Everything Matters in Programmable Packet Scheduling

Albert Gran Alcoz [1], Balázs Vass [2], Pooria Namyar [3], Behnaz Arzani [4],
Gábor Rétvári [2], Laurent Vanbever [1]

[1] *ETH Zürich,* [2] *BME-TMIT,* [3] *USC,* [4] *Microsoft Research*

Abstract

Operators can deploy any scheduler they desire on existing switches through programmable packet schedulers: they tag packets with ranks (which indicate their priority) and schedule them in the order of these ranks. The ideal programmable scheduler is the Push-In First-Out (PIFO) queue, which schedules packets in a perfectly sorted order by "pushing" packets into any position of the queue based on their ranks. However, it is hard to implement PIFO queues in hardware due to their need to sort packets at line rate (based on their ranks).

Recent proposals approximate PIFO behaviors on existing data-planes. While promising, they fail to *simultaneously* capture both of the *necessary* behaviors of PIFO queues: their *scheduling behavior* and *admission control*. We introduce PACKS, an approximate PIFO scheduler that addresses this problem. PACKS runs on top of a set of priority queues and uses packet-rank information and queue-occupancy levels during enqueue to determine 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 (Intel Tofino).

1 Introduction

Packet scheduling is a classical problem in networking — it defines the time and order at which a buffer drains packets to optimize a given performance metric. Researchers have proposed many scheduling algorithms but most of them have never been deployed in production, due to the cost and time required to implement them on new ASIC designs [4].

Programmable schedulers solve this limitation: they build an abstraction that can represent all possible scheduling algorithms [22, 31–33]. The idea is that, if we find such an abstraction and we can implement it in hardware, then we can run any algorithm on top without need for new ASICs.

Figure 1: PACKS navigates the space between SP-PIFO [6] and AIFO [37], optimizing for both rank ordering and drops.

The first attempt is the "PIFO" abstraction, which relies on the observation that we can split most scheduling functions in: a ranking algorithm that indicates the priority with which each packet should be scheduled, and a queuing structure that can schedule packets in the order of these ranks. Push-In First Out (PIFO) queues serve as a natural candidate since they sort arbitrary packet sequences based on the packets' ranks at line rate — hence the name of the abstraction [32].

PIFO queues "push" packets into arbitrary positions in the queue based on their ranks and serve them from their head. This behavior allows them to satisfy the requirements for programmable packet scheduling: (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.

However, it is hard to implement PIFO queues in hardware because they need to sort packets at line rate (even after they have been enqueued); and they may have to drop high-rank packets after they have been enqueued (e.g., if a low-rank packet arrives). Recent works *approximate* PIFO's behaviors to provide implementations that can run on existing programmable data planes [6, 16, 28, 35, 37]. But these works *only* approximate *one* of the two key PIFO behaviors (Fig. 1).

For example, SP-PIFO [6], QCluster [35], AFQ [27], PCQ [28], and Gearbox [16], only approximate PIFO's *scheduling behavior*. They map incoming packets to priority queues to minimize the rank inversions at the output. How-

ever, 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. §2).

AIFO [37] 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 We propose 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 across various loads 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 Our main contributions are:

- PACKS, a programmable scheduler that emulates PIFO queues on top of a set of strict-priority queues (§3).
- An admission-control algorithm and a queue-mapping technique that approximate all PIFO behaviors (§4).
- A performance analysis of PACKS on MetaOpt [24] to study its performance gaps and adversarial inputs (§4.5).
- An implementation[1] of PACKS in Java and P4 (§5).
- An evaluation showing PACKS's effectiveness in approximating PIFO using simulations and hardware (§6).

[1]We will publicly release our code (also available via the PC chairs).

2 Background

We first describe SP-PIFO [6] (§2.1) and AIFO [37] (§2.2), two programmable packet schedulers that represent how prior works approximate PIFO's scheduling and admission control behavior, respectively (Fig. 1). We then motivate PACKS based on where these schedulers fall short (§2.3).

2.1 SP-PIFO

SP-PIFO [6] 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 number of 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.

2.2 AIFO

AIFO [37] 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).

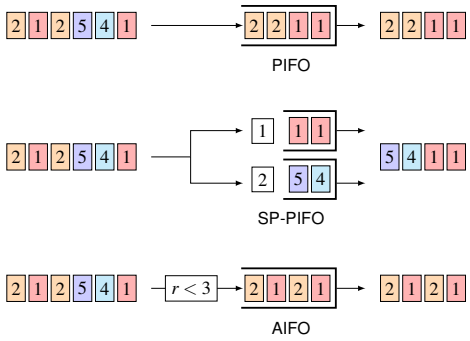


Figure 2: SP-PIFO and AIFO cannot fully approximate PIFO.

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. 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 $r < 3$.

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.

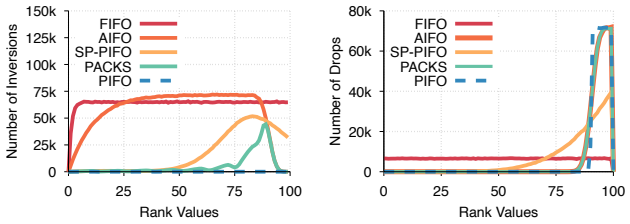
SP-PIFO maps packets 1 1 to the highest-priority queue, and packets 2 2 5 4 to the lowest-priority queue (c.f., §2.1). Since the lowest-priority queue only has room for two packets, it drops the last packets to arrive (2 2). The output sequence is 5 4 1 1, which has sorted ranks (it approximates PIFO's 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 does not prioritize any packet, 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. 5).

Experiment These limitations generalize across ranks. We implement SP-PIFO, AIFO, PACKS and FIFO in Netbench [2, 19], and schedule a stream of packets over a bottleneck link where the ranks are distributed uniformly across $[0, 100]$ (details in §6). 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. 3a). SP-PIFO approximates this behavior,



(a) Scheduling inversions (b) Packet drops

Figure 3: Scheduling performance, uniform rank distribution.

especially for lower-rank packets: it maps packets with lower ranks to higher-priority queues. AIFO 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. 3b) since it prioritizes low-rank packets. AIFO closely approximates PIFO's behavior [2] 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 because it combines the best of both worlds: an admission-control scheme, similar to 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 (§6.2).

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. 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.

[2] Note that the curve for AIFO and PACKS significantly overlap.

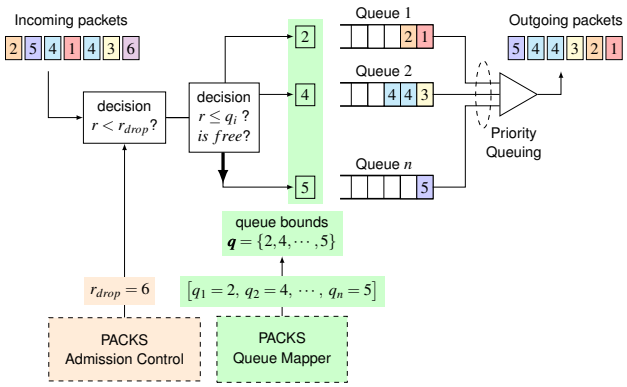


Figure 4: Overview of PACKS data-plane pipeline.

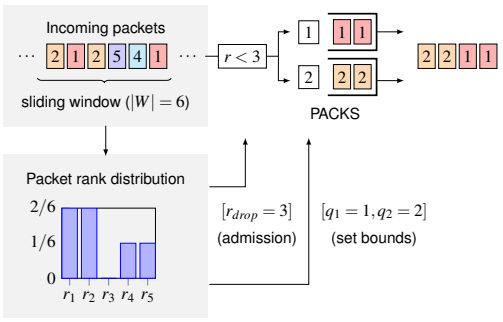


Figure 5: PACKS closely approximates PIFO's behavior.

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 into each priority 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 considers this estimate of the rank distribution to be the best possible up-to-date estimate.

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

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. 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. 2).

4 PACKS design

We now describe the theoretical basis supporting the design of PACKS. First, we frame the problem and introduce the design space (§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 (§4.2). Third, we generalize the algorithm to the online setup (§4.3). Finally, we formalize the PACKS's algorithm (§4.4), and analyze it both theoretically and with MetaOpt [24], an heuristic analyzer (§4.5).

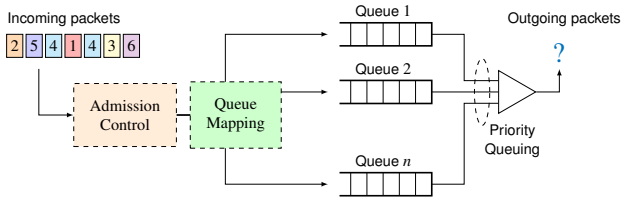


Figure 6: PACKS's design space.

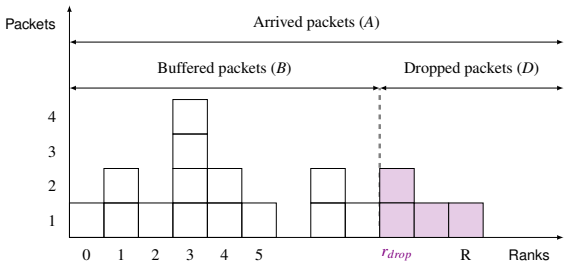


Figure 7: Admission control for a rank distribution, $\mathcal{W}$.

4.1 Design space

Let us consider the scheduling design space in Fig. 6, which represents the available resources in existing data planes [1,3]. 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. [3] 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.

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

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. 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:

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

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, 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.

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. 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 $r \leq q_i$.[4] 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 scheduler. We define a loss function $\mathcal{U}_S : \mathrm{R}^n \times$

[3] Some devices allow extra functionalities such as flexible priority-queue configuration, round-robin scheduling, or buffer management. We use Fig. 6's abstraction for generality and to guarantee line-rate processing for *all* packets.

[4] Note that PACKS scans queues top-down, while SP-PIFO bottom-up [6].

$R_{\geq 0} \rightarrow R_{\geq 0}$, 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 Q be the space of all valid queue-bound vectors and $\mathcal{W}$ the distribution of packet ranks. Then, $\boldsymbol{q}^*_{\mathcal{S}}$ are:

$$\boldsymbol{q}^*_{\mathcal{S}} = \arg\min_{\boldsymbol{q} \in Q} \mathcal{U}_{\mathcal{S}}(\boldsymbol{q}, r) \tag{2}$$

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:

$$\mathcal{U}_{\mathcal{S}}(\boldsymbol{q}, r) = \sum_{1 \leq i \leq n} \mathcal{U}_{\mathcal{S}}(q_i) \tag{3}$$

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:

$$\mathcal{U}_{\mathcal{S}}(q_i) = \sum_{\substack{q_{i-1} < r \leq q_i \\ r < r' \leq q_i}} p_{\mathcal{W}}(r) \cdot p_{\mathcal{W}}(r') \tag{4}$$

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, [34] proposes a modified version of the Bellman-Ford 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:

$$\begin{aligned} \hat{\mathcal{U}}_{\mathcal{S}}(q_i) &= \sum_{q_{i-1} < r \leq q_i} p_{\mathcal{W}}(r) \\ &= \mathcal{W}.quantile(q_i) - \mathcal{W}.quantile(q_{i-1}) \end{aligned} \tag{5}$$

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. 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.

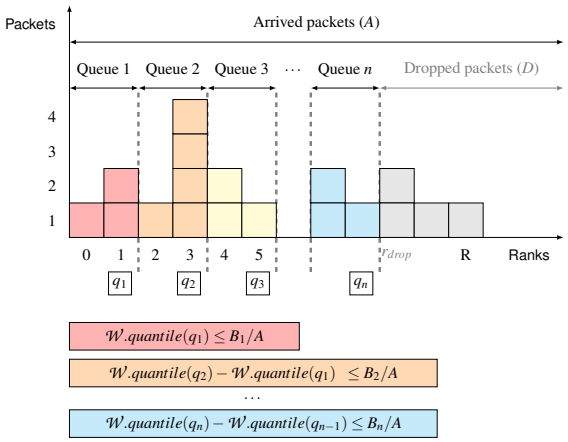


Figure 8: Queue mapping for a rank distribution, $\mathcal{W}$.

Therefore, 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 priority 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 usage of the buffer space.

In the following, we 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 output of the scheduler, $\boldsymbol{q}^*_{\mathcal{S}}$.

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:

$$\begin{aligned} m_1 &= [A \cdot \mathcal{W}.quantile(q_1)] \\ m_2 &= [A \cdot \mathcal{W}.quantile(q_2)] - m_1 \\ m_n &= [A \cdot \mathcal{W}.quantile(q_n)] - m_{n-1} \end{aligned} \tag{6}$$

We define a loss function $\mathcal{U}_{\mathcal{D}} : R^n \times R_{\geq 0} \rightarrow R_{\geq 0}$, 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:

$$\boldsymbol{q}^*_{\mathcal{D}} = \underset{\boldsymbol{q} \in \mathcal{Q}}{\arg\min}\, \mathcal{U}_{\mathcal{D}}(\boldsymbol{q}) \tag{7}$$

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

$$\mathcal{U}_{\mathcal{D}}(\boldsymbol{q}) = \sum_{1 \leq i \leq n} \mathcal{U}_{\mathcal{D}}(q_i) \tag{8}$$

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, if the number of packets mapped to the queue is greater than the queue space, or 0, otherwise:

$$\mathcal{U}_{\mathcal{D}}(q_i) = \begin{cases} m_i - B_i & \text{if } m_i > B_i \\ 0 & \text{otherwise.} \end{cases} \tag{9}$$

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., $m_i \leq B_i$):

$$\forall i : A \cdot (\mathcal{W}.quantile(q_i) - \mathcal{W}.quantile(q_{i-1})) \leq B_i \tag{10}$$

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. 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 to the queue i are carried over to the next queue, $i+1$, which has to account them as part of 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.5 and eq.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 §6. As a result, we leverage $\boldsymbol{q}^*_{\mathcal{D}}$, as the queue bounds for our design.

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 [34, 37], 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. [5] As a result, given b, the buffer-occupancy level at a certain packet's enqueue time, we decide to admit the packet if $W.quantile(r_{drop} - 1) \leq \left[\frac{1}{1-k} \cdot \frac{B-b}{B} \right]$, where k is an optional

[5] This is a common approach in queue-management [18, 25, 37]. We could also have used the *sojourn-time* of packets, as proposed by CoDel [25].

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:

$$
\begin{aligned}
q_1 &:= \max_{r_1 \in \mathbb{N}} s.t.\ W.quantile(r_1) \ \leq \ \frac{1}{1-k} \cdot \left[\frac{(B_1 - b_1)}{B} \right] \\
q_2 &:= \max_{r_2 \in \mathbb{N}} s.t.\ W.quantile(r_2) \leq \\
&\qquad \leq \ \frac{1}{1-k} \cdot \left[\frac{(B_1 - b_1)}{B} + \frac{(B_2 - b_2)}{B} \right] \qquad (11) \\
&\quad \ldots \\
q_n &:= \max_{r_n \in \mathbb{N}} s.t.\ W.quantile(r_n) \leq \frac{1}{1-k} \left[\frac{\sum_{j=1}^{n} (B_j - b_j)}{B} \right]
\end{aligned}
$$

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. 1).

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.

4.4 PACKS algorithm

Algorithm 1 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** $W.quantile(r) \leq \frac{1}{1-k} \cdot \left[\frac{\sum_{j=1}^{i} (B_j - b_j)}{B} \right]$ **then**
7: **if** $b_i < B_i$ **then** ▷ Queue i not full
8: $Queues(i).enqueue(pkt)$ ▷ Select queue
9: **return;**
10: $Drop(pkt)$ ▷ Drop packet

We detail the PACKS algorithm in alg 1. 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 for the monitored rank distribution is lower than the fraction of available buffer space: $W.quantile(r) \leq \left[\frac{1}{1-k} \cdot \frac{B-b}{B} \right]$. Note that we weight the admission condition by an optional parameter, k, to allow for some burstiness. Also, note that in alg. 1, the admission condition is *implicit* in the queue-mapping process. Indeed, the drop action in line 10, executed when the packet has *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: $W.quantile(r) \leq \frac{1}{1-k} \cdot \left[\frac{\sum_{j=1}^{n} (B_j - b_j)}{B} \right]$. 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.

4.5 PACKS analysis

Similarly to other networking algorithms [7, 18, 20, 25, 34, 37], PACKS uses a window-based approach instead of a per-packet heuristic. As a result, PACKS outperforms under a stable rank distribution, if the window size is large enough to capture it.

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. 15), it also represents PACKS's Achilles' heel.

We evaluated PACKS on MetaOpt [24]–a recent heuristic analysis tool–, to understand its performance gap relative to SP-PIFO, AIFO and PIFO, and to identify adversarial inputs (cf. Appendix B). 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. B.3). In §6.1 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. In these cases, PACKS relaxes its admission criteria and its behavior converges to the one of per-packet heuristics such as SP-PIFO (cf. Fig 10).

In Appendix A, 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. 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).

5 Implementation

We implemented PACKS in P4$_{16}$ for Intel Tofino 2 [1] using 577 lines of code. Our implementation uses 12 stages and the resources outlined in Table 1. 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 examine 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 [37]). 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 by counting how many times the packet's rank is lower than a rank in the sliding window and then dividing this count by the window size. We perform this computation using the register's stateful ALU during the same register accesses used to update the sliding window. Indeed, while traversing all the window registers to update the oldest one with the rank of the new packet, we *also* compare the packet's rank with each register's value. We output the result of each comparison into a binary metadata field, $output_j$, that is set to 1 if the packet rank is smaller than the register value and 0 otherwise. To compute the rank's quantile, we sum all the output values and then divide the result by the total number of registers, $|W|$: $W.quantile(r) = (\sum_j output_j)/|W|$. We aggregate the output values by progressively summing pairs of them at each stage using non-stateful ALUs, which requires $\log_2 |W|$ stages.

Queue-occupancy monitoring We use a *ghost thread* [5], 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 cross the traffic manager to access it. We solve this problem by setting up a ghost thread that periodically writes each queue's occupancy (from the egress) to a register accessible from the ingress pipeline. This ghost thread updates each queue's state in two clock cycles, handling one queue per invocation, resulting in a total of 8 clocks to update the state of all queues. To run PACKS on a very large set of queues and ports, we can either use traditional packet-recirculation to convey queue-occupancy information to the ingress (as done in previous works [37]), or we can approximate the admission and queue-mapping conditions by considering the overall buffer occupancy instead of the per-queue usage (e.g., $W.quantile(r) \leq \frac{1}{1-k} \cdot \frac{B-b}{B} \cdot \frac{i}{n}$). The first option sacrifices throughput, while the second, accuracy.

Admission and queue mapping After obtaining the quantile of the packet's rank based on the monitored rank distribution, $W.quantile(r)$, and the queue occupancies, b_i, we combine them to derive the admission and mapping conditions. We rewrite them as: $B \cdot (1-k) \cdot W.quantile(r) \leq \sum_{j=1}^{i}(B_j - b_j)$ (cf. §4). To compute the right side of the equation, we first determine the available space in each queue using the math unit, which requires one stage, and then sequentially sum them using one stage per queue. Simultaneously, we compute the left side of the equation by picking a k 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 select the corresponding drop or enqueue action based on its outcome.

6 Evaluation

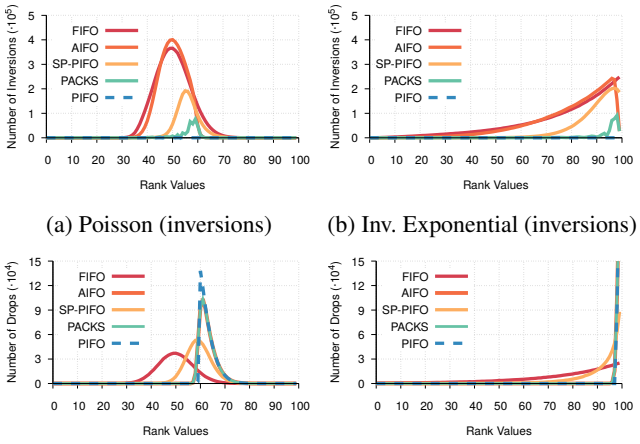
We evaluate PACKS's performance in three steps. First, we study its performance in approximating PIFO's scheduling and admission behaviors for various rank distributions, and analyze its sensitivity to the configuration parameters (§6.1). Second, we study its practicality even under complex traffic workloads (§6.2). Finally, we evaluate PACKS's reaction and bandwidth allocation when deployed on hardware (§6.3).

6.1 Performance analysis

First, we analyze PACKS's behavior across different rank distributions to assess its performance in approximating PIFO's packet admission and scheduling. Across all distributions, PACKS closely approximates ideal behavior and consistently outperforms the state-of-the-art, even in case of suboptimal window-size configurations and rank-distribution shifts.

Methodology We implement PACKS, PIFO, FIFO, SP-PIFO and AIFO in Netbench [2], 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, 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 §2.3, we have seen how PACKS outperforms existing schemes under a uniform rank distribution (cf. Fig.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×, 10× and 12× with respect to SP-PIFO, AIFO and FIFO. While all schemes drop a similar percentage of packets (within ±0.03%), PACKS



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

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

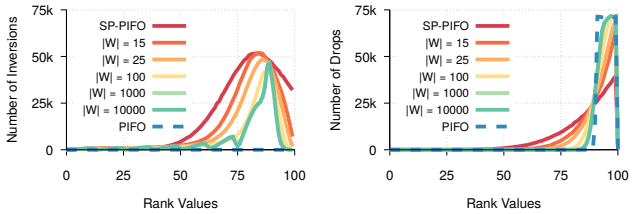
Figure 9: PIFO approximations for various rank distributions.

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. 9 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 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 [6]. 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),

[6]Note that PACKS's and AIFO's drop distribution significantly overlap.



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

Figure 10: PACKS's window-size sensitivity (UDP, uniform).

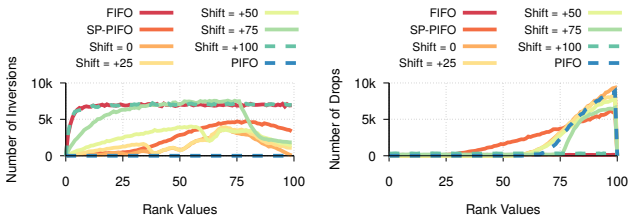
SP-PIFO drops 42% more packets than them. This is due to the highly skewed nature of the distribution, which is hard for SP-PIFO to manage without admission control (cf. §4.4).

Sensitivity to window size Fig. 10 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 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 so significantly, only reducing inversions by 1% and raising the lowest-dropped rank from 78 to 80, compared to $|W| = 1000$.

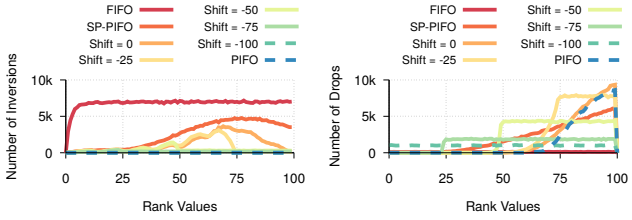
Window sizes below $|W| = 100$ lead to worse performance since the window cannot capture the entire distribution. Interestingly, as we reduce the window size, PACKS's behavior approaches that of SP-PIFO. Nevertheless, even with very small window sizes, PACKS still outperforms. For instance, with $|W| = 15$, 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. 11a and Fig. 11b 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

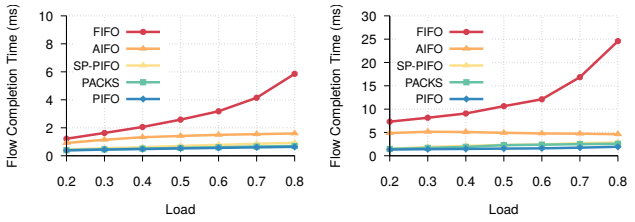


(a) Positive shift (inversions) (b) Positive shift (drops)

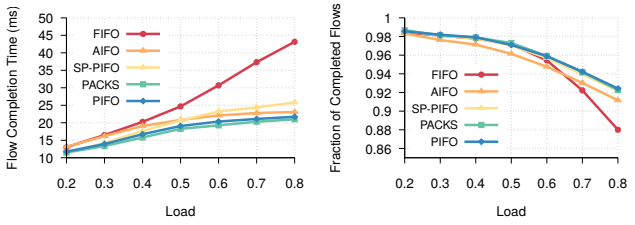


(c) Negative shift (inversions) (d) Negative shift (drops)

Figure 11: Rank-distribution sensitivity (TCP uniform).



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



(c) [All flows]: Average FCT (d) Fraction of completed flows

Figure 12: pFabric: FCT statistics across different flow sizes.

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. 11c and Fig. 11d 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.

6.2 Performance in typical use cases

We now study PACKS's performance under two common scheduling objectives: minimizing flow completion times and enforcing fairness [6,8,37,38]. These scenarios are especially challenging for PACKS's design because they involve *very large* and *non-stationary* rank distributions, which are hard to predict and act upon. Nevertheless, PACKS still achieves near-optimal performance, outperforming existing schemes.

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 to 1Gbps and 4Gbps, respectively. We generate traffic flows following the pFabric web-search workload [8]. 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.

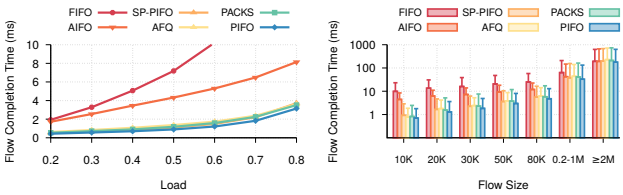
Setup pFabric We run pFabric [8] (without starvation prevention [7]) on top of PIFO, AIFO, SP-PIFO and PACKS, and assess their efficacy in minimizing flow completion times. pFabric assigns ranks to packets based on their remaining flow sizes. As suggested in [8], we approximate pFabric's rate control 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 $|W|$ to 20 packets, and k to 0.1.

Results pFabric Fig. 12 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.

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$\times$ to 2.6$\times$, and FIFO by 3.2$\times$ to 9.2$\times$ (biggest benefits under heavy loads).

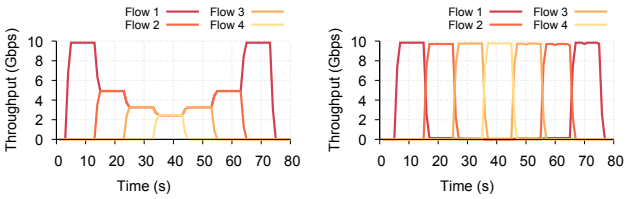
At the 99th pctl., PACKS achieves FCTs 8% to 49% longer than PIFO, but remains better than SP-PIFO (from 2.2% to 12% better), AIFO (1.8$\times$ to 3.3$\times$), and FIFO (5$\times$ to 10$\times$).

[7] Starvation [8] is a limitation inherent to PIFO, and therefore of all its approximations. Previous works have already proposed solutions to the starvation problem (e.g., PDA [35]) which can also run on PACKS.



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

Figure 13: Fairness: FCT statistics at different loads.



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

Figure 14: Bandwidth allocation for increasing-priority flows.

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. 12d). 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%).

Setup fair queuing We run the Start-Time Fair Queueing rank design [17] on top of the schedulers and evaluate their performance at enforcing fairness across flows. We compare to FIFO and AFQ [27] 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 [6,37], 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. 13a 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. 13b 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%.

6.3 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 [1]. Same as previous works [6, 37, 38], 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 [13, 14]. 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. 14 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.

7 Related work

Packet scheduling has been extensively studied for decades [8, 10–12, 17, 21, 22, 26, 27, 29]. The concept of programmable scheduling was introduced by [31, 32], 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 [30], BMW-Tree [36], BBQ [9], and Sifter [15]. Other works have focused on approximating PIFO behaviors on existing programmable data planes: SP-PIFO [6], QCluster [35], PCQ [28], AIFO [37], Spring [34], and Gearbox [16]. PACKS falls into the latter category.

8 Conclusions

We present 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 leverages packet-priority information and queue-occupancy levels during enqueue, to schedule packets in order of priority. We show that PACKS is practical, achieves close-to-PIFO behavior, and outperforms the state-of-the-art.

References

[1] The Edgecore Networks DCS810 Switch with Tofino2. https://www.edge-core.com/productsInfo.php?cls=1&cls2=349&cls3=577&id=916, 2017.

[2] Netbench. http://github.com/ndal-eth/netbench, 2018.

[3] Broadcom StrataXGS Switch Solutions. https://www.broadcom.com/products/ethernet-connectivity/switching, 2023.

[4] Network Programmability: The Road Ahead. https://www.youtube.com/watch?v=CtxfmES4T7E, 2023.

[5] Anurag Agrawal and Changhoon Kim. Intel Tofino2: A 12.9 Tbps P4-Programmable Ethernet Switch. In *IEEE Hot Chips Symposium (HCS 32)*, 2020.

[6] Albert Gran Alcoz, Alexander Dietmüller, and Laurent Vanbever. SP-PIFO: Approximating Push-In First-Out Behaviors using Strict-Priority Queues . In *USENIX NSDI*, Santa Clara, CA, USA, 2020.

[7] 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, 2011.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and Simulation of a Fair Queuing Algorithm. In *ACM SIGCOMM*, New York, NY, USA, 1989.

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

[14] 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.

[15] 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, April 2024.

[16] 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.

[17] 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.

[18] V. Jacobson. Congestion Avoidance and Control. In *ACM SIGCOMM '88*, Stanford, California, USA, 1988.

[19] 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.

[20] 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.

[21] Paul E McKenney. Stochastic Fairness Queueing. In *IEEE INFOCOM*, 1990.

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

[23] 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.

[24] 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.

[25] Kathleen Nichols and Van Jacobson. Controlling Queue Delay. In *ACM Queue*, New York, NY, USA, 2012.

[26] Linus E Schrage and Louis W Miller. The Queue M/G/1 with the Shortest Remaining Processing Time Discipline. 1966.

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

[28] 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.

[29] M. Shreedhar and George Varghese. Efficient Fair Queueing Using Deficit Round Robin. In *ACM SIGCOMM*, Cambridge, Massachusetts, USA, 1995.

[30] Vishal Shrivastav. Fast, Scalable, and Programmable Packet Scheduler in Hardware. In *SIGCOMM '19*, Beijing, China, 2019.

[31] 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.

[32] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. Programmable Packet Scheduling at Line Rate. In *ACM SIGCOMM*, Florianopolis, Brazil, 2016.

[33] 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.

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

[35] Tong Yang, Jizhou Li, Yikai Zhao, Kaicheng Yang, Hao Wang, Jie Jiang, Yinda Zhang, and Nicholas Zhang. QCluster: Clustering Packets for Flow Scheduling, 2020.

[36] 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.

[37] 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*, New York, NY, USA, 2021.

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

A 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 PIFO(t) and 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. In the following, we denote the the maximal and minimal rank probabilities with $\delta_+ := \max_i p(i)$ and $\delta_- := \min_i p(i)$.

Theorem 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 queue 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, again both for PIFO and PACKS. □

Next, we present an asymptotically tight upper bound for the number of inversions that PACKS produces for a packet sequence with respect 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 see 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 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 [37], 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.

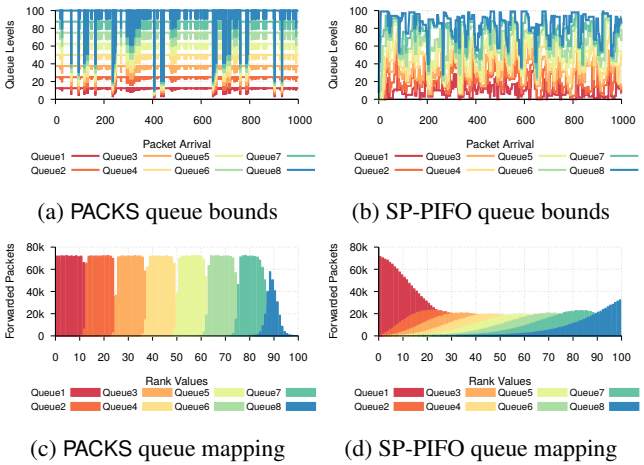
Case 1: PACKS enqueued r_t, while AIFO did not. Here we have the following inequalities explained below, yielding a contradiction:

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

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

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



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

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

Figure 15: Queue-bounds evolution and rank mapping for PACKS and SP-PIFO under a uniform distribution (8 queues).

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 *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. 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 $I_{\text{PACKS}} \leq I_{AIFO}$. 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. □

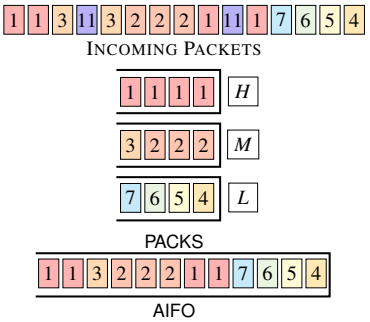


Figure 16: Adversarial input that maximizes the gap between weighted priority inversion of AIFO compared to PACKS. AIFO can delay the highest priority packet by more than 60% of the queue length. (Starting window = [1, 1, 1, 1]).

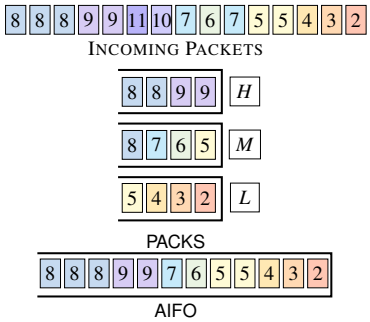


Figure 17: Adversarial input that maximizes the gap between weighted priority inversion of PACKS compared AIFO. (Starting window = [1, 1, 1, 1]).

B PACKS's analysis with MetaOpt

MetaOpt [23, 24] is a tool to compare the performance of two competing heuristics or a heuristic and an optimal solution. It identifies adversarial inputs that cause the maximum difference between the performance of the two algorithms.

We model PACKS in MetaOpt and compare it to AIFO [37], SP-PIFO [6], and PIFO [32]. 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 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.

B.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. 2.

Rank inversions Fig. 16 and Fig. 17 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. 16, 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. 17). Due to the distribution shift, PACKS ends up mapping higher-priority packets to lower-priority queues and lower-priority packets to higher-priority queues.

Note that, 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 packets arrive.

This adversarial sequence (Fig. 17) consists of packets with higher ranks than the adversarial input to AIFO (Fig. 16), 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 also prove theoretically in Theorem. 3).

B.2 Comparison with SP-PIFO

Packet drops Fig. 18 and Fig. 19 show the adversarial inputs that MetaOpt discovered for SP-PIFO with respect to PACKS and vice versa. We find 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 (all with rank 1). In this case, SP-PIFO maps all the packets to

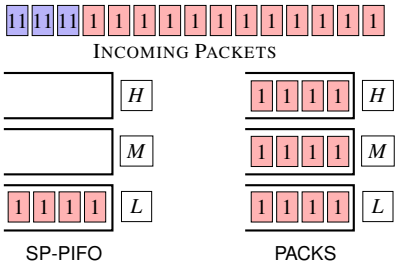


Figure 18: Adversarial input that maximizes the gap between weighted packet drop of SP-PIFO compared to PACKS. (Starting window = [1, 1, 1, 1]).

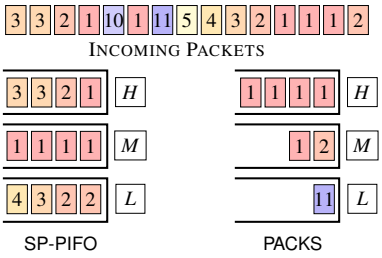


Figure 19: Adversarial input that maximizes the gap between weighted packet drop of PACKS compared to SP-PIFO. (Starting window = [1, 2, 1, 1]).

its lowest-priority queue and ends up 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).

Rank inversions In order to capture only the impact of rank inversions, we set the queue sizes long enough so that packet drops do not occur. Fig. 20 and Fig. 21 show the adversarial inputs that MetaOpt discovered. We see 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.

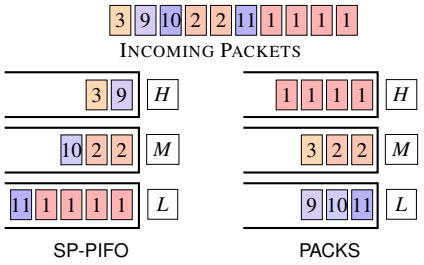


Figure 20: Adversarial input that maximizes the gap between weighted priority inversion of SP-PIFO compared to PACKS. (Starting window = [1, 1, 1, 1]).

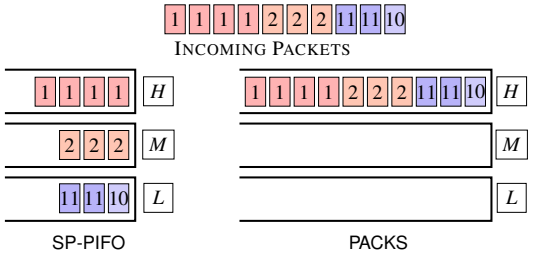


Figure 21: Adversarial input that maximizes the gap between weighted priority inversion of PACKS compared to SP-PIFO. (Starting window = [1, 1, 11, 11]).

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.

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.

B.3 Comparison with PIFO

Fig. 23 and Fig. 22 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)

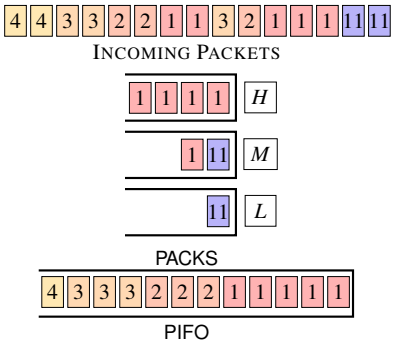


Figure 22: Adversarial input that maximizes the gap between weighted packet drop of PACKS compared to PIFO. (Starting window = [1, 1, 1, 1]).

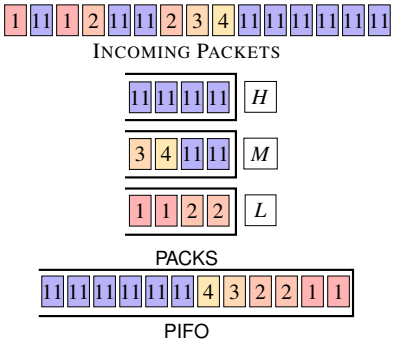


Figure 23: Adversarial input that maximizes the gap between weighted priority inversion of PACKS compared to PIFO. (Starting window = [1, 11, 1, 11]).

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. 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 quick insight after its proof).

C PACKS's resource usage

We describe the resource requirements of our implementation of PACKS on Intel Tofino 2 [1] in Table. 1.

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 1: Resource requirements of PACKS on Intel Tofino 2.