scheduling for mitigation.

design decisions required to make it fit into existing devices: using a fast approach and reducing the size of the distance space (§5.4). With newer programmable switches, some of which are available [114, 115], more-complete versions of *ACC-Turbo* become implementable.

5.9 LIMITATIONS

In this section, we discuss techniques by which attackers could try to evade *ACC-Turbo* (§5.9.1), and mechanisms by which attackers could try to weaponize *ACC-Turbo* to block benign traffic (§5.9.2).

5.9.1 *Evading ACC-Turbo*

ACC-Turbo infers attack traffic in an unsupervised fashion, looking for *unexpectedly-high rates* of *traffic aggregates* (i.e., packets that share some similarity). Based on this premise, an attacker willing to evade *ACC-Turbo* has potentially two options: (i) trying to generate low-rate attack traffic and/or (ii) trying to diversify attack packets.

Pulse-wave DDoS attacks are volumetric by nature (i.e., they need *high* traffic rates in order to congest the target link). As such, the first option *per se* is not actually viable. This only leaves the attacker one option: breaking the packet similarity. We now discuss two granularities at which the attacker can break packet similarity: at the packet level and at the aggregate level.

Breaking packet similarity at packet level The goal of this approach is to make it harder for *ACC-Turbo* to correlate attack packets, by adding

randomness to one or multiple packet features. Various techniques serve this purpose: e.g., carpet bombing or IP spoofing [117–119]. These techniques spread the feature values of attack packets all over the space, making *ACC-Turbo* unable to identify any relation among them. While *ACC-Turbo* is robust to a certain degree of randomness (cf. §5.7), it can not infer attack traffic if all the clustering features are randomized. In the worst scenario, this approach can end up with attack packets mapped to all the clusters and with *ACC-Turbo* not being able to mitigate the attack.

Since these techniques are widely used, we propose two solutions to prevent them. First, network operators can pick clustering features strategically to avoid these behaviors. For example, we can prevent carpet bombing by clustering longer IP prefixes rather than individual destination IPs [119]. Second, operators can leverage network-monitoring tools to proactively detect and mitigate these patterns, e.g., directly in the data plane or as a part of the cluster's assessment of *ACC-Turbo* in the control plane.

Breaking packet similarity at aggregate level This approach consists in generating multiple low-rate attack aggregates formed by different traffic types, such that *ACC-Turbo* can not identify any relation between them. In the worst case, this attack can result in *ACC-Turbo* deprioritizing traffic clusters that do not accurately represent attack traffic and not being able to mitigate the attack.

The simplest instantiation of this attack (in number of attack vectors required) is the one composed of $|\mathcal{C}|$ spread-out attack vectors, where each vector targets a different cluster. The goal is to attack all the clusters simultaneously, such that *ACC-Turbo* becomes ineffective. While possible in theory, this attack is challenging to execute in practice. Indeed, it requires the attacker to: (i) infer the clustering features and the ranking policy of the victim *ACC-Turbo* instance, (ii) find out which are the attack vectors that maximize their respective distance in the feature space and minimize their probability to be deprioritized by the ranking policy, and (iii) generate these attack vectors, while making sure that they can still reach the victim target. Even though it may be possible to execute this attack for some scenarios, it requires higher complexity than the needed for a conventional DDoS attack. Further, this attack becomes harder to execute, and decreases its effectiveness, linearly to the number of clusters. This is encouraging given the higher number of resources available in newer generations of programmable switches (§5.7).

5.9.2 *Weaponizing ACC-Turbo*

We now discuss two attacks which are enabled by *ACC-Turbo*. Here, the goal of the attacker is not to evade *ACC-Turbo*, but to trick it into treating some portion of benign traffic as malicious traffic.

Swapping attack This attack aims at causing *most* benign (resp. malicious) traffic to be treated as malicious (resp. benign). For that to happen, benign traffic needs to already have a high packet similarity and a high rate. In that case, an attacker can generate lower-rate traffic with randomized packet headers to congest the target link while evading *ACC-Turbo*'s inference. In the worst case, this attack can result in malicious traffic being prioritized over benign traffic. This attack can be perceived as a special case of packet-level evasion (§5.9.1) and can be defended using the same techniques.

Imitation attack This attack involves generating traffic that closely resembles production traffic, with the goal of causing *ACC-Turbo* to deprioritize both the attack and the victim's traffic. Similar to the previous attack, this approach is challenging to execute in practice. Attackers must predict the appearance of the victim's traffic at the time of the attack and accurately replicate it at a high rate, at least for the features supported by *ACC-Turbo*.

The problem of imitation attacks (or attacks composed of indistinguishable flows) has been studied by previous literature in the context of link-flooding attacks [99, 107]. Solutions involve rate-change tests or historical-pattern analysis to shed light on whether a certain aggregate is a legitimate service or a spoofed replica [107].

5.10 DISCUSSION

Are today's DDoS still "aggregates"? Yes. Even though DDoS attacks can *theoretically* be arbitrarily complex, most DDoS attacks today are still formed by well-defined traffic subsets [119, 122], thus being characterizable as "aggregates" [106].

For instance, the Mirai attack, which is probably the most famous botnet-based attack to date, included several flooding attacks such as UDP flood, SYN flood, ACK flood, or HTTP flood [123]. Each such attack generated highly-similar packets [124]: for instance, all SYN-flood packets had the same size, protocol, flags, and shared the same source IP subnets.

Amplification attacks, such as the NTP-based attack on Cloudflare (2014), DNS-based attack on Google (2017), or the Memcached-based attack against GitHub (2018), can be generally characterized by unusually-large packets sourcing from a specific port, using the same protocol, and originating from a common subset of IPs [125, 126].

Finally, alleged link-flooding attacks [127, 128] (cf. [107]) were also composed by (i) "DNS responses of 3000 bytes and TCP reflections targeting specific addresses of the victim IXP" [127], and (ii) "amplification vectors such as NTP, UDP, TCP, and ICMP-floods" [128].

Can ACC-Turbo cover other attack types? While *ACC-Turbo* has been designed to cover pulse-wave DDoS attacks, it can also mitigate conventional DDoS attacks as long as they are volumetric and composed of clear traffic aggregates (as we show in §5.8). On the other hand, *ACC-Turbo* does *not* cover application-layer attacks nor low-bandwidth attacks. Since these attacks are not volumetric, they do not generate congestion in the network and remain unnoticed by *ACC-Turbo*. However, as we discuss in §5.9, *ACC-Turbo* can be deployed together with complementary defenses, which may protect against attacks that *ACC-Turbo* can not cover.

What is the impact of leaving ACC-Turbo always-on? Under sporadic congestion, the possible packet reorderings produced by *ACC-Turbo* could be happening without our solution as well, e.g., if the deprioritized packets suffered from longer delays on the network. In case of sustained congestion (not an attack), *ACC-Turbo* will deprioritize groups of packets with higher rates (heavy hitters) and give more priority to less aggressive groups of packets. The impact could be similar to a fair-queuing scheme, with the difference that the definition of a flow is inferred dynamically.

What about reordering? Assuming that the features used are common across all packets of a given flow, all these packets will be mapped into the same cluster. As such, reordering can only happen when the priority given to the flow's cluster increases over time while there are still packets of the same flow in the queue with the old (lower) priority. Since we update priorities in time windows of milliseconds to seconds, potential reordering would only impact large flows with already-high flow completion times.

What about interpretability? *ACC-Turbo*'s range-based clustering allows operators to know which packets are being mapped into each cluster, as well as the exact mapping of these clusters to the priority queues. Contrary

to a black-box approach, an operator can access the complete information of every action performed in real-time. An operator could further modify the table entries in *ACC-Turbo* to reduce the number of priority queues to be used to drop specific parts of traffic or treat some known-benign traffic preferentially (e.g., by mapping it to a dedicated priority queue).

What future research do we envision? Given the limitations of *ACC-Turbo*, we expect future work to tackle two main research directions. First, the design of enhanced inference techniques, which can identify attack patterns with higher accuracy and robustness. Second, the combination of signature-based defenses, which outperform in detecting known attacks (§5.8), with generic defenses like *ACC-Turbo*, which offer the potential to infer new attacks. The combination of both approaches can also result beneficial in preventing adversarial attacks to unsupervised DDoS defenses (cf. §5.9).

5.11 RELATED WORK

Given the novelty of pulse-wave DDoS attacks, only a few works have studied their mitigation [129]. Therefore, in this section, we cover some in-network defenses for conventional DDoS attacks, which are also related to our work. Another group of attacks, called *shrew* or *pulsing attacks*, also leverage traffic pulses to disrupt the victim's connectivity [130]. However, they differ from pulse-wave DDoS attacks in that they are low-rate and target TCP vulnerabilities.

Conventional-DDoS defenses Poseidon [91], Jaqen [98], and Ripple [99] use programmable switches to mitigate DDoS attacks. Poseidon proposes a system-level solution to orchestrate traffic to pre-defined defenses in dedicated servers. Jaqen and Ripple are both signature-based and rely on a network-wide view of attack signals to deploy pre-configured defenses. *ACC-Turbo* runs autonomously in a switch (not requiring network orchestration nor switch reprogramming), achieves fast reaction times, and covers unknown attacks. Bohatei [90] uses network function virtualization to adapt the placement, scale, and the type of pre-defined defenses depending on the detected attacks. SPIFFY [107] detects link-flooding attacks by actively modifying the available bandwidth and analyzing the traffic reaction. Kitsune [104] trains an ensemble of autoencoders to identify anomalies in network traffic. Euclid [105] uses statistical analysis to detect attacks from the network. While these solutions achieve high accuracy, they execute

drastic mitigation policies such as filtering. In contrast, *ACC-Turbo* uses programmable scheduling to mitigate attacks directly from the network which is safer and reduces collateral damage in case of misclassification.

Scheduling-based DDoS defenses DDoS-Shield [131] uses scheduling to mitigate DDoS attacks. While *ACC-Turbo* faces pulse-wave DDoS attacks from the network, DDoS-Shield targets application-layer attacks from the end-host. In [132], a DDoS defense is proposed, using two priority queues and assigning suspected flows to the low-priority queue. Suspected flows are identified based on the difference in harmonic means of the arrival rate of incoming packets. *ACC-Turbo* uses more priority queues to enable finer-grained decisions and accommodates more complex DDoS attacks.

5.12 CONCLUSIONS

In this chapter, we emphasized the need for in-network pulse-wave DDoS defenses, and showcased the potential of building them by combining unsupervised inference and programmable scheduling. We presented *ACC-Turbo*, the *first* aggregate-based congestion control mechanism that mitigates pulse-wave DDoS attacks by running at line rate on commodity hardware. *ACC-Turbo* leverages two key insights: online clustering—to effectively identify (possibly unknown) attack pulses, and programmable scheduling—to safely deprioritize traffic according to its maliciousness. We implemented *ACC-Turbo* in P4 and deployed it on programmable hardware. *ACC-Turbo* can mitigate pulse-wave DDoS attacks in almost real-time.

6

CONCLUSION AND OUTLOOK

In this dissertation, we have developed four systems to facilitate in-network congestion management on existing devices, showcasing its potential to significantly enhance the performance and security of the modern Internet. Firstly, we introduced SP-PIFO and PACKS, two frameworks designed to adapt the PIFO abstraction to the limitations of current programmable data planes, enabling flexible in-network congestion management on existing hardware. Secondly, we presented QVISOR, a scheduling hypervisor aimed at extending in-network congestion management capabilities to multi-tenant environments like data centers and cloud networks. QVISOR allows tenants to program their own congestion management algorithms while sharing a set of underlying hardware, ensuring fair and effective resource allocation. Lastly, we introduced ACC-Turbo, which showcases the power of in-network congestion management in the context of security by effectively countering the most sophisticated forms of denial-of-service attacks. These systems demonstrate the advantages of in-network congestion management and their potential to shape the future of Internet performance and security.

In Chapter §2, we introduced SP-PIFO, an implementation of in-network congestion management on existing programmable data planes. SP-PIFO relies on a per-packet adaptation heuristic that approximates the behavior of a PIFO queue using widely-available priority queues without requiring prior traffic knowledge. It dynamically adjusts the mapping between incoming packets and priority queues to minimize the scheduling difference with respect to the ideal PIFO queue. We showcased SP-PIFO's practicality with as few as 8 priority queues, its quick reaction to traffic variations, and its ability to run at line rate on existing hardware, on an Intel Tofino device.

In Chapter §3, we identified a limitation in SP-PIFO's design: it only approximates PIFO's scheduling behavior, neglecting to simultaneously approximate its admission control. To address this, we introduced PACKS, an approximate PIFO scheduler that outperforms SP-PIFO by emulating PIFO queues in both scheduling and admission. Unlike SP-PIFO, PACKS incorporates an admission-control mechanism alongside its queue-mapping technique. It leverages prior knowledge of the traffic distribution and queue

occupancy levels to determine which packets to admit and how to map them to priority queues. We demonstrated that PACKS approximates PIFO more accurately than SP-PIFO, reducing scheduling inversions by up to 7× and the number of packet drops by up to 60%. Both SP-PIFO and PACKS showcase the feasibility of deploying in-network congestion management on existing infrastructure, allowing operators to start benefiting from it.

In Chapter §4, we addressed a limitation of both SP-PIFO and PACKS: their inability to implement multiple in-network congestion management policies concurrently. We highlighted the necessity of supporting multi-tenancy in programmable in-network congestion management, given the heterogeneous nature of modern network infrastructures like data centers and cloud environments. To tackle this challenge, we introduced QVISOR, a framework designed to empower multiple tenants to apply their own congestion management policies to their specific traffic subsets. Simultaneously, QVISOR allows operators to define how hardware resources should be allocated among tenants. The core idea behind QVISOR is to serve as a hypervisor, simplifying the management of the congestion-management resources across multiple tenants with different objectives and requirements.

In Chapter §5, we focused on security, demonstrating the advantages of in-network congestion management in mitigating pulse-wave denial-of-service attacks. These attacks can evade existing defenses by exploiting their slow convergence time. By running directly in the network, in-network congestion management techniques can identify attack patterns in less than a second and effectively mitigate them. We introduced ACC-Turbo, an in-network defense mechanism that leverages online clustering to identify sections of malicious traffic at line rate, even if they use zero-day vectors, and programmable scheduling to safely deprioritize malicious traffic, thereby mitigating the attacks. We showcased how ACC-Turbo effectively mitigates sophisticated pulse-wave DDoS attacks in real-time, rapidly adapting to attack variations, and being ten times faster than the state of the art.

6.1 FUTURE WORK

We see opportunities for future work in a number of areas of in-network congestion management. In the following, we describe a subset of them.

6.1.1 *Introducing buffer-management techniques to the equation*

In Chapter 2 and Chapter 3, we built two abstractions aimed at enabling in-network congestion management on existing devices by approximating the behaviors of a PIFO queue. SP-PIFO approximated PIFO's scheduling behavior and PACKS incorporated its admission behavior. In both SP-PIFO's and PACKS's design, we considered a complete partitioning scheme where each priority queue is statically provisioned with a fixed buffer size. We see research opportunities in studying the potential benefits of other buffer management techniques, such as dynamic thresholds or complete sharing, some of which are supported on existing devices. A better understanding of these techniques and their interaction with scheduling and admission control can not only open doors to closer-to-PIFO implementations but also help us better design in-network congestion management techniques.

6.1.2 *Studying the benefits of rotating priority queues*

We designed SP-PIFO and PACKS to operate on a set of strict priority queues, each with a fixed priority. Recent works have introduced alternative designs that rely on rotating priority queues [33, 42]. These rotating priority queues offer increased flexibility but require more complex configuration. Although most programmable schedulers currently lack support for rotation of queue priorities at line rate, we advocate for a competitive analysis between the two schemes to study in depth their respective advantages.

6.1.3 *Automating window-size configuration*

Similar to other networking algorithms, PACKS relies on a window-based approach and therefore requires two configuration parameters: the burstiness allowance and, most importantly, the window size. In our evaluation, we have explored the performance impact of configuring both large and small window sizes (see Fig. 3.19) and have observed that longer windows outperform under stationary distributions, while shorter ones work better for less stable scenarios. Consequently, we see opportunities for research in designing techniques to automatically adjust window size configurations based on the monitored traffic conditions. Such an approach would enhance PACKS's adaptability and eliminate its need for manual configuration.

6.1.4 *Designing new abstractions for in-network congestion management*

In this thesis, we have relied on the PIFO abstraction as the primary driver of in-network congestion management. While PIFO queues can express a wide range of policies, they do come with limitations. For instance, ranking algorithms only allow operators to determine the importance of each packet, and not whether it is preferable to drop or reorder it. Moreover, these algorithms cannot express packet ranks as a function of the delivery or non-delivery of other packets. Although recent works have already proposed new abstractions to address some of PIFO's limitations (e.g. its inability to represent non-work-conserving schedulers), we expect future research to introduce new abstractions for in-network congestion management.

6.1.5 *Enabling multi-tenant scheduling on existing devices*

In Chapter 4, we introduced the vision for QVISOR but did not present a complete design or implementation. We expect follow-up work in this area to address several key questions, such as how to best manage the interaction between tenants, how to normalize different rank policies at scale and at runtime, what is the right definition of isolation in this context, and what is the appropriate granularity for ensuring fairness between tenants. We believe that hierarchical PIFO trees [20] offer the right abstraction to express the requirements of multiple tenants. However, realizing them at line rate on existing programmable data planes remains an open challenge.

6.1.6 *Improving the performance of ACC-Turbo*

In Chapter 5, we introduced ACC-Turbo, a dedicated in-network congestion management technique tailored to mitigating denial-of-service attacks. We see research opportunities in improving the performance of the proposed design across two dimensions. First, by leveraging smarter data structures that can better discern attack traffic from benign traffic (e.g., using sketches or probabilistic data structures). Second, by integrating data from other sources to the ranking function beyond observed traffic (e.g., whitelists, blacklists, or IP spoofing reports based on BGP routing information). In this direction, we also expect future research to explore the combination of unsupervised techniques, like ACC-Turbo, with supervised methods.

6.1.7 *Extending in-network congestion-management to other domains*

We believe that several other applications, which are sensitive to network congestion, could greatly benefit from in-network congestion management. For instance, we see research opportunities in designing congestion management techniques for video delivery over content delivery networks or distributed training within machine learning clusters. Likewise, we also see potential in applying in-network congestion management to other network domains with constrained bandwidth resources, such as cellular networks.

6.1.8 *Synthesizing congestion management policies from high-level intents*

So far, we assumed that operators are in charge of manually programming their in-network congestion management techniques based on their goals. As we continue to design novel policies and gain a better understanding of their intricacies, we anticipate the emergence of mechanisms that can automatically generalize this knowledge and create new algorithms tailored to arbitrary performance objectives. These mechanisms could synthesize the optimal policy for a given workload and performance goal, simplifying the task for operators, who would only need to provide a high-level description of their performance intent instead of a low-level ranking function.

6.1.9 *Combining in-network congestion management with end-to-end approaches*

In-network congestion management is just one piece of the puzzle when it comes to managing congestion on the Internet. We see research opportunities in co-designing in-network techniques with end-host control mechanisms, especially beneficial in private networks like data centers where both end hosts and network devices are controlled by the same operator. We envision "rank-aware congestion control" schemes, where in-network abstractions (e.g., PIFO, PACKS, or SP-PIFO) provide feedback to end hosts (such as the rank distribution of dropped packets), enabling them to adjust transmissions for improved performance. Works like pFabric [31] and Homa [133] have demonstrated the benefits of combining these techniques for minimizing flow completion times. This design space offers opportunities across all levels: from dedicated solutions to optimize specific performance goals to higher-level abstractions to facilitate such co-designs.

OWN PUBLICATIONS

[1] Albert Gran Alcoz, Alexander Dietmüller, and Laurent Vanbever. "SP-PIFO: Approximating Push-In First-Out Behaviors using Strict-Priority Queues". In: *USENIX NSDI*. 2023.

[2] Albert Gran Alcoz, Martin Strohmeier, Vincent Lenders, and Laurent Vanbever. "Aggregate-Based Congestion Control for Pulse-Wave DDoS Defense". In: *ACM SIGCOMM*. 2022.

[3] Albert Gran Alcoz and Laurent Vanbever. "QVISOR: Virtualizing Packet Scheduling Policies". In: *ACM HotNets*. 2023.

[4] Albert Gran Alcoz, Balázs Vass, Pooria Namyar, Behnaz Arzani, Gábor Rétvári, and Laurent Vanbever. "Everything Matters in Programmable Packet Scheduling". In: *USENIX NSDI*. 2025.

[5] Lloyd Brown, Albert Gran Alcoz, Akshay Narayan, Mohammad Alizadeh, Hari Balakrishnan, Eric Friedman, Ethan K.-Bassett, Arvind Krishnamurthy, Michael Schapira, and Scott Shenker. "Principles for Internet Congestion Management". In: *ACM SIGCOMM*. 2024.

[6] Albert Gran Alcoz, Coralie Busse-Grawitz, Eric Marty, and Laurent Vanbever. "Reducing P4 Language's Voluminosity using Higher-Level Constructs". In: *ACM EuroP4*. 2022.

[7] Alexander Dietmüller, Albert Gran Alcoz, and Laurent Vanbever. "FitNets: An Adaptive Framework to Learn Accurate Traffic Distributions". In: *arXiv preprint, arXiv:2405.10931*. 2024.

[8] Sarah McClure, Albert Gran Alcoz, Laurent Vanbever, Sylvia Ratnasamy, and Scott Shenker. "Inter-Cloud QoS with FlexEgress". In: *Under submission*. 2024.

[9] Vasileios Giotsas, Albert Gran Alcoz, Lucas Castanheira, Theophilus Benson, Georgios Smaragdakis, and Marwan Fayed. "Spoof-Shield: Mitigating IP Spoofing Attacks at the Internet's Edge". In: *Under submission*. 2024.

[10] J. Noel Chiappa. *ARPANET Technical Information: Geographic Maps*. 2012.

[11] Vinton G. Cerf and Robert E. Kahn. "A Protocol for Packet Network Intercommunication". In: *IEEE ToC*. 1974.

[12] John Nagle. "Congestion Control in IP/TCP Internetworks". In: *Request For Comments: 896*. 1984.

[13] V. Jacobson. "Congestion Avoidance and Control". In: *ACM SIGCOMM*. Stanford, California, USA, 1988.

[14] James Manyika and Charles Roxburgh. "The Great Transformer: The Impact of the Internet on Economic Growth and Prosperity". In: *McKinsey Global Institute*. 2011.

[15] Danny Palmer. *This Massive DDoS Attack Took Large Sections of a Country's Internet Offline*. 2021.

[16] *Network Programmability: The Road Ahead.* 2023.

[17] Radhika Mittal, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. "Universal Packet Scheduling". In: *USENIX NSDI*. Santa Clara, CA, USA, 2016.

[18] Anirudh Sivaraman, Keith Winstein, Suvinay Subramanian, and Hari Balakrishnan. "No Silver Bullet: Extending SDN to the Data Plane". In: *ACM HotNets*. College Park, MD, USA, 2013.

[19] Anirudh Sivaraman, Suvinay Subramanian, Anurag Agrawal, Sharad Chole, Shang-Tse Chuang, Tom Edsall, Mohammad Alizadeh, Sachin Katti, Nick McKeown, and Hari Balakrishnan. "Towards Programmable Packet Scheduling". In: *ACM HotNets*. Philadelphia, PA, USA, 2015.

[20] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, S.T. Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. "Programmable Packet Scheduling at Line Rate". In: *ACM SIGCOMM*. Florianópolis, Brazil, 2016.

[21] Intel. *Intel Tofino Programmable Chipset Series*. 2021.

[22] *Broadcom Trident II*. 2016.

[23] Alexander Barkalov, Larysa Titarenko, and Malgorzata Mazurkiewicz. *Foundations of Embedded Systems*. Springer International, 2019.

[24] Chuck Semeria. "Supporting Differentiated Service Classes: Queue Scheduling Disciplines". In: *Juniper Networks White Paper*. Sunnyvale, CA, USA, 2001.

[25] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. "NetCache: Balancing Key-Value Stores with Fast In-Network Caching". In: *ACM SOSP*. Shanghai, China, 2017.

[26] *Netbench*. 2018.

[27] Simon Kassing, Asaf Valadarsky, Gal Shahaf, Michael Schapira, and Ankit Singla. "Beyond Fat-trees Without Antennae, Mirrors, and Disco-balls". In: *ACM SIGCOMM*. Los Angeles, CA, USA, 2017.

[28] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, et al. "P4: Programming Protocol-Independent Packet Processors". In: *ACM SIGCOMM CCR*. 2014.

[29] David D. Clark, Scott Shenker, and Lixia Zhang. "Supporting Real-time Applications in an Integrated Services Packet Network: Architecture and Mechanism". In: *ACM SIGCOMM*. Baltimore, MD, USA, 1992.

[30] The P4 Language Consortium. *P4-16 Language Specification, Version 1.1.0-rc*. 2018.

[31] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. "pFabric: Minimal Near-optimal Datacenter Transport". In: *ACM SIGCOMM*. Hong Kong, China, 2013.

[32] Pawan Goyal, Harrick M. Vin, and Haichen Chen. "Start-time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Switching Networks". In: *ACM SIGCOMM*. Palo Alto, CA, USA, 1996.

[33] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. "Approximating Fair Queueing on Reconfigurable Switches". In: *USENIX NSDI*. Renton, WA, USA, 2018.

[34] Sebastian Gallenmüller, Paul Emmerich, Daniel Raumer, and Georg Carle. "MoonGen: Software Packet Generation for 10 Gbit and Beyond". In: *USENIX NSDI*. Oakland, CA, USA, 2015.

[35] Juniper Networks. *Class of Service Feature Guide for Security Devices*. 2018.

[36] Ahmed Saeed, Yimeng Zhao, Nandita Dukkipati, Ellen Zegura, Mostafa Ammar, Khaled Harras, and Amin Vahdat. "Eiffel: Efficient and Flexible Software Packet Scheduling". In: *USENIX NSDI*. Boston, MA, USA, 2019.

[37] James McCauley, Aurojit Panda, Arvind Krishnamurthy, and Scott Shenker. "Thoughts on Load Distribution and the Role of Programmable Switches". In: *ACM SIGCOMM*. New York, NY, USA, 2019.

[38] Carmelo Cascone, Nicola Bonelli, Luca Bianchi, Antonio Capone, and Brunilde Sansò. "Towards Approximate Fair Bandwidth Sharing via Dynamic Priority Queuing". In: *IEEE LANMAN*. Osaka, Japan, 2017.

[39] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. "Information-Agnostic Flow Scheduling for Commodity Data Centers". In: *USENIX NSDI*. Oakland, CA, USA, 2015.

[40] Fernando J. Corbató, Marjorie Merwin-Daggett, and Robert C. Daley. "An Experimental Time-sharing System". In: *ACM AIEE-IRE*. New York, NY, USA, 1962.

[41] Zhuolong Yu, Chuheng Hu, Jingfeng Wu, Xiao Sun, Vladimir Braverman, Mosharaf Chowdhury, Zhenhua Liu, and Xin Jin. "Programmable Packet Scheduling with a Single Queue". In: *ACM SIGCOMM*. Virtual Event, USA, 2021.

[42] Naveen Kr Sharma, Chenxingyu Zhao, Ming Liu, Pravein G. Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. "Programmable Calendar Queues for High-speed Packet Scheduling". In: *USENIX NSDI*. Santa Clara, CA, USA, 2020.

[43] Tong Yang, Jizhou Li, Yikai Zhao, Kaicheng Yang, Hao Wang, Jie Jiang, Yinda Zhang, and Nicholas Zhang. "QCluster: Clustering Packets for Flow Scheduling". In: *ACM WWW*. Virtual, 2022.

[44] Peixuan Gao, Anthony Dalleggio, Yang Xu, and H. Jonathan Chao. "Gearbox: A Hierarchical Packet Scheduler for Approximate Weighted Fair Queuing". In: *USENIX NSDI*. Renton, WA, USA, 2022.

[45] Pooria Namyar, Behnaz Arzani, Ryan Beckett, Santiago Segarra, Himanshu Raj, Umesh Krishnaswamy, Ramesh Govindan, and Srikanth Kandula. "Finding Adversarial Inputs for Heuristics using Multi-level Optimization". In: *USENIX NSDI*. Santa Clara, CA, USA, 2024.

[46] *Broadcom StrataXGS Switch Solutions*. 2023.

[47] Balázs Vass, Csaba Sarkadi, and Gábor Rétvári. "Programmable Packet Scheduling With SP-PIFO: Theory, Algorithms and Evaluation". In: *IEEE INFOCOM Workshops*. 2022.

[48] Nichols, Kathleen and Jacobson, Van. "Controlling Queue Delay". In: *Communications of the ACM*. 2012.

[49] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. "Swift: Delay is Simple and Effective for Congestion Control in the Datacenter". In: *ACM SIGCOMM*. Virtual Event, USA, 2020.

[50] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. "Data Center TCP (DCTCP)". In: *ACM SIGCOMM*. New Delhi, India, 2010.

[51] Pooria Namyar, Behnaz Arzani, Ryan Beckett, Santiago Segarra, Himanshu Raj, and Srikanth Kandula. "Minding The Gap Between Fast Heuristics and Their Optimal Counterparts". In: *ACM HotNets*. 2022.

[52] Anurag Agrawal and Changhoon Kim. "Intel Tofino2: A 12.9 Tbps P4-Programmable Ethernet Switch". In: *IEEE HotChips*. 2020.

[53] Zhuolong Yu, Jingfeng Wu, Vladimir Braverman, Ion Stoica, and Xin Jin. "Twenty Years After: Hierarchical Core-Stateless Fair Queueing". In: *USENIX NSDI*. 2021.

[54] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. "Moongen: A Scriptable High-Speed Packet Generator". In: *ACM IMC*. Tokyo, Japan, 2015.

[55] Alan Demers, Srinivasan Keshav, and Scott Shenker. "Analysis and Simulation of a Fair Queuing Algorithm". In: *ACM SIGCOMM*. New York, NY, USA, 1989.

[56] Paul E McKenney. "Stochastic Fairness Queueing". In: *IEEE INFOCOM*. 1990.

[57] M. Shreedhar and George Varghese. “Efficient Fair Queueing Using Deficit Round Robin”. In: *ACM SIGCOMM*. Cambridge, Massachusetts, USA, 1995.

[58] Linus E Schrage and Louis W Miller. “The Queue M/G/1 with the Shortest Remaining Processing Time Discipline”. In: *INFORMS Operations Research*. 1966.

[59] Vishal Shrivastav. “Fast, Scalable, and Programmable Packet Scheduler in Hardware”. In: *ACM SIGCOMM*. Beijing, China, 2019.

[60] Ruyi Yao, Zhiyu Zhang, Gaojian Fang, Peixuan Gao, Sen Liu, Yibo Fan, Yang Xu, and H. Jonathan Chao. “BMW Tree: Large-scale, High-throughput and Modular PIFO Implementation using Balanced Multi-Way Sorting Tree”. In: *ACM SIGCOMM*. New York, NY, USA, 2023.

[61] Nirav Atre, Hugo Sadok, and Justine Sherry. “BBQ: A Fast and Scalable Integer Priority Queue for Hardware Packet Scheduling”. In: *USENIX NSDI*. Santa Clara, CA, USA, 2024.

[62] Peixuan Gao, Anthony Dalleggio, Jiajin Liu, Chen Peng, Yang Xu, and H. Jonathan Chao. “Sifter: An Inversion-Free and Large-Capacity Programmable Packet Scheduler”. In: *USENIX NSDI*. Santa Clara, CA, USA, 2024.

[63] Sarah McClure, Zeke Medley, Deepak Bansal, Karthick Jayaraman, Ashok Narayanan, Jitendra Padhye, Sylvia Ratnasamy, Anees Shaikh, and Rishabh Tewari. “Invisinets: Removing Networking from Cloud Networks”. In: *USENIX NSDI*. Boston, MA, 2023.

[64] Jiarong Xing, Kuo-Feng Hsu, Matty Kadosh, Alan Lo, Yonatan Piasetzky, Arvind Krishnamurthy, and Ang Chen. “Runtime Programmable Switches”. In: *USENIX NSDI*. Renton, WA, USA, 2022.

[65] Xiangyu Gao, Taegyun Kim, Michael D. Wong, Divya Raghunathan, Aatish Kishan Varma, and Pravein et al. Kannan. “Switch Code Generation Using Program Synthesis”. In: *ACM SIGCOMM*. Virtual, 2020.

[66] Brent Stephens, Aditya Akella, and Michael Swift. “Loom: Flexible and Efficient NIC Packet Scheduling”. In: *USENIX NSDI*. Boston, USA, 2019.

[67] Anshuman Mohan, Yunhe Liu, Nate Foster, Tobias Kappé, and Dexter Kozen. “Formal Abstractions for Packet Scheduling”. In: *OOPSLA*. Cascais, Portugal, 2023.

[68] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. "Lyra: A Cross-Platform Language and Compiler for Data Plane Programming on Heterogeneous ASICs". In: *ACM SIGCOMM*. Virtual, 2020.

[69] Nik Sultana, John Sonchack, Hans Giesen, Isaac Pedisich, Zhaoyang Han, Nishanth Shyamkumar, Shivani Burad, André DeHon, and Boon Thau Loo. "Flightplan: Dataplane Disaggregation and Placement for P4 Programs". In: *USENIX NSDI*. 2021.

[70] Sol Han and Seokwon Jang et al. "Virtualization in Programmable Data Plane: A Survey and Open Challenges". In: *IEEE OJ-COMS*. 2020.

[71] Andreas Blenk, Arsany Basta, Martin Reisslein, and Wolfgang Kellerer. "Survey on Network Virtualization Hypervisors for Software Defined Networking". In: *IEEE Communications Surveys and Tutorials*. 2016.

[72] Hang Zhu, Tao Wang, Yi Hong, Dan RK Ports, Anirudh Sivaraman, and Xin Jin. "NetVRM: Virtual Register Memory for Programmable Networks". In: *USENIX NSDI*. Renton, WA, USA, 2022.

[73] Peng Zheng, Theophilus Benson, and Chengchen Hu. "P4visor: Lightweight Virtualization and Composition Primitives for Building and Testing Modular Programs". In: *ACM CoNEXT*. Heraklion, Greece, 2018.

[74] Cheng Zhang, Jun Bi, Yu Zhou, Abdul Basit Dogar, and Jianping Wu. "HyperV: A High Performance Hypervisor for Virtualization of the Programmable Data Plane". In: *IEEE ICCCN*. Vancouver, Canada, 2017.

[75] Tao Wang, Xiangrui Yang, Gianni Antichi, Anirudh Sivaraman, and Aurojit Panda. "Isolation Mechanisms for High-Speed Packet-Processing Pipelines". In: *USENIX NSDI*. Renton, WA, USA, 2022.

[76] Tao Wang, Hang Zhu, Fabian Ruffy, Xin Jin, Anirudh Sivaraman, Dan R. K. Ports, and Aurojit Panda. "Multitenancy for Fast and Programmable Networks in the Cloud". In: *USENIX HotCloud*. Virtual, 2020.

[77] Mateus Saquetti, Guilherme Bueno, Weverton Cordeiro, and Jose Azambuja. "P4VBox: Enabling P4-Based Switch Virtualization". In: *IEEE Communications Letters*. 2020.

[78] Xin Jin, Jennifer Gossels, Jen Rexford, and David Walker. "CoVisor: A Compositional Hypervisor for SDN". In: *USENIX NSDI*. Oakland, CA, 2015.

[79] Radostin Stoyanov and Noa Zilberman. "MTPSA: Multi-Tenant Programmable Switches". In: *EuroP4*. Barcelona, Spain, 2020.

[80] Hasanin Harkous, Chrysa Papagianni, Koen De Schepper, Michael Jarschel, Marinos Dimolianis, and Rastin Pries. "Virtual Queues for P4: A Poor Man's Programmable Traffic Manager". In: *IEEE TNSM*. 2021.

[81] Yan-Wei Chen, Chi-Yu Li, Chien-Chao Tseng, and Min-Zhi Hu. "P4-TINS: P4-Driven Traffic Isolation for Network Slicing With Bandwidth Guarantee and Management". In: *IEEE TNSM*. 2022.

[82] David Hancock and Jacobus Merwe. "HyPer4: Using P4 to Virtualize the Programmable Data Plane". In: *ACM CoNEXT*. Irvine, CA, 2016.

[83] Proton Team. *Message Regarding the ProtonMail DDoS Attacks*. 2015.

[84] *Hidden Threat of Pulse Wave DDoS attacks*. Publication Title: DDoS-Guard Blog. 2019.

[85] Igal Zeifman. *Attackers Use DDoS Pulses to Pin Down Multiple Targets*. Publication Title: Imperva Blog. 2017.

[86] *Pulse-Wave DDoS Attacks Mark a New Tactic in Q2*. Publication Title: InfoSecurity Magazine. 2017.

[87] Alethea Toh. *Azure DDoS Protection—2021 Q3 and Q4 DDoS attack trends*. Publication Title: Microsoft Azure Blog. 2022.

[88] Jai Vijayan. *'Pulse Wave' DDoS Attacks Emerge As New Threat*. Publication Title: Dark Reading News. 2017.

[89] Damian Menscher. *Practical Solutions for Amplification Attacks*. 2019.

[90] Seyed K Fayaz, Yoshiaki Tobioka, Vyas Sekar, and Michael Bailey. "Bohatei: Flexible and Elastic DDoS Defense". In: *USENIX Security*. Washington, DC, USA, 2015.

[91] Menghao Zhang, Guanyu Li, Shicheng Wang, Chang Liu, Ang Chen, Hongxin Hu, Guofei Gu, Qi Li, Mingwei Xu, and Jianping Wu. "Poseidon: Mitigating Volumetric DDoS Attacks with Programmable Switches". In: *NDSS Symposium*. San Diego, CA, USA, 2020.

[92] Thomas Lintemuth, Patrick Hevesi, and Sushil Aryal. "Solution Comparison for DDoS Cloud Scrubbing Centers". In: *Gartner*. 2020.

[93] Akamai. *Prolexic Routed: Protect Your Entire Application Infrastructure Against Large, Complex DDoS Attacks*. 2020.

[94] Cloudflare. *Cloudflare Advanced DDoS Protection*. 2017.

[95] Radware. *DefensePro: Advanced DDoS Defense and Attack Mitigation*. 2021.

[96] Jared M. Smith and Max Schuchard. "Routing Around Congestion: Defeating DDoS Attacks and Adverse Network Conditions via Reactive BGP Routing". In: *IEEE S&P*. San Francisco, CA, USA, 2018.

[97] Muoi Tran, Min Suk Kang, Hsu-Chun Hsiao, Wei-Hsuan Chiang, Shu-Po Tung, and Yu-Su Wang. "On the Feasibility of Rerouting-Based DDoS Defenses". In: *IEEE S&P*. San Diego, CA, USA, 2019.

[98] Zaoxing Liu, Georgios Nikolaidis, Jeongkeun Lee, Changhoon Kim, Xin Jin, Vladimir Braverman, Minlan Yu, and Vyas Sekar. "Jaqen: A High-Performance Switch-Native Approach for Detecting and Mitigating Volumetric DDoS Attacks with Programmable Switches". In: *USENIX Security*. Virtual, 2021.

[99] Jiarong Xing, Wenqing Wu, and Ang Chen. "Ripple: A Programmable, Decentralized Link-Flooding Defense Against Adaptive Adversaries". In: *USENIX Security*. 2021.

[100] Yuhei Hayashi, Chen Meiling, and Li Su. *Use Cases for DDoS Open Threat Signaling (DOTS) Telemetry*. Internet-Draft.

[101] Link 11. *The Evolution of DDoS Reflection Amplification Vectors: a Chronology*. 2020.

[102] Catalin Cimpanu. *FBI Warns of New DDoS Attack Vectors: CoAP, WS-DD, ARMS, and Jenkins*. 2020.

[103] Robin Sommer and Vern Paxson. "Outside the Closed World: On Using Machine Learning for Network Intrusion Detection". In: *IEEE S&P*. Oakland, CA, 2010.

[104] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. "Kitsune: An Ensemble of Autoencoders for Online Network Intrusion Detection". In: *NDSS Symposium*. San Diego, CA, USA, 2018.

[105] Alexandre da Silveira Ilha, Ângelo Cardoso Lapolli, Jonatas Adilson Marques, and Luciano Paschoal Gaspary. "Euclid: A Fully In-Network, P4-based Approach for Real-Time DDoS Attack Detection and Mitigation". In: *IEEE TNSM*. 2020.

[106] Mahajan, Ratul and Bellovin, Steven M. and Floyd, Sally and Ioannidis, John and Paxson, Vern and Shenker, Scott. "Controlling High Bandwidth Aggregates in the Network". In: *ACM SIGCOMM CCR*. New York, NY, USA, 2002.

[107] Min Suk Kang, Virgil D Gligor, Vyas Sekar, et al. "SPIFFY: Inducing Cost-Detectability Tradeoffs for Persistent Link-Flooding Attacks". In: *NDSS Symposium*. San Diego, CA, USA, 2016.

[108] Ali Kheradmand. "Automatic Inference of High-Level Network Intents by Mining Forwarding Patterns". In: *ACM SOSR*. San Jose, CA, USA, 2020.

[109] Moses Charikar, Chandra Chekuri, Tomás Feder, and Rajeev Motwani. "Incremental Clustering and Dynamic Information Retrieval". In: *SIAM JoC*. 2004.

[110] Sudipto Guha, Adam Meyerson, Nina Mishra, Rajeev Motwani, and Liadan O'Callaghan. "Clustering Data Streams: Theory and Practice". In: *IEEE TKDE*. 2003.

[111] Sanjoy Dasgupta. *Notes for CSE 291: Topics in Unsupervised Learning, Lecture 6: Online and Streaming Algorithms for Clustering*. 2008.

[112] Albert Bifet, Ricard Gavaldà, Geoff Holmes, and Bernhard Pfahringer. "Machine Learning for Data Streams with Practical Examples in MOA". In: *MIT Press*. 2018.

[113] Rui Xu and Don Wunsch. "Clustering (Chapter 5.2.1)". In: *John Wiley & Sons*. 2008.

[114] Edgecore Networks. *The Edgecore Networks DCS810 Switch with Tofino2*. 2017.

[115] Intel. *Tofino 3 Intelligent Fabric Processor*.

[116] CAIDA. *Anonymized Traces from CAIDA Equinix NYC Internet Data Collection Monitor-B, 2018*. 2018.

[117] Catalin Cimpanu. *'Carpet-bombing' DDoS Attack Takes Down South African ISP for an Entire Day*. 2019.

[118] Steinthor Bjarnason. *DDoS Defences in the Terabit Era: Attack Trends, Carpet Bombing*. 2018.

[119] Tiago Heinrich, Rafael R Obelheiro, and C. Alberto Maziero. "New Kids on the DRDoS Block: Characterizing Multiprotocol and Carpet Bombing Attacks". In: *ACM PAM*. 2021.

[120] Iman Sharafaldin, Arash Habibi Lashkari, Saqib Hakak, and Ali A Ghorbani. "Developing Realistic Distributed Denial of Service (DDoS) Attack Dataset and Taxonomy". In: *ICCST*. 2019.

[121] Christopher D. Manning, Hinrich Schütze, and Prabhakar Raghavan. "Introduction to Information Retrieval (Chapter 16.3)". In: *Cambridge University Press*. 2008.

[122] Oleg Kupreev, Ekaterina Badovskaya, and Alexander Gutnikov. *Kaspersky DDoS Reports. DDoS attacks in Q1 2020*. 2020.

[123] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. "Understanding the Mirai Botnet". In: *USENIX Security*. Vancouver, Canada, 2017.

[124] Ron Winward. *Mirai Inside of an IoT Botnet*. 2017.

[125] Cloudflare. *Technical Details Behind a 400Gbps NTP Amplification DDoS Attack*. 2014.

[126] Akamai. *Memcached-fueled 1.3 Tbps Attacks*. 2018.

[127] Matthew Prince. *The DDoS That Almost Broke the Internet*. 2013.

[128] Proton Team. *Guide to DDoS Protection*. 2015.

[129] Ilya V. Chugunkov, Leonid O. Fedorov, Bela Sh. Achmiz, and Zarina R. Sayfullina. "Development of the Algorithm for Protection Against DDoS-attacks of Type Pulse Wave". In: *IEEE EIConRus*. 2018.

[130] Chen, Yu and Hwang, Kai and Kwok, Yu-Kwong. "Collaborative Defense Against Periodic Shrew DDoS Attacks in Frequency Domain". In: *ACM TISSEC*. 2005.

[131] Supranamaya Ranjan, Ram Swaminathan, Mustafa Uysal, and Edward W. Knightly. "DDoS-Resilient Scheduling to Counter Application Layer Attacks Under Imperfect Detection". In: *IEEE INFOCOM*. Barcelona, Spain, 2006.

[132] Chu-Hsing Lin, Jung-Chun Liu, Hsun-Chi Huang, and Tsung-Che Yang. "Using Adaptive Bandwidth Allocation Approach to Defend DDoS Attacks". In: *IEEE MUE*. Busan, Korea, 2008.

[133] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. "Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities". In: *ACM SIGCOMM*. 2018, 221.