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

Albert Gran Alcoz, Alexander Dietmüller, and Laurent Vanbever, *ETH Zürich*

https://www.usenix.org/conference/nsdi20/presentation/alcoz

This paper is included in the Proceedings of the
17th USENIX Symposium on Networked Systems Design
and Implementation (NSDI '20)

February 25–27, 2020 • Santa Clara, CA, USA

978-1-939133-13-7

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

Albert Gran Alcoz
ETH Zürich

Alexander Dietmüller
ETH Zürich

Laurent Vanbever
ETH Zürich

Abstract

Push-In First-Out (PIFO) queues are hardware primitives which enable programmable packet scheduling by providing the abstraction of a priority queue at line rate. However, implementing them at scale is not easy: just hardware designs (not implementations) exist, which support only about 1k flows.

In this paper, we introduce SP-PIFO, a programmable packet scheduler which closely approximates the behavior of PIFO queues using strict-priority queues—*at line rate, at scale, and on existing devices*. The key insight behind SP-PIFO is to dynamically adapt the mapping between packet ranks and available strict-priority queues to minimize the scheduling errors with respect to an ideal PIFO. We present a mathematical formulation of the problem and derive an adaptation technique which closely approximates the optimal queue mapping without any traffic knowledge.

We fully implement SP-PIFO in P4 and evaluate it on real workloads. We show that SP-PIFO: (i) closely matches PIFO, with as little as 8 priority queues; (ii) scales to large amount of flows and ranks; and (iii) quickly adapts to traffic variations. We also show that SP-PIFO runs at line rate on existing hardware (Barefoot Tofino), with a negligible memory footprint.

1 Introduction

Until recently, packet scheduling was one of the last bastions standing in the way of complete data-plane programmability. Indeed, unlike forwarding whose behavior can be adapted thanks to languages such as P4 [7] and reprogrammable hardware [2], scheduling behavior is mostly set in stone with hardware implementations that can, at best, be configured.

To enable programmable packet scheduling, the main challenge was to find an appropriate abstraction which is flexible enough to express a wide variety of scheduling algorithms and yet can be implemented efficiently in hardware [22]. In [23], Sivaraman et al. proposed to use Push-In First-Out (PIFO) queues as such an abstraction. PIFO queues allow enqueued packets to be pushed in arbitrary positions (according to the packets rank) while being drained from the head.

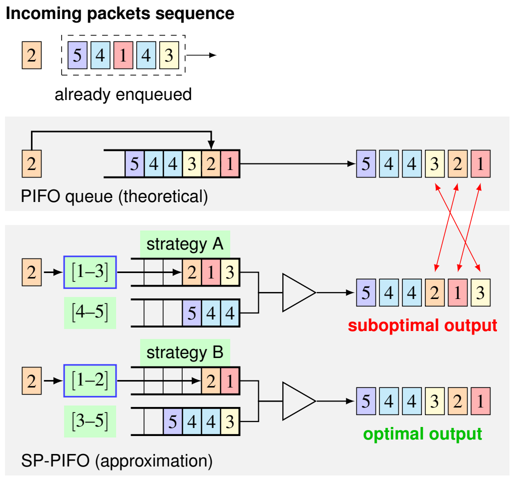


Figure 1: SP-PIFO approximates the behavior of PIFO queues by adapting how packet ranks are mapped to priority queues.

While PIFO queues enable programmable scheduling, implementing them in hardware is hard due to the need to arbitrarily sort packets at line rate. [23] described a possible hardware design (not implementation) supporting PIFO on top of Broadcom Trident II [1]. While promising, realizing this design in an ASIC is likely to take years [6], not including deployment. Even ignoring deployment considerations, the design of [23] is limited as it only supports ~1000 flows and relies on the assumption that the packet ranks increase monotonically within each flow, which is not always the case.

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

Example First, we provide an intuition how SP-PIFO approximates PIFO behaviors using SP queues in Fig. 1. The example illustrates the scheduling behavior of two SP-PIFO systems which receive the input packet sequence $\boxed{2}\boxed{5}\boxed{4}\boxed{1}\boxed{4}\boxed{3}$. By convention, we write the first packet being enqueued on the far-right ($\boxed{3}$) and the last one on the far-left ($\boxed{2}$). Similarly to [23], we also consider that lower-rank packets have higher priority (and use corresponding color codes). The figure illustrates the scheduling decision of each system for the sixth packet ($\boxed{2}$), assuming the first 5 have been enqueued already.

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

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

SP-PIFO approximates the best mapping strategy by dynamically shifting the ranks mapped to each queue to reduce the scheduling mistakes it observes in real time. We show that SP-PIFO's adaptation strategy achieves almost the same performance as provably-correct adaptation strategies while being implementable in programmable data planes.

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

Contributions Our main contributions are:

- A novel approach for approximating PIFO queues using strict-priority queues (§3).
- An adaptation algorithm which dynamically adapts the queue mapping according to the network conditions, closely-approximating an optimal scheme (§4).
- An implementation[1] of SP-PIFO in Java and P4 (§5).
- A comprehensive evaluation showing SP-PIFO effectiveness in approximating perfect PIFO behavior with as little as 8 queues and on actual hardware switches (§6).

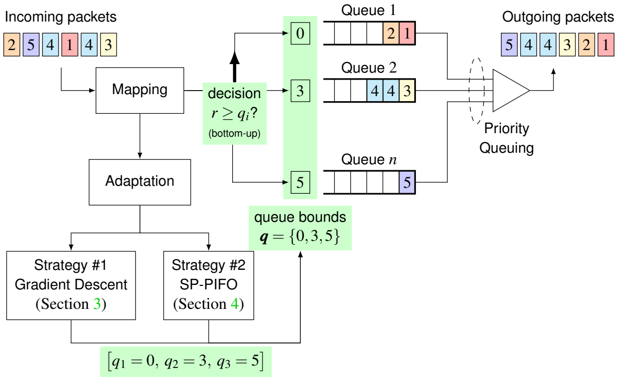


Figure 2: Overview of SP-PIFO data-plane pipeline.

2 Overview

In this section, we provide an informal overview of how SP-PIFO manages to closely approximate PIFO behaviors. At a high level, SP-PIFO is a priority-queuing scheduling discipline (see Fig. 2) which maps incoming packets to n priority queues. SP-PIFO assumes that packets are tagged with a rank indicating the intended scheduling order, with lower ranks being preferred over higher ones. Packets enqueued in a queue are scheduled according to their order of arrival (i.e., First-In First-Out), after *all* packets enqueued in any higher-priority queue have been scheduled. Unlike classical priority-queuing disciplines [20], SP-PIFO dynamically adapts the mapping between the packet ranks and the priority queues according to the observed network conditions. In particular, SP-PIFO adapts the mapping so as to minimize the scheduling "unpifoness", that is, the number of times a higher-rank packet is scheduled *before* an enqueued lower-rank packet. We refer to such scheduling mistakes as *inversions*.

Mapping SP-PIFO maps each incoming packet to queues according to the queue *bounds*. These queue bounds identify, for each queue i, the smallest packet rank that can be enqueued. Whenever a packet is received, SP-PIFO scans the queue bounds bottom-up, starting from the lowest-priority queue, and enqueues the packet in the first queue with a bound smaller or equal to the packet rank. Given a packet with rank $r \in \mathbb{Z}_{\geq 0}$ and n priority queues, let $\boldsymbol{q}$ be the vector of queue bounds $(q_1, \cdots, q_n) \in \mathbb{Z}^n$ such that $0 \leq q_1 \leq q_2 \leq \cdots \leq q_n$. For instance, consider a vector $\boldsymbol{q} = \{0,3,5\}$ indicating the bounds of 3 priority queues, with 0 (resp. 5) indicating the bound of the highest- (resp. lowest-) priority queue. Given $\boldsymbol{q}$, SP-PIFO enqueues packets with rank 2 in the first (highest-priority) queue, packets with rank 3 in the second queue and packets with rank 10 in the third (lowest-priority) queue.

Adaptation "Unpifoness" can be minimized across multiple packets, e.g. by monitoring the rank distribution over periodic time windows and adapting the bounds through a gradient descent, or on a per-packet basis (see Fig. 2). De-

[1] Available at https://github.com/nsg-ethz/sp-pifo

pending on the characteristics of the rank distribution, the first strategy can provably converge to the optimal mapping. Unfortunately, its requirements exceed the capabilities of existing programmable data planes. SP-PIFO addresses these two limitations: it works for *any* rank distribution, on existing hardware. SP-PIFO dynamically adapts **q** such that the resulting scheduling closely approximates an ideal PIFO queue, minimizing the amount of observed *inversions* by dynamically shifting the ranks mapped to each queue. SP-PIFO operates online, *without* prior knowledge of the incoming packet ranks.

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

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

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

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

The same process is applied to the subsequent packets until the second $\boxed{1}$ is encountered, creating an inversion (grayed area in Fig. 3). Indeed, SP-PIFO enqueues $\boxed{1}$ in the highest-priority queue *after* having enqueued $\boxed{2}$. Once the inversion is detected, SP-PIFO adapts the queue bounds to 1 and $5-1=4$, respectively. Observe that if $\boxed{1}$ and $\boxed{2}$ keep arriving, the bound of the lowest-priority queue will decrease, eventually reaching 2. At this point, future $\boxed{1}$ will not experience inversions anymore as they will have a dedicated queue.

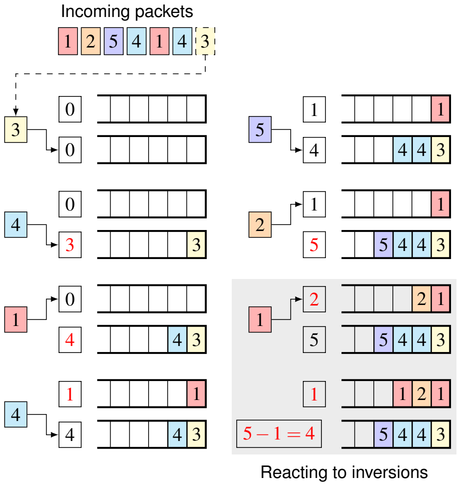


Figure 3: SP-PIFO mapping and adaptation mechanisms.

3 SP-PIFO design

In this section, we describe the theoretical basis supporting the design of SP-PIFO. We first phrase the problem of finding the optimal queue bounds as an empirical risk minimization problem in which a loss function—how "unpifo" the current mapping is—is minimized (§3.1). We then develop an algorithm based on gradient descent which provably converges to the optimal bounds for stable rank distributions (§3.2). We show how the convergence requirements make the algorithm impractical (§3.3). In the following, we present SP-PIFO which relaxes the requirements at the benefit of practicality (§4).

3.1 Problem statement

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

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

$$\boldsymbol{q}^* = \underset{\boldsymbol{q} \in \mathcal{Q}}{\arg\min} \underset{r \sim \mathcal{R}}{\mathbb{E}} \left[\, \mathcal{U}(\boldsymbol{q}, r) \, \right] \tag{1}$$

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

$$\boldsymbol{q}^* = \underset{\boldsymbol{q} \in \mathcal{Q}}{\arg\min} \frac{1}{|\mathcal{D}|} \sum_{r \in \mathcal{D}} \mathcal{U}_{emp}(\mathcal{D}, \boldsymbol{q}, r) \tag{2}$$

Evaluating empirical losses For a given rank r, we measure the empirical loss $\mathcal{U}_{emp}$ as the *expected* number of inversions that r would encounter, if the rank distribution $\mathcal{D}$ was scheduled given the queue bounds $\boldsymbol{q}$, weighted by the *cost* that each inversion would cause to the system performance. This cost can be just a constant value, if all inversions are treated the same, or it can measure the magnitude of the inversion (i.e., how big is the difference between ranks causing it). Since r receives inversions only from higher ranks in the distribution, $\mathcal{U}_{emp}$ can be rewritten as:

$$\mathcal{U}_{emp}(\mathcal{D},\boldsymbol{q},r) = \frac{1}{|\mathcal{D}|}\sum_{\substack{r'\in\mathcal{D}\\ r'>r}} \mathrm{cost}_{\boldsymbol{q}}(r',r) \tag{3}$$

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

3.2 Gradient-based adaptation algorithm

We first introduce a greedy, gradient-based algorithm, which provably converges to the optimal queue bounds $\boldsymbol{q}^*$ provided that the rank distribution stays constant. The algorithm builds upon the fact that inversions cannot occur between ranks mapped to different priority queues. This allows to instantiate the empirical risk minimization in eq. 2 at a *queue level* by simply adding the individual losses of each queue. Letting $\mathcal{U}(q_i)$ be the loss function corresponding to the queue with bound q_i, this is:

$$\boldsymbol{q}^* = \underset{\boldsymbol{q}\in\mathcal{Q}}{\arg\min} \sum_{q_i\in\boldsymbol{q}} \mathcal{U}(q_i) \tag{4}$$

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

$$\mathcal{U}(q_i) = \sum_{\substack{q_i\le r<q_{i+1}\\ r<r'<q_{i+1}}} p_{\mathcal{D}}(r)\cdot p_{\mathcal{D}}(r')\cdot \mathrm{cost}(r',r) \tag{5}$$

Overview Considering this problem instantiation, the greedy algorithm first computes the rank distribution over a set of k packets before minimizing the expected per-queue unpifoness by incrementing (resp. decrementing) the queue bounds. Specifically, after processing the k-th packet, the greedy algorithm selects, for each queue, the bound that most decreases the overall system unpifoness. Although comparing the performance of all bound combinations is not possible, we introduce an efficient computation mechanism that allows to prune the search space while preserving convergence. We prove the optimality of the algorithm in Appendix A.

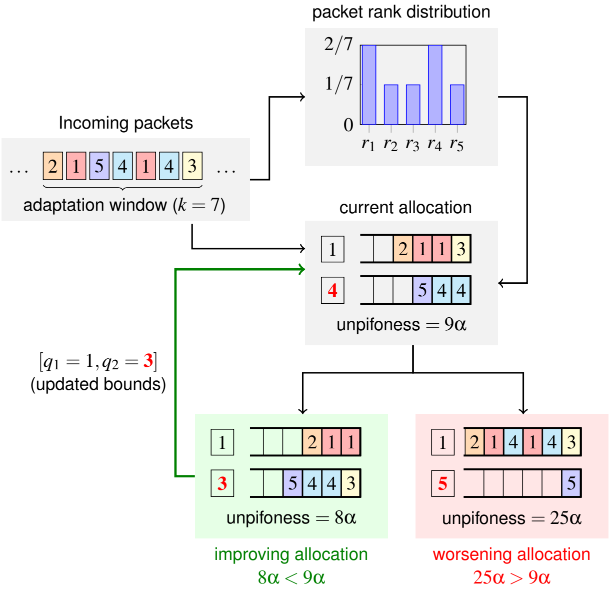


Figure 4: The gradient-based algorithm greedily minimizes the expected *unpifoness*.

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

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

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

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

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

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

3.3 Limitations

While the adaptation algorithm described above provably converges to the optimal mapping (see A.1), two key limitations make it impractical. First, it is not currently implementable in existing programmable data planes due to resource constraints. Second, the algorithm only converges for stable rank distributions, which is rarely the case, and its convergence time directly depends on the distribution size, which can be large. We explain how to overcome these limitations in §4.

Hardware restrictions Monitoring the rank distributions over periodic adaptation windows requires a high amount of memory and computational resources, both of which are scarce in current programmable data planes. In particular, implementing the greedy algorithm in hardware (see A.2) requires to: (i) store the value of each queue bound; (ii) compute the current unpifoness; and (iii) estimate the unpifoness obtained by incrementing or decrementing each queue bound. As we explain in A.3, the amount of resources required to run the algorithm on a practical number of queues (8 queues or more) exceeds the capabilities of current switch designs.

Convergence In A.4, we study the performance of the gradient-based algorithm and analyze the effects on convergence when the adaptation window, the number of queues, and the rank range is modified. We show that, for the algorithm to converge, the rank distribution needs to be stable in time. However, this is unrealistic in most practical scenarios where not only the rank distribution *is unknown* but also varies through time (e.g., virtual times in fair-queuing schemes).

4 Our approach: SP-PIFO

We now present SP-PIFO, an approximation of the gradient-based adaptation algorithm (§3.2) which is implementable in existing data planes and rapidly adapts to varying rank distributions. SP-PIFO substitutes the gradient computation by a simpler adaptation process which minimizes the probability of inversions *per packet*, rather than per *k*-packets.

In the following, we first show how to instantiate the empirical risk minimization problem (eq. 2) at the packet level and describe how SP-PIFO solves it (§4.1). We then systematically characterize how SP-PIFO handles inversions (§4.2).

4.1 Per-packet adaptation algorithm

The SP-PIFO adaptation algorithm (alg. 1) is based on two competing stages that act in opposing direction. We show that this combination manages to strike a balance in the number of inversions observed by all queues, resulting in a good PIFO approximation. In the following, we first show how to phrase the empirical risk minimization problem at the per-packet level before describing both mechanisms.

Problem statement In contrast to §3.2, we aim at minimizing the cost generated by scheduling each individual packet. Formally, we aim to find the optimal bound vector $\boldsymbol{q}^*$ that minimizes the unpifoness for all enqueued packets $\mathcal{P}$:

$$\boldsymbol{q}^* = \underset{\boldsymbol{q}\in\mathcal{Q}}{\arg\min}\ \mathcal{U}(\mathcal{P},\boldsymbol{q}) \tag{6}$$

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

$$\mathcal{U}(\mathcal{P},\boldsymbol{q}) = \sum_{p\in\mathcal{P}} \text{cost}_{\boldsymbol{q}}(p) \tag{7}$$

where

$$\text{cost}_{\boldsymbol{q}}(p) = r_p(p,\ \boldsymbol{q}) - r(p) \tag{8}$$

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

We discuss how we update these parameters as well as the tradeoffs of this relaxation below.

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

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

Algorithm 1 SP-PIFO adaptation algorithm

```
Require: An incoming packet with rank $r$.
procedure PUSH-UP
    for $q_i : q_1$ to $q_n$, $q_i \in \boldsymbol{q}$ do            ▷ Scan bottom-up
        if $r \geq q_i$ or $i = 1$ then
            $q_i \leftarrow r$                             ▷ Update queue bound
            ENQUEUE$(r, i)$                          ▷ Select queue
procedure PUSH-DOWN
    if $r < q_1$ then                                ▷ Detect inversion
        $cost \leftarrow q_i - r$                     ▷ Compute cost inversion
        for $q_j \in \boldsymbol{q}$, $j \neq i$ do
            $q_j \leftarrow q_j - cost$                ▷ Adapt queue bounds
```

Stage 2: "Push-down" As illustrated in §2, the first stage can lead to inversions in the highest-priority queue. The second stage aims at counteracting that effect by reducing the number of ranks enqueued in the highest-priority queue. This is achieved by decreasing *all* queue bounds by some given amount. Different decreasing strategies exist. In SP-PIFO, we decrease each q_i proportionally to the cost of the inversion. That is, we decrease all queue bounds by $q_1 - r(p)$. This choice is both (i) practical, as it can be efficiently implemented in hardware; and (ii) functional, as it results in a reasonable balance between inversions in the highest-priority queue and shifts in the other queues. Below, we provide some insights on the nature of this balance and why it is important for a good PIFO approximation. We simulate the performance of different decreasing strategies in §4.2.

Tradeoffs Unlike the gradient-based algorithm (§3.2), SP-PIFO may converge to a sub-optimal solution exhibiting inversions. One can distinguish three sources of inversions. First, there can be inversions in the highest-priority queue. These inversions are proportional to the probability of observing packets with rank $r(p) < q_1$. Second, after the "push-down" stage, the queue bounds do not necessarily match the highest rank packet in the queue anymore. This may lead to inversions for future packets and is proportional to how often, and how much, queue bounds are decreased. Finally, because only the highest rank in a queue is tracked, it can happen that a packet is enqueued in a higher-priority queue because $r(p) < q_i$, while $r(p)$ is greater than the *lowest* rank in queue i, causing an inversion. This is proportional to the number of ranks between the minimum rank in the queue and the queue bound.

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

As a result, on average workloads, SP-PIFO provides a good approximation, and can adapt to arbitrary rank distributions. Nevertheless, there are adversarial packet orderings circumventing these mechanisms, resulting in large unpifoness (§7). We provide the theoretical foundations for these statements in Appendix B and verify them by simulation in §4.2.

4.2 SP-PIFO analysis

We now dive deeper into understanding SP-PIFO using switch-level simulations. We compare its behavior to that of an ideal PIFO queue, along with several well-known scheduling schemes (e.g., FIFO). We first describe the high-level behavior using a uniform rank distribution (§4.2.1), before systematically exploring the design space (§4.2.2).

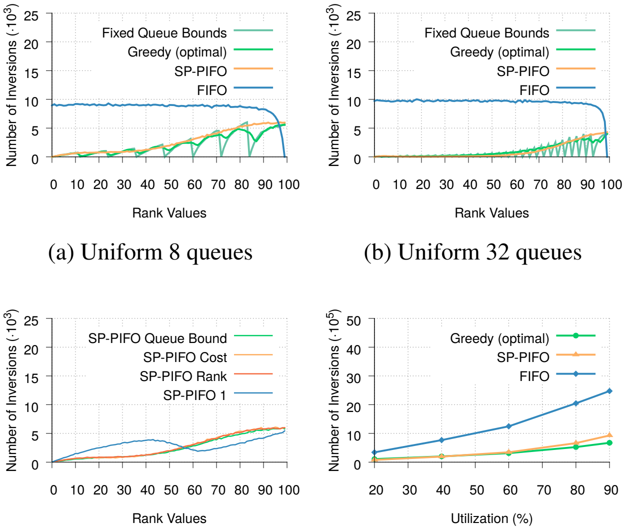
Methodology We implement various scheduling schemes (including SP-PIFO, FIFO, and our gradient-based algorithm) in Netbench [3, 15], a packet-level simulator. We analyze the performance of a single switch scheduling 1500 flows of 1MB (fixed), which start according to a Poisson distribution. We run the simulation during one second. We limit the transmission through an output link of 10 Gbps which corresponds to an average port utilization of 75%. We measure the number of inversions *generated by each rank* at *dequeue*. Whenever a packet is polled, we check whether its rank is higher than any of the ranks remaining at any of the queues. When this occurs, we count an inversion *to the rank generating it* (i.e., the one of the polled packet), making sure that inversions are counted at most once per polled-packet, regardless of the number of packets affected by it.

We compare four scheduling schemes: (i) SP-PIFO (§4); (ii) the gradient-based algorithm (§3, see implementation in A.2); (iii) a strict-priority scheme fixed to the optimal mapping for a uniform distribution (i.e., bounds distributed uniformly across ranks, $q_i = 12i$); and (vi) a FIFO queue, as baseline. All strict-priority schemes (SP schemes) use 8 queues of 10 packets, while the FIFO queue has a capacity of 80 packets.

4.2.1 Characterizing general SP-PIFO behavior

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

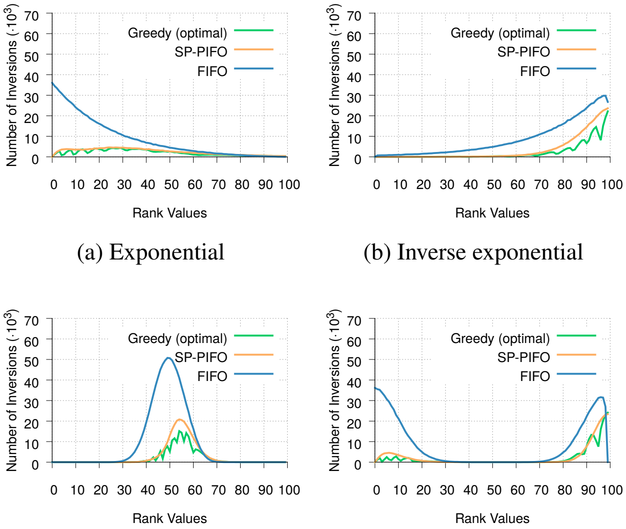
Fig. 5a illustrates the number of inversions generated by each rank for the different SP schemes in comparison with FIFO. We see that a FIFO queue generates a uniform number of inversions across all ranks (since they all share the same queue). In contrast, SP schemes (all the others in Fig. 5a) generate a progressively-higher number of inversions as rank val-



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

(c) Adaptation strategies (d) Utilization

Figure 5: SP-PIFO performance (uniform rank distribution).



(a) Exponential (b) Inverse exponential

(c) Poisson (d) Convex

Figure 6: SP-PIFO performance (alternative distributions).

ues increase. This occurs as higher ranks are mapped to lower-priority queues, which drain packets less frequently. Since those queues have a higher occupancy on average, the potential number of inversions increases. This behavior, however, is not preserved for the lowest-priority queue (the far-right peak in the graph) as a result of starvation. Despite having the largest average queue size, this queue drains fewer packets and, as such, the number of inversions it sees decreases.

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

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

4.2.2 Characterizing SP-PIFO design space

We now systematically explore the design space of SP-PIFO along four dimensions: the number of queues, the adaptation strategy when encountering an inversion (in the push-down stage, §4.1), the utilization levels, and the rank distributions.

SP-PIFO manages to approximate the optimal algorithms in all rank distributions and utilization levels, with as little as 8 queues. The best performances are obtained under low utilizations and with 32 queues.

Number of queues (Fig. 5b) When using only 8 queues, SP-PIFO is already within ∼20–29% of the gradient-descent algorithm and the optimal mapping. With 32 queues, it gets even closer, producing only ∼22% more inversions than the optimal and achieving on-par behavior to the gradient-descent algorithm. Overall, it improves FIFO performance ∼3.3× (resp. ∼10×) when only 8 (resp. 32) queues are used.

Push-down strategies (Fig. 5c) We evaluate four adaptation strategies for decreasing each queue bound in the push-down stage: (i) to the value of the next-higher queue bound (“Queue Bound”); (ii) by the cost of the inversion ($q_1 - r(p)$, the strategy in SP-PIFO, “Cost”); (iii) by the rank of the packet causing the inversion (“Rank”); and (iv) by 1 (“1”).

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

Utilization (Fig. 5d) SP-PIFO performance is close to the gradient-based algorithm. For utilizations below 60%, SP-PIFO is on-par with the gradient-based algorithm. The number of inversions slightly increases at higher utilizations: 26% and 38% for 80% and 90%.

Rank distributions (Fig. 6) We analyze the performance of SP-PIFO under four alternative rank distributions: exponential, inverse exponential, Poisson and convex. SP-PIFO performs better than FIFO and is close to the gradient-based algorithm for each distribution.

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

5 Implementation

In this section, we describe our implementation of SP-PIFO in $P4_{16}$ [7] and $P4_{14}$.[2] Our implementation follows the algorithm described in §4 and spans 190 ($P4_{16}$) and 735 ($P4_{14}$) lines of code. It performs three main operations: (i) computing/extracting the rank from a packet header; (ii) mapping packets to queues (§2); and (iii) updating the queue bounds.

Rank computation We implemented and tested multiple rank computation functions such as LSTF [17], STFQ [23], and FIFO+ [9] in $P4_{16}$. We note that the reduced memory usage in SP-PIFO leaves room to compute ranks directly on the switch. That said, most ranking algorithms can directly be computed by the end-hosts [17].

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

Adaptation When the mapping process detects an inversion, we need to update all queue bounds. While accessing multiple registers is not restricted by the P4 specification [10], current architectures do not support it (among others, to guarantee line rate). We address this problem by relying on the packet-resubmission primitive to access the queue bounds a second time and update them with the measured inversion cost. While resubmission can possibly break the line-rate guarantees, we only require it occasionally, upon inversions.

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

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

6 Evaluation

We now evaluate SP-PIFO performance and practicality. We first use packet-level simulations to evaluate how SP-PIFO approximates well-known scheduling objectives under realistic traffic workloads (§6.1). We then evaluate SP-PIFO scheduling performance when deployed on hardware switches (§6.2).

6.1 Performance analysis

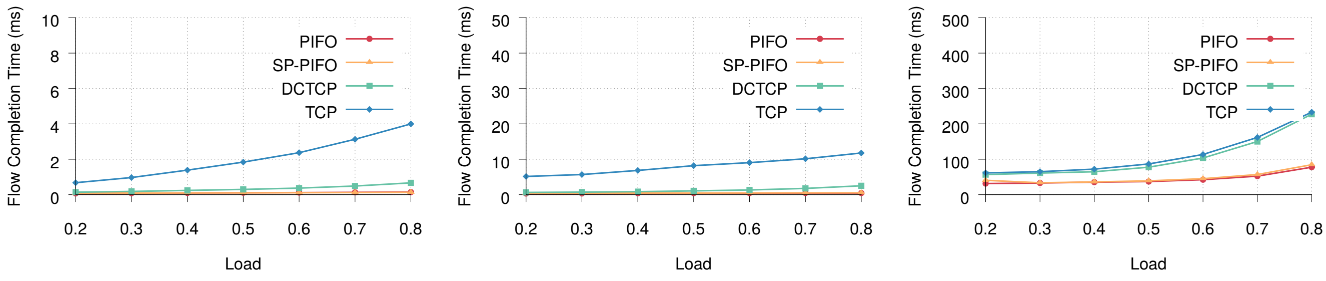
We consider two scheduling objectives: (i) minimizing Flow Completion Times (FCTs); and (ii) enforcing fairness. We consider that ranks are set at the end hosts for the former objective and computed in the switch for the latter. For both objectives, we show that SP-PIFO scheduling capabilities achieve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

Methodology We integrated SP-PIFO in Netbench [3, 15], a packet-level simulator. Similar to [4], we use a leaf-spine topology with 144 servers connected through 9 leaf and 4 spine switches. We set the access and leaf-spine links to 1Gbps and 4Gbps, respectively. This results in a theoretical end-to-end Round-Trip-Time (RTT) of 32.12μs when crossing the spine (4 hops) and 26μs under the leaf (2 hops). We generate traffic flows following two widely-used heavy-tailed workloads: pFabric web application and data mining [4]. Flow arrivals are Poisson-distributed and we adapt their starting rates to achieve different utilization levels. We use ECMP and draw source-destination pairs uniformly at random.

6.1.1 Minimizing Flow Completion Times

Rank definition & benchmarks We minimize FCTs by implementing the pFabric algorithm [4] which sets the packet ranks according to their remaining flow sizes. Specifically, we compare pFabric performance when run on top of PIFO and SP-PIFO. We also analyze TCP NewReno with traditional drop-tail queues and DCTCP with ECN-marking drop-tail

[2] The $P4_{14}$ code is used for running SP-PIFO on the Tofino platform [2].

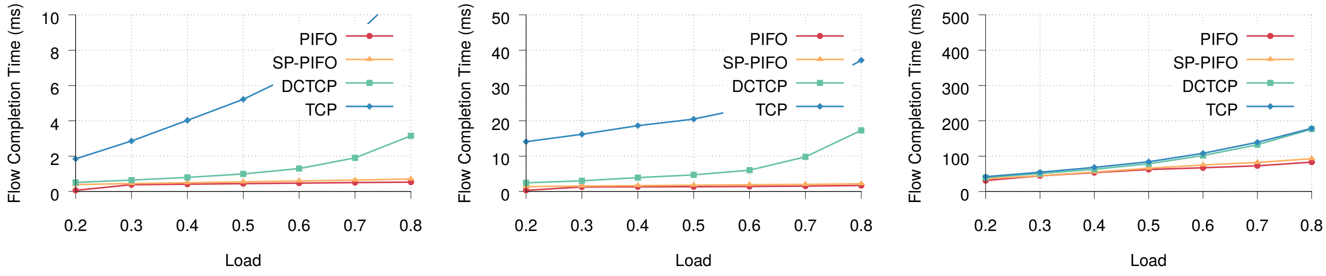


(a) (0,100KB): Average

(b) (0,100KB): 99th percentile

(c) [1MB,∞): Average

Figure 7: pFabric: FCT statistics across different flow sizes in data mining workload.



(a) (0,100KB): Average

(b) (0,100KB): 99th percentile

(c) [1MB,∞): Average

Figure 8: pFabric: FCT statistics across different flow sizes in web search workload.

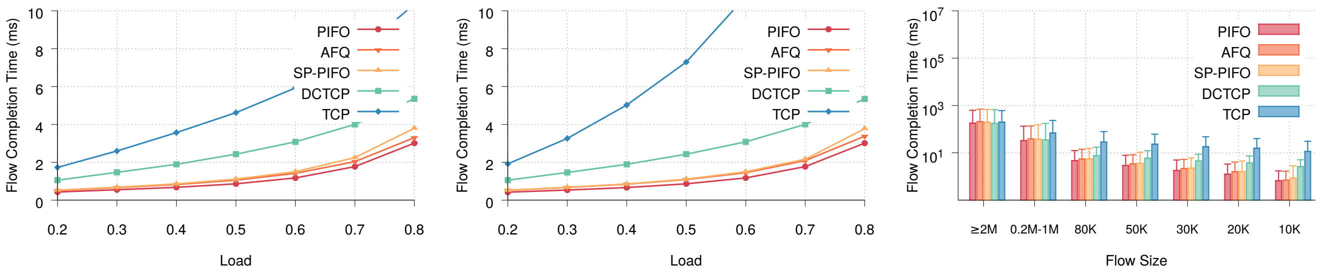
queues. Our pFabric implementation does not consider starvation prevention. As suggested in [4], we approximate pFabric rate control by using standard TCP with a retransmission time-out of 3 RTTs, balancing the difference in RTOs between schemes with the proportional queue size. That is, we use an RTO of 96μs and 8 queues×10 packets for SP-PIFO (resp. 1 queue×80 packets in PIFO), and an RTO of 300μs and 146KB drop-tail queues for both TCP and DCTCP, with ECN marking at 14.6KB, i.e. ∼10 packets.

Summary Fig. 7 and Fig. 8 depict the average and 99th percentile FCTs of large ($\geq$ 1MB) and small flows ($<$ 100KB) for both data mining and web search workloads. We see that SP-PIFO achieves close-to-PIFO performance in both distributions. When comparing performance across flow sizes, we see that SP-PIFO achieves better performance for small flows. This is not surprising since those flows are mapped into higher-priority queues. As discussed in §4.2, strict-priority schemes provide higher unpifoness protection for packets mapped into higher-priority queues.

When comparing the two traffic distributions, we see that SP-PIFO performs better under the data mining workload. This is again expected. While both distributions are heavy-tailed, the data mining one is more skewed [4] and therefore easier to handle for SP-PIFO. Indeed, the probability of having large flows simultaneously sharing the same port (potentially blocking smaller flows) is lower for the data mining workload.

Data mining (Fig. 7) The average FCTs achieved by PIFO and SP-PIFO are similar for small flows, i.e. within ∼0.4–11%. Concretely, SP-PIFO outperforms DCTCP and TCP by a factor of ∼2–5× and ∼8–30×, respectively. When considering the 99th percentile, the gap between PIFO and SP-PIFO slightly accentuates to ∼9.6–26.6%. Still, SP-PIFO outperforms DCTCP and TCP by a factor of ∼1.5–4.7× and ∼12.5–22×, respectively. The largest performance gap between PIFO and SP-PIFO occurs at low utilization. In this regime, the number of packets scheduled is low and the transient adaptation of SP-PIFO is more visible. Whenever the utilization is >40%, the difference is consistently below 20%. Finally, SP-PIFO and PIFO still perform similarly among large flows: within ∼1.9–9%, representing improvements with respect to TCP and DCTCP of ∼1.4–2.7× and ∼1.5–2.8×, respectively.

Web search (Fig. 8) The results are similar to the data mining one, with slightly worse performance for SP-PIFO, especially amongst big flows. Indeed, since the distribution is less skewed, bigger flows have higher chances to reach higher-priority queues, blocking transmissions of smaller flows. Still, we see that the performance of SP-PIFO is within ∼16.54–32.5% of PIFO for small flows, and between ∼1.3–4.4× and ∼4.7–16.7× better than DCTCP and TCP. Even at the 99th percentile, the difference between SP-PIFO and PIFO stays within ∼20.7–32%. Note that, while the percentages might seem high, the values we are looking at are very small.



(a) (0,100KB): Average on 8 queues

(b) (0,100KB): Average on 32 queues

(c) FCT breakdown 70%: Average on 32 queues

Figure 9: Fairness: FCT statistics for all flows at different loads, over the web search workload.

6.1.2 Enforcing fairness across flows

Rank definition & benchmarks We enforce fairness across flows by implementing the Start-Time Fair Queueing (STFQ) rank design [13] on top of PIFO and SP-PIFO. We benchmark our solution with AFQ [21] (§8). We analyze the performance for different flow sizes and number of queues. Specifically, we use 8 queues×10 packets in SP-schemes (resp. 1 queue×80 packets for single-queue schemes) and 32 queues×10 packets in SP-schemes (resp. 1 queue×320 packets for single-queue schemes). For AFQ, we select the bytes-per-round parameter which gives the best performance. In our testbed, this is 320 and 80 BpR for the 8-queue and 32-queue scenario, respectively. As in [21], we use DCTCP as transport layer for AFQ, PIFO and SP-PIFO (with an RTO of 300μs). We set ECN marking to 48KB, i.e. ∼32 packets. We generate traffic following the pFabric web search distribution.

Summary Fig. 9a and Fig. 9b depict the average FCTs of small flows across different levels of utilization, when 8 queues and 32 queues are used. Fig. 9c depicts the FCTs across flow sizes at 70% utilization and for 32 queues. In all cases SP-PIFO achieves near-PIFO behavior and is on-par performance with AFQ (current state-of-the-art).

Impact of the utilization (Fig. 9a & Fig. 9b) SP-PIFO stays within ∼23–28% (resp. ∼21–28%) of ideal PIFO across all levels of utilization when 8 queues (resp. 32) are used. Even in the highest utilizations, it is consistently below ∼26% (resp. ∼25%). SP-PIFO performance is at the level of AFQ, within ∼3–10% (resp. ∼0.5–11%), generating improvements of ∼1.4–2.3× and ∼2.7–4.2× (resp. ∼1.4–2.3× and ∼3.7–7.4×) over DCTCP and TCP. The fact that SP-PIFO performance is equivalent with 8 and 32 queues is not surprising: as the bandwidth-delay product is low, only a reduced queue size is required for efficient link utilization.

Impact of flow sizes (Fig. 9c) At 70% utilization, we see that SP-PIFO lies within ∼10–30% of PIFO performance for all flow sizes and is on-par with AFQ. The only exception is for very small flows (<10K) in which AFQ performs 20% better. SP-PIFO improves DCTCP and TCP behaviors for small flows, within ∼1.5–3X and ∼2–13X, respectively. Considering the 99th percentile, we see that SP-PIFO stays within ∼8–35% of PIFO and improves between ∼12–78% and ∼1.5–10.76× with respect to DCTCP and TCP.

Impact of the number of queues (Fig. 10) We analyze the impact of the number of queues on average FCTs for both AFQ and SP-PIFO. We set the BpR at MSS for all queue configurations, as in [21], avoiding AFQ dropping packets too often for cases of fewer queues. We see that while AFQ has a higher sensitivity with respect to the number of queues, SP-PIFO preserves a similar level of performance, without any configuration or prior traffic knowledge.

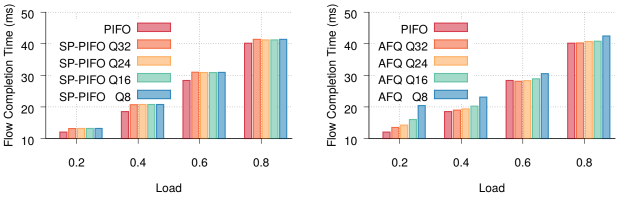
6.2 Hardware testbed

We finally evaluate our hardware-based implementation of SP-PIFO on the Barefoot Tofino Wedge 100BF-32X platform [2]. We perform two experiments. First, we analyze the bandwidth allocated by SP-PIFO to flows with different ranks when scheduled over a bottleneck link. Second, we measure the impact on the FCT when SP-PIFO runs pFabric. We show that SP-PIFO efficiently schedules traffic at potentially Tbps.

Bandwidth shares We transmit 8 UDP flows of 20Gbps between two servers. We generate the flows progressively, in increasing order of priority (decreasing rank). We use 4 priority queues and schedule the flows over a 10Gbps interface. We generate the flows using Moongen [12] and use an intermediate switch to amplify them to the required throughput.

Fig. 11 depicts the flows' bandwidth and how SP-PIFO manages to virtually extend the number of queues. As expected, the first 4 flows receive the complete bandwidth, since they are mapped to dedicated queues. As the number of flows exceeds the number of queues, flows start to share queue space and see a reduced bandwidth.

Flow completion times We simultaneously generate 1000 TCP flows of different sizes, going from 1GB to 100GB in steps of 100MB, and schedule them over a bottleneck link of 7Gbps. We set the rank of each flow to the absolute flow size,



(a) All: Average

(b) All: Average

Figure 10: Fairness: FCT statistics for all flows at different loads, when the number of queues is modified.

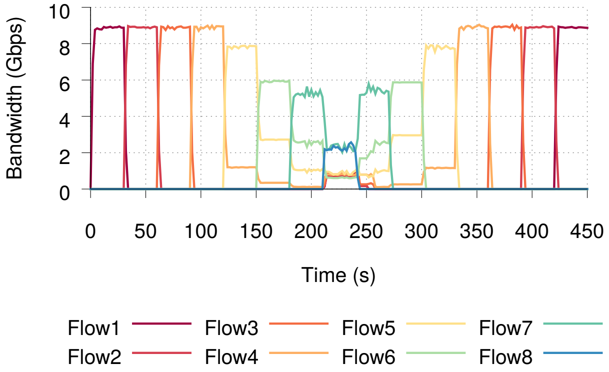


Figure 11: Tofino: Bandwidth allocation under progressive flow generation with increasing priorities.

following [4]. We compare the FCTs achieved by SP-PIFO scheduling and the ones achieved by a FIFO queue.

Fig. 12 shows the resulting FCTs. As expected, the FIFO queue leads to increased FCTs by not considering flow size. In contrast, SP-PIFO prioritizes short flows over long ones, minimizing their FCTs and the overall transmission time.

7 Discussion

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

PIFO-inherited limitations Individual PIFO queues suffer from two main limitations. First, they cannot rate-limit their egress throughput preventing them from implementing non-work-conserving scheduling algorithms. SP-PIFO also shares the same limitation. Second, PIFO queues cannot directly implement hierarchical scheduling algorithms. Yet, as proposed by [23], multiple SP-PIFO schemes (i.e., using different set of priority queues) can be grouped as a tree to approximate hierarchical scheduling algorithms. The key challenge consists in figuring out how to allow access of multiple queues with existing traffic manager capabilities. While this is orthogonal to this paper, one option would be to recirculate packets, enabling access to the traffic manager (and therefore the queues) multiple times in the same pipeline. Doing so, while limiting the impact on performance, is an interesting open question.

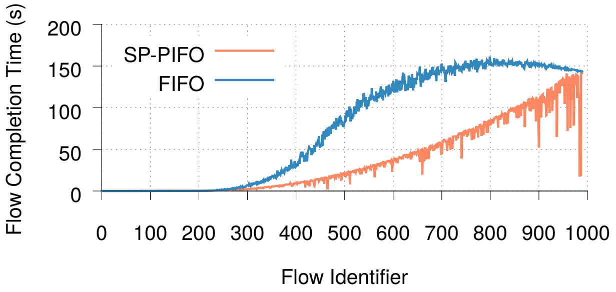


Figure 12: Tofino: FCT statistics across different flow sizes with pFabric ranks.

SP-PIFO-specific limitations The main limitation of SP-PIFO is that, as an approximation scheme, it cannot guarantee to perfectly emulate the behavior of a theoretical PIFO queue for all ranks. We note two things. First, our evaluation (§6) shows that, for realistic workloads, SP-PIFO performance is often on-par with PIFO performances. Second, we note that SP-PIFO *can* provide strong PIFO-like guarantees *for some ranks* by dedicating some queues to them at the price of reduced performance for the other ranks. We believe this is an interesting tradeoff as current switches can support up to 32 queues per port [21].

Adversarial workloads We have argued that, on average, SP-PIFO can adapt to any kind of rank distribution. This has certain limitations. First, we assume that all queues are drained at some point. Nonetheless, a malicious host could send a large number of high-priority packets and, as a result, packets in lower-priority queues would never be drained. Such "starvation" attacks are common to any type of priority scheme. For instance, a malicious host could try to grab a bigger slice of the network resources by setting ranks to 0 in slack-based algorithms [4,9,17] or resetting flow identifiers in fair-queuing schemes [23]. The problem of starvation in strict-priority scheduling is also well-known in the context of QoS and is typically addressed by policing high-priority traffic at the edge of the network [18].

Aside from starvation attacks we also assume that, for a given rank distribution, the particular order of ranks is random. In practice, this is reasonable. While the ranks for individual flows might have some structure (e.g., monotonically-increasing ranks), when various flows are scheduled together the ordering of their packet ranks is randomized. Yet, attackers could try to coordinate large numbers of flows to create adversarial orderings, which "outplay" the adaptation mechanisms (§B.3). Nevertheless, any non-malicious flow which is active at the same time can thwart such strategies by randomly breaking the adversarial order. Aside from that, the network could be monitored to detect such adversarial attacks.

Facilitating PIFO in the future On a forward-looking perspective, we note some improvements in hardware primitives that would facilitate PIFO implementations in the future. As we already discussed in §5, a higher number of stages would facilitate per-queue state storage and a higher number of queues would directly increase PIFO performance. Further than that, multiple and dynamic memory access between the ingress and egress pipelines would allow state updates after inversions in the highest-priority queue without having to rely on resubmission techniques. In the same direction, access to queue information from the ingress pipeline or an enhanced flexibility in the management of strict-priority queues directly from the data plane would enable more accurate unpifoness prediction at enqueue, opening the doors to higher-performance SP-PIFO algorithms.

8 Related work

Programmable packet scheduling While scheduling has been extensively studied over the years, the idea of making it programmable is relatively recent [17, 22]. In [24], Sivaraman et. al. suggested programmable scheduling by proving that the best scheduling algorithm to use depends on the desired performance objective. In [17], Mittal et. al. made the observation that, despite certain algorithms accept configurations to approximate a wide range of objectives, a universal packet scheduling outperforming in all scenarios does not exist.

Several abstractions for programmable scheduling have been proposed afterwards. In addition to PIFO [24], Eiffel [19] presents an alternative queue structure which approximates fine priorities by exploiting the characteristics that define packet ranks in most scenarios to diminish the required computational complexity. In contrast to [19, 24], which rely on new hardware designs, SP-PIFO shows that efficient programmable packet scheduling can be achieved today, at scale, and on existing devices.

Exploiting priority queues Other (recent) schemes leverage multiple priority queues for specific performance objectives. They highlight the need of programmable scheduling in existing devices [16], and illustrate how rank designs producing close-to-optimal results can already be implemented in existing data planes. For enforcing fairness, FDPA [8] simplifies the computational cost of per-flow virtual counters or individual user queues in traditional-fair-queuing schemes by using arrival-rate information at a user level. AFQ [21], instead, emulates ideal fair queuing by implementing per-flow counters on a count-min sketch and dynamically rotating priorities in a strict-priority scheme to imitate the round-robin behavior. SP-PIFO differs by fixing queue priorities and dynamically adapting the mapping of packets to those queues. This actually makes SP-PIFO implementable *at line rate* in existing data planes.

pFabric [4] and PIAS [5] show the use of priority queues in flow completion time minimization. While pFabric relies in general on a PIFO-queue design, [4] includes experiments in which flows are mapped to priority queues based on their size. While pFabric experiments use thresholds fixed from the knowledge of flow distributions, SP-PIFO adapts the mapping design automatically per-packet, without any traffic knowledge required in advance. PIAS [5] approaches the case of unknown flow sizes and uses Multi-level Feedback Queues (MLFQ) [11] to achieve the desired Shortest Job First (SJF) behavior, by gradually switching flows from higher to lower-priority queues as their number of transmitted bytes increase.

In contrast to these proposals, SP-PIFO supports a much wider range of performance objectives. SP-PIFO (like PIFO [24]) can be used to implement *any* scheduling algorithm in which the relative scheduling order does not change with future packet arrivals. As illustrated in the evaluation section (§6), the algorithms presented in AFQ [21], FDPA [8], pFabric [4] and PIAS [5] can be used as ranking designs (i.e., setting packet ranks to scheduling virtual rounds, estimated arrival rates, shortest remaining processing time of flows, or number of packets transmitted under the MLFQ aging design) to be run on top of SP-PIFO.

9 Conclusions

We presented SP-PIFO, a programmable packet scheduler which closely approximates the theoretical behavior of PIFO queues, today, on programmable data planes. The key insight behind SP-PIFO is to dynamically adapt the mapping between the packet ranks and a (fixed) set of strict-priority queues.

Our evaluation on realistic workloads shows that SP-PIFO is practical: it closely approximates PIFO behaviors and, in many cases, perfectly matches them. We also confirm that SP-PIFO runs on actual programmable hardware.

Overall, we believe that our work shows that the benefits of programmable packet scheduling—experimenting with new scheduling algorithms—can be fulfilled today, in existing networks.

Acknowledgments

We are grateful to the NSDI reviewers and our shepherd, Anirudh Sivaraman, for their insightful comments. We also thank the members of the Networked Systems Group at ETH Zürich (especially Edgar Costa Molero) together with Changhoon Kim from Barefoot for their valuable feedback.

References

[1] Broadcom Trident II. https://www.broadcom.com/products/Switching/DataCenter/BCM56850-Series, 2016.

[2] Barefoot Tofino. http://barefootnetworks.com/products/brief-tofino/, 2017.

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

[4] 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.

[5] 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.

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

[7] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming Protocol-independent Packet Processors. 2014.

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

[9] 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.

[10] The P4 Language Consortium. P4-16 Language Specification, version 1.1.0-rc. https://p4.org/p4-spec/docs/P4-16-v1.1.0-draft.pdf, 2018.

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

[12] 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.

[13] 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.

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

[15] 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.

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

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

[18] Juniper Networks. Class of Service Feature Guide for Security Devices. page 115, 2018.

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

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

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

[22] 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.

[23] 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.

[24] 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.

A Gradient-based algorithm

In this appendix we detail the greedy iterative algorithm presented in §3.2. We first motivate and proof how the algorithm converges to the optimal solution (A.1). Second, we show how to effectively prune the search space making computation efficient while keeping convergence (A.2). Finally, we analyze its implementation (A.3) and convergence requirements (A.4).

A.1 Greedy optimization

The algorithm (alg. 2) iteratively minimizes the risk by adjusting queue bounds, one queue and one step at a time, until reaching convergence. At each iteration, the algorithm predicts, for every q_i, whether moving the bound by one (in either direction) decreases the expected risk, and moves the bound in the direction of maximum decrease. In the following, we discuss first, how the algorithm can predict the expected change in risk, and second, why checking a single step is sufficient to converge.

Algorithm 2 Greedy optimization

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

Risk difference In §3.2, we demonstrated that the risk can be analyzed on a per-queue basis from the cost of mapping packets with different ranks to the same queue. Consequently, changes in the risk resulting from changing the bound vector $\boldsymbol{q}$ can be analyzed by comparing the risk difference in affected queues. To be precise, every change of a single element q_i in $\boldsymbol{q}$ affects two queues, queue i and $i-1$, as ranks are either moved from i to $i-1$ (increase in q_i) or moved from $i-1$ to i (decrease in q_i).

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

$$\Delta_i^+ = p(r^*)\Big(\sum_{r\in Q_{i-1}} p(r)cost(r^*,r) - \sum_{r\in Q_i} p(r)cost(r,r^*)\Big) \tag{9}$$

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

$$\Delta_i^- = p(r^*)\Big(\sum_{r\in Q_i} p(r)cost(r^*,r) - \sum_{r\in Q_{i-1}} p(r)cost(r,r^*)\Big) \tag{10}$$

Proof Increasing q_i effectively removes the lowest rank from queue i, which now becomes the highest rank in queue $i-1$. As the new highest rank in queue $i-1$, it causes possible inversions and therefore risk for *all* other ranks in queue $i-1$, resulting in the first, positive term in eq. 9. Conversely, as the lowest rank in queue, it was prone to receive inversions from any other element in the queue, supposing a risk in queue i that is removed with the change. This risk reduction results in the second, negative, term.

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

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

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

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

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

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

Theorem 2 *Let Δ_i^+ and Δ_i^- denote the prospective in- and decreases from incrementing/decrementing q_i by 1. Let Δ_i^{++} and Δ_i^{--} denote the in- and decreases from incrementing/decrementing q_i by more than 1. Let the cost function used to compute the differences be non-decreasing in $|r^* - r|$ and 0 if and only if $r^* = r$. Then:*

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

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

Proof

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

$$\sum_{r \in Q_{i-1}} p(r)\text{cost}(r^*, r) > \sum_{r \in Q_i} p(r)\text{cost}(r, r^*) \tag{11}$$

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

$$\Delta_i^{++} = \tag{12}$$

$$p(r^*)(\sum_{r \in Q_{i-1}} p(r)\text{cost}(r^*, r) - \sum_{r \in Q_i} p(r)\text{cost}(r, r^*)) + \tag{13}$$

$$p(r^{**})(\sum_{r \in Q_{i-1}} p(r)\text{cost}(r^{**}, r) - \sum_{r \in Q_i} p(r)\text{cost}(r, r^{**})) \tag{14}$$

Note that we can omit the cost between r^* and r^{**} in eq. 14: as the cost function is by definition symmetric, the additional increase in the left-hand term is exactly equal in magnitude to the additional decrease in the right-hand term, and thus they cancel each other. Thus we omit the term to not clutter the notation. Next, again by definition of the cost function, if $r^{**} > r^* > r$, then $cost(r^{**}, r) \geq cost(r^*, r)$, and if $r > r^{**} > r^*$, then $cost(r, r^{**}) \leq cost(r, r^*)$. Additionally, we note that the order of arguments in the cost function does not matter, as it is symmetrical. Applied to the risk of the lower- and higher-priority queue respectively (eq. 14), this gives:

$$\begin{aligned} \sum_{r \in Q_{i-1}} p(r)cost(r^{**}, r) &\geq \sum_{r \in Q_{i-1}} p(r)cost(r^*, r) \\ \sum_{r \in Q_i} p(r)cost(r, r^{**}) &\leq \sum_{r \in Q_i} p(r)cost(r, r^*) \end{aligned} \tag{15}$$

And in conclusion, the left hand term in eq. 14 is larger than the left hand term in eq. 13, and the right hand term in eq. 14 is smaller then the left hand term in eq. 13. Consequently, if eq. 13 is positive, eq. 14 must also be positive (as probabilities are always positive), proving that if one step does increase the risks, two steps will also increase the risk. The exact same procedure can be repeated for larger step sizes, which we omit here.

2: This proof is conceptually identical to the other direction, and we will thus omit it. The guiding principle is the same: moving more than one rank can only cause higher increase in risk in the queue the ranks are moved to, and lower decrease in risk in the queue the ranks are taken from, compared to the previous ranks. Thus, if already moving one rank causes a higher increase in risk in one queue than decrease in the other, moving additional ranks does not change this.

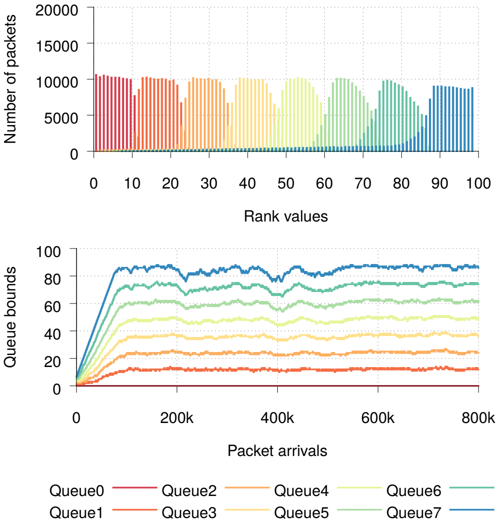


Figure 13: Greedy convergence for uniform rank distribution.

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

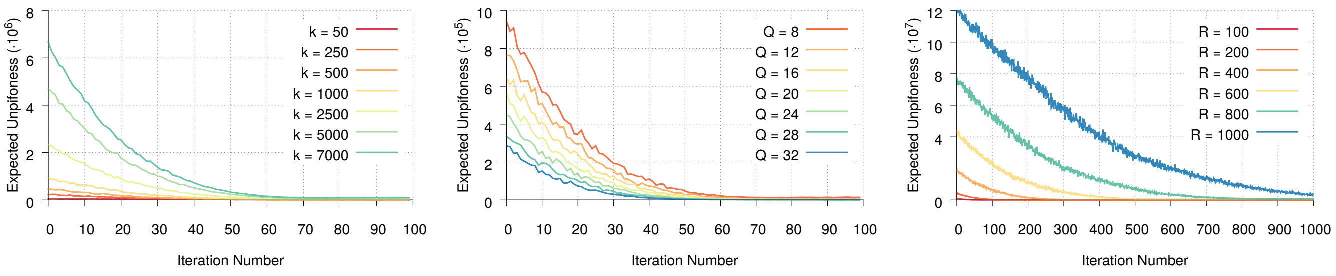
A.2 Efficient computation

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

From the empirical probability definition, $p_{\mathcal{D}}(r) = |r_{\mathcal{D}}|/|\mathcal{D}|$, we can rewrite eq. 9 and eq. 10 as:

$$\begin{aligned} \Delta_i^+ &= \frac{|q_i|}{|\mathcal{D}|^2} \cdot (\sum_{r \in Q_{i-1}} |r|\text{cost}(q_i, r) - \sum_{r \in Q_i} |r|\text{cost}(r, q_i)) \\ \Delta_i^- &= \frac{|q_i - 1|}{|\mathcal{D}|^2} \cdot (\sum_{r \in Q_i} |r|\text{cost}(q_i - 1, r) - \sum_{r \in Q_{i-1}} |r|\text{cost}(r, q_i - 1)) \end{aligned} \tag{16}$$

Since the queue bound q_i stays constant throughout the adaptation window, each of the summations in eq. 16 can be implemented through a counter which gets updated every time a new packet arrives, with its carried rank. Note that the number of counters required increases linearly with the number of queues. Also, observe that the counters in eq. 16, only allow the computation of one step in the gradient. However, this is enough since, as can be seen in Fig. 13, the one-step version manages to converge in practice.



(a) Convergence vs. adaptation window

(b) Convergence vs. number of queues

(c) Convergence vs. rank range

Figure 14: Greedy algorithm adaptation microbenchmark.

A.3 Implementation requirements

With the computation presented in A.2, implementing the gradient-based algorithm on top of n priority queues, requires n registers for queue-bound storage and $(4 \cdot n)$ registers for the gradient computation. The mapping process §2 requires packets to potentially read all the queue-bound values (i.e., for packets scheduled in the highest-priority queue). In the same direction, while most packets only need to update the two counters corresponding to their queue, the k_{th} packet in each sequence needs to access *all* counters to perform the adaptation decision. This supposes being able to read $n + (4 \cdot n)$ different registers for a single packet (without even considering the updates). Since existing devices only support up to 12-16 stages, with a single register access per stage [14], the implementation of the greedy algorithm is not feasible for a practical number of queues (i.e., $n \geq 8$).

A.4 Convergence analysis

We now show how the greedy-algorithm performance varies when modifying the three main degrees of freedom: (i) the adaptation window (i.e., the number of packets that are monitored before the adaptation mechanism is executed); (ii) the number of queues available in the strict-priority scheme; and (iii) the number of ranks in the distribution. For that, we analyze the unpifoness evolution of a single switch running the greedy algorithm for a uniform rank distribution from 0 to 100 until convergence. We compute unpifoness as specified in §3.1, based on the packets scheduled and the queue bounds used during the adaptation window.

Effects of varying the adaptation window Fig. 14a shows the unpifoness evolution when we run the greedy algorithm on top of a strict-priority scheme of 8 queues, and we vary the adaptation window from 50 to 7000 packets. We observe that, for the algorithm to converge, the adaptation window needs to be broad enough to cover a *complete* sample of the rank distribution (i.e., one that characterizes all its representative behaviors). In our case, any adaptation window below 100 packets can not characterize completely the rank distribution. Indeed, Fig. 14a depicts how the greedy algorithm correctly converges as soon as more than 200 packets are monitored per iteration. In general, the broader the adaptation window, the more precise the rank distribution estimate, and the better the adaptation decision. However, while a too narrow adaptation window can suppose missing important information of the rank distribution and breaking convergence guarantees, a too broad adaptation window can make the algorithm too slow to converge, negatively impacting the performance.

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

Effects of varying the number of queues Fig. 14b depicts the case in which we fix an adaptation window of 1000 packets, and modify the number of queues from 8 to 32. All queues have a constant size of 10 packets. We see how the higher number of queues the lower the unpifoness, and the better the PIFO approximation. This is expected since each queue can be perceived as an opportunity to sort packets with different ranks, and therefore to reduce the number of inversions. Also, we can see how the number of iterations required by the algorithm to converge does not directly depend on the number of queues. This results from the fact that each adaptation decision analyzes (and, if required, updates) potential redesigns for *all* the different queue bounds.

Effects of varying the number of ranks Fig. 14c presents the effects of modifying the range of the uniform rank distribution from 100 to 1000 ranks, when we fix the number of queues to 8 and the adaptation window to 1000 packets. As expected, under the same number of queues, a higher number of ranks implies an increase in unpifoness. Also, as the rank ranges get closer to the adaptation window, the distribution estimates get worse, and the adaptation gets tougher.

B Theoretical analysis of SP-PIFO

SP-PIFO is a highly-dynamic probabilistic system. In particular, its queue bounds $\boldsymbol{q}$ change with nearly every incoming packet. Nevertheless, in this section we show that the system has an attractive equilibrium $\boldsymbol{q}^*$ (B.1), how this equilibrium balances the different causes of inversions (B.2), and we discuss the limitations and open question of our analysis (B.3).

B.1 Stable equilibrium

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

$$\boldsymbol{q}^{t+1} = \boldsymbol{q}^t + \Delta(r^t) \tag{17}$$

where r^t is the rank of the t-th packet, and $\Delta(r^t)$ is the change this packet causes on the queue bounds. The queue-bound change is given by the "push-down" and "push-up" stages of SP-PIFO, respectively. If the packet causes an inversion in the highest-priority queue, all queue bounds are *decreased* by $q_1^t - r^t$. Otherwise, there is exactly one queue i such that $q_i^t \leq r^t < q_{i+1}^t$, and only q_i is set to r^t, or equivalently, is *increased* by $r^t - q_i^t$. Finally, let $p(r^t)$ be the probability of rank r for the t-th packet. Then, the expected value of the queue bounds at step $t+1$, and the expected difference to the queue bounds at step t are, respectively: [3]

$$\mathrm{E}\left[q_i^{t+1}\right] = \mathrm{E}\left[q_i^t\right] \tag{18}$$

$$+ \underbrace{\sum_{q_i^t \leq r^t < q_{i+1}^t} p(r^t)(r^t - q_i^t)}_{\Delta_i^+(\boldsymbol{q}^t, r^t)} \tag{19}$$

$$- \underbrace{\sum_{r^t < q_1^t} p(r^t)(q_1^t - r^t)}_{\Delta^-(\boldsymbol{q}^t, r^t)} \tag{20}$$

$$\Leftrightarrow \mathrm{E}\left[q_i^{t+1} - q_i^t\right] = \Delta_i^+(\boldsymbol{q}^t, r^t) - \Delta^-(\boldsymbol{q}^t, r^t) \tag{21}$$

Equilibrium As expected, we can see from eq. 21 that the change of queue bounds is determined by the "push-up" (Δ_i^+) and "push-down" (Δ^-) stages working against each other. Indeed, if Δ_i^+ is larger than Δ^-, the queue bound increases, and vice versa. The system has an equilibrium $\boldsymbol{q}^*$, where $\Delta_i^+ = \Delta^-$ and the expected change is 0. Note that this equilibrium depends on the rank probability.

Attraction The equilibrium $\boldsymbol{q}^*$ is attractive, i.e. if $q_i^t < q_i^*$, $\mathrm{E}[q_i^{t+1} - q_i^t] > 0$, and vice versa. For small perturbations, this is straightforward. Assume that all queue bounds are in equilibrium, except q_i. If $q_i^t < q_i^*$, then $\Delta_i^+(\boldsymbol{q}^t, r^t) > \Delta_i^+(\boldsymbol{q}^*, r^t)$, because the sum in eq. 19 has (i) more (non-negative) terms; and (ii) each term is weighted stronger, as the difference $r^t - q_i^t$ is larger. On the other hand, $\Delta^-(\boldsymbol{q}^t, r^t)$ is either equal to $\Delta^-(\boldsymbol{q}^*, r^t)$ (for $i > 1$) or even smaller (for $i = 1$, as there are less, and lesser weighted, terms in the sum 20). Thus, the increase is larger than the decrease, and the expected change to q_i is positive. The argument for $q_i^t > q_i^*$ is symmetrical.

For larger disturbances, the equilibrium is also attractive, but it might take more than a single time step, as the "push-up" stage for q_i also depends on q_{i+1}: if both $q_i < q_i^*$ and $q_{i+1} < q_{i+1}^*$, the "push-up" might be too weak to pull q_i towards the equilibrium. However, this is not the case for the lowest-priority queue q_n, for which the "push-up" does not depend on another queue. Thus, lower-priority queues (at least q_n) might be pulled towards the equilibrium at first, while other q_i are not. Notice that an expected increase of q_{i+1}^t increases the "push-up" mechanism for q_i^{t+1} and decreases it for q_{i+1}^{t+1} (eq. 19). Eventually, as the lower-priority queue bound is getting closer to the equilibrium, the higher-priority queue bound is also pulled towards the equilibrium. This continues until the highest-priority queue, where an expected increase of q_1^t also increases the "push-down" mechanism for all bounds at step $t+1$ (eq. 20). As a result, over multiple time steps, the expected effects of the "push-up" and "push-down" stages equalize, eventually pulling all q_i towards q_i^*.

B.2 Balance

As explained in §4, there are three main reasons for unpifoness: (i) inversions in the highest-priority queue, after which all queue bounds are decreased; (ii) inversions in a lower-priority queue after its queue bound has been decreased; (iii) inversions in a lower-priority queue, if its highest rank "overtakes" the lowest rank of a higher-priority queue.

As we can see in eq. 19, eq. 20, and eq. 21, all these factors play a role in the dynamics of SP-PIFO. At the equilibrium, the probability of "push-down", which is exactly the probability of an inversion in the highest-priority queue (weighted by its severity), is equalized with the probability of a packet being mapped to any other queue (again weighted, more on this below). While this does not directly correspond to inversions, the more packets are mapped to lower-priority queues, the higher is the probability of an inversion in those queues after a "push-down". SP-PIFO thus keeps a balance between inversions (i) and (ii), as decreasing (i) would require a stronger "push-down", which would then increase (ii), and vice versa.

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

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

B.3 Assumptions and limitations

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

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

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

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

Adversarial workloads Based on the previous assumptions, we have shown that SP-PIFO is attracted towards an expected equilibrium, in which the different sources of unpifoness are balanced. However, there are also some limitations.

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

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