Programmable Packet Scheduling with a Single Queue

Zhuolong Yu
Johns Hopkins University

Chuheng Hu
Johns Hopkins University

Jingfeng Wu
Johns Hopkins University

Xiao Sun
Stony Brook University

Vladimir Braverman
Johns Hopkins University

Mosharaf Chowdhury
University of Michigan

Zhenhua Liu
Stony Brook University

Xin Jin
Peking University

ABSTRACT

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

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

CCS CONCEPTS

• **Networks** → **Programmable networks**; **Packet scheduling**; **In-network processing**; *Data center networks.*

ACM Reference Format:
Zhuolong Yu, Chuheng Hu, Jingfeng Wu, Xiao Sun, Vladimir Braverman, Mosharaf Chowdhury, Zhenhua Liu, and Xin Jin. 2021. Programmable Packet Scheduling with a Single Queue. In *ACM SIGCOMM 2021 Conference (SIGCOMM '21), August 23–28, 2021, Virtual Event, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3452296.3472887

SIGCOMM '21, August 23–28, 2021, Virtual Event, Netherlands

ACM ISBN 978-1-4503-8383-7/21/08…$15.00
https://doi.org/10.1145/3452296.3472887

1 INTRODUCTION

Packet scheduling is a central research topic in computer networking. Over the past several decades, a great many packet scheduling algorithms have been designed to provide different properties and optimize diverse objectives [6, 11, 23, 40, 41]. Unfortunately, most of these algorithms, despite many novel ideas among them, never have found their way to impact the real world. This is largely due to the high cost to design and deploy switch ASICs to implement them, since packet scheduling algorithms must run in the data plane at line rate in order to process every single packet.

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

A Push-In First-Out (PIFO) queue is a popular abstraction for programmable packet scheduling [3, 47]. PIFO associates a rank with each packet and maintains a sorted queue to buffer packets. Newly arrived packets are inserted into the queue based on their ranks, and packets are dequeued from the head. Different packet scheduling algorithms can be implemented on top of PIFO by changing the rank computation function. Prior works have shown that PIFO can support a wide range of popular scheduling algorithms, such as Shortest Remaining Processing Time (SRPT) [41] for minimizing flow completion times (FCTs) and Start-Time Fair Queueing (STFQ) [13] for weighted fairness.

PIFO, while elegant in theory, is challenging to implement in practice. A recent work [47] proposes a hardware design to support PIFO at a clock frequency of 1 GHz on shared-memory switches. The major design complexity lies in supporting a sorted queue at 1 GHz. Yet, there is a gap from the design to a real switch ASIC implementation, and the design has scalability limitations—it can only support a few thousand flows. SP-PIFO [3] is an approximation of PIFO that can run on existing hardware. The basic idea is to map the possibly large number of ranks into a small set of priorities, and then simply schedule the small number of queues based on

their priorities. This solution, however, requires multiple precious strict-priority queues.

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

AIFO is motivated by the confluence of two recent trends in datacenter networking: shallow buffers in the switches [5] and fast-converging congestion control protocols implemented in end hosts [22]. Together, they significantly reduce the queueing latency inside the network, which is especially important for datacenter environments where low latency is critical for real-time online services with strict Service Level Objectives (SLOs) [5]. Given these trends, we observe that the decisive factor in modern datacenters is often *which* packets are enqueued or dropped by the switch, not the *ordering* in which they leave the switch. For example, dropping packets of an elephant flow when it competes with two mice flows is more important to the flow completion times of the mice flows than the ordering that their packets are dequeued, especially when the queue length is kept small such that only *a few* packets occupy it at any moment.

Based on this insight, the major technical challenge we tackle in this paper is *finding the right set of packets to admit into the queue.* Ideally, AIFO should admit the same set of packets as PIFO to closely approximate it. AIFO addresses this challenge by maintaining a sliding window to track the ranks of recent packets in the window and computing the relative rank of an arriving packet in order to decide whether to admit or proactively drop it even when the queue may still have room! Unlike traditional active queue management (AQM) solutions, AIFO drops packets based on their relative ranks instead of using threshold comparisons against average queue length [12, 34, 35] or delay estimations [32]. Theoretically, we prove that AIFO provides performance close to that of PIFO. We complement it with a concrete data plane design and implementation to show how to efficiently realize AIFO on Barefoot Tofino.

AIFO explores an interesting design question: what are the minimal hardware requirements for programmable packet scheduling? AIFO is an extreme point in the design space—it only requires a single FIFO queue. This is not only theoretically interesting, but also has important practical implications. Our conversations with industry collaborators, including a large-scale search engine and a large-scale e-commerce service, indicate that physical queues are critical resources, and are reserved to ensure strong physical isolation and differentiation between applications of multiple tenants; modern datacenters are already short of physical queues available in switches. Unlike SP-PIFO which requires multiple physical queues for packet scheduling, AIFO enables operators to continue using physical queues for strong physical isolation and differentiation between tenants, and additionally use AIFO to program the packet scheduling algorithm for intra-tenant traffic (e.g., SRPT to minimize the flow completion time).

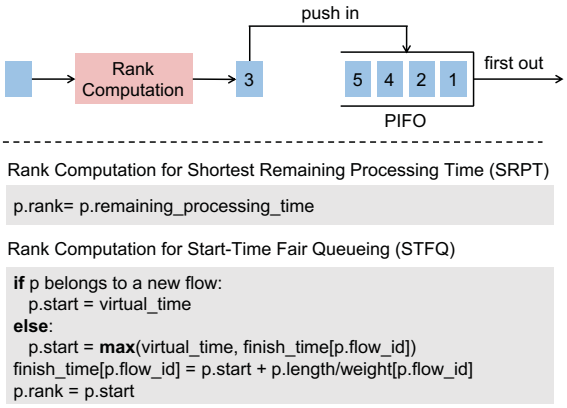


Figure 1: Background on programmable packet scheduling with PIFO.

As an unexpected positive byproduct, AIFO naturally supports *starvation prevention* by design, a necessary feature of pFabric [6] that schedules the packets of the same flow in FIFO to prevent packet reordering. While PIFO supports a wide variety of scheduling algorithms, it cannot support starvation prevention needed by pFabric because the latter packets of a flow would be scheduled first when PIFO is programmed to use SRPT. With the strong demand on minimizing FCTs for low-latency online services, pFabric is arguably the killer application of programmable packet scheduling, as pFabric is considered to be one of the best solutions for minimizing FCTs. AIFO enables us to implement and deploy pFabric on existing hardware.

In summary, we make the following contributions.

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

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

2 BACKGROUND AND MOTIVATION

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

2.1 Programmable Packet Scheduling

Programmable packet scheduling enables the packet scheduling algorithm in a switch to be changed without the need to change the switch ASIC. PIFO [47] is a proposal for programmable packet scheduling. It contains two components: a PIFO queue and a rank computation component. Each packet is associated with a *rank*. The

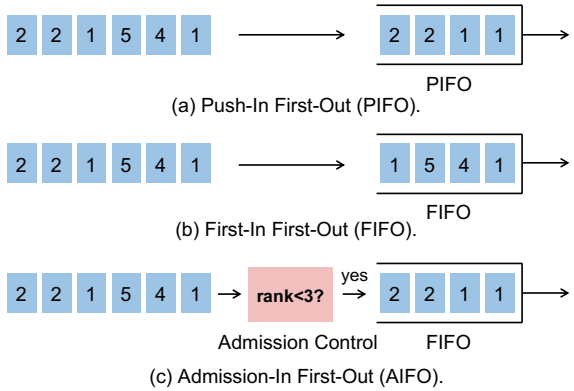


Figure 2: Motivating example of AIFO.

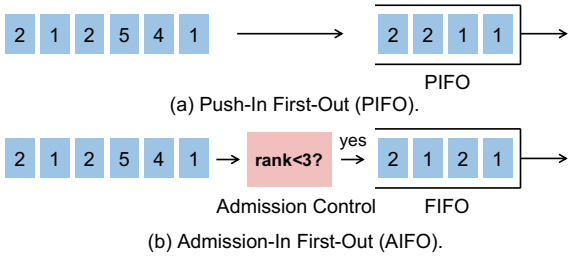


Figure 3: An example that PIFO and AIFO dequeue the same set of packets ({1, 1, 2, 2}), but the dequeueing orderings are different ([1, 1, 2, 2] vs. [1, 2, 1, 2]).

PIFO queue is a priority queue that sorts packets based on their ranks. Packets are inserted into the queue based on their ranks, and are dequeued from the head (i.e., the smallest rank).

Programmability lies in the rank computation component. Programming a packet scheduling algorithm in the context of PIFO refers to programming how a rank for each packet is computed. One simple example is to program SRPT [41] for minimizing FCTs, as shown in Figure 1. In this example, the rank of a packet is simply the remaining processing time of the flow (or simply the remaining bytes of the flow). Note that SRPT requires end hosts to put the remaining processing time in an appropriate field in the packet header [6, 41], which is orthogonal to packet scheduling in the switch. Given such rank computation, the PIFO queue would schedule the packet with the shortest remaining processing time first, i.e., realizing SRPT.

A more complicated example is to program STFQ [13] for weighted fairness, which is also shown in Figure 1. In this example, the rank of a packet is the virtual start time of the packet in STFQ. The virtual start time is computed as the maximum of the virtual time and the virtual finish time of the previous packet of the same flow. The virtual time maintains the virtual start time of the last dequeued packet across all flows. The virtual finish time of a packet is the virtual start time of the packet plus the length of the packet divided by the flow weight. Given such rank computation, the PIFO queue would schedule the packet with the smallest virtual start time first, i.e., realizing STFQ.

Beyond these two example, it has been shown that PIFO can support a wide range of packet scheduling algorithms, such as Least-Slack Time-First [23], Service-Curve Earliest Deadline First (SC-EDF) [40], etc.

2.2 Motivating Example

While PIFO is an appealing solution for programmable packet scheduling, it is challenging to implement in hardware, especially in switch ASICs. The rank computation component is relatively easy. It can be implemented as a packet transaction [46] in the data plane of existing programmable switches. The major challenge is to implement the PIFO queue. Existing switches do not support a sorted queue in the data plane. There is a proposal on how to support a sorted queue in the data plane at 1 GHz [47]. But the proposal only provides a design, not a real implementation, and the design is not scalable as it can only support a few thousand flows. SP-PIFO [3] provides an approximation of a PIFO queue using multiple strict-priority queues. But strict-priority queues are precious hardware resources as commodity switches have a limited number of strict-priority queues and the operators would like to use them to ensure strong physical isolation between multiple tenants. In this paper, we aim to design a solution that has the minimal hardware requirements for programmable packet scheduling.

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

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

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

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

3 DESIGN GOAL

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

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

$$\Delta(t) = \frac{|\mathbb{P}(t) \setminus \mathbb{A}(t)| + |\mathbb{A}(t) \setminus \mathbb{P}(t)|}{|\mathbb{P}(t)| + |\mathbb{A}(t)|} \quad (1)$$

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

We have $\Delta(t) \in [0, 1]$, and a large value of Δ indicates a large gap between AIFO and PIFO. When AIFO and PIFO dequeue the same set of packets (i.e., no gap), $\Delta = 0$; when AIFO and PIFO dequeue completely different packets, $\Delta = 1$. We theoretically prove that the difference between AIFO and PIFO is negligible when the system is stationary (§4), and empirically demonstrate that AIFO provides close performance as PIFO with a range of real workloads (§5).

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

We argue that this metric is less important than the first metric, and sometimes is even undesirable to optimize for. First, there are two important trends for datacenter networking: (*i*) the trend towards shallow buffers for low latency in modern datacenters [5]; and (*ii*) the trend towards tight control loops at end hosts [22]. The confluence of these two trends ensures that the switch queues would not buffer many packets, making the difference on the dequeueing ordering between PIFO and AIFO minimal. As we are essentially emulating PIFO with a FIFO queue, we want to keep the buffer shallow so that the packets can have a short waiting time in the queue. Empirically, we show in Section 5 that such a difference would not impact the flow-level metrics like flow completion time (FCT) much and AIFO behaves almost the same as PIFO.

Second, strictly following PIFO causes packet reorderings, which is undesirable. SRPT achieves near optimality on minimizing FCTs [41]; it schedules flows based on the remaining flow size, so that small flows are scheduled before big flows to minimize FCTs. Packet reorderings happen when PIFO is programmed to implement SRPT by using the remaining flow size as the rank (like Figure 1). This is because for the *same* flow, a latter packet would have a *smaller* remaining flow size than its previous packet, and thus is scheduled first by the switch if both packets are enqueued by the switch.

This is a known issue, and pFabric [6] addresses this issue by adding an extra feature called *starvation prevention* to SRPT. Starvation prevention dequeues the packets of the same flow in the order they arrive, so that the first packet of a flow would be dequeued first if the flow is scheduled and the first packet would not be *starved*. Between flows, pFabric uses SRPT to select which flow to schedule first. Given the strong demand on low-latency for datacenter networks, pFabric is arguably the killer application of programmable packet scheduling. Yet, it cannot be supported by PIFO [47]. Unexpectedly, a positive byproduct of AIFO is that it naturally supports starvation prevention and eliminates packet reordering by design.

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

4 AIFO DESIGN

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

4.1 Key Ideas

AIFO only uses a single FIFO queue, instead of a PIFO queue or multiple strict-priority FIFO queues. It adds admission control in front of the FIFO queue to decide whether to admit or drop an arriving packet. Admitted packets are buffered and sent by the queue in FIFO order, and no extra scheduling is needed. The admission control is designed to minimize Δ described in Section 3, in order to minimize the gap between AIFO and PIFO.

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

- **Temporal component.** The threshold of admission control is dynamic, instead of fixed. It is updated based on the real-time discrepancy between arrival rate and departure rate. When the arrival rate significantly exceeds the departure rate, the threshold becomes more aggressive. It ensures the rate of admitted packets roughly matches the departure rate.
- **Spatial component.** The admission control treats packets differently based on their ranks, instead of using a naive rank-agnostic criteria (e.g., randomly dropping 10% packets). It prefers to drop high-rank packets over low-rank packets, as low-rank packets are expected to be scheduled first. The threshold is decided based

Algorithm 1 AIFO

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

on the arrival rate distribution of different ranks. It ensures the admitted packets have similar ranks as those admitted by PIFO.

We note that the basic idea of dynamic, proportional adaption is widely used, and in particular for networking, it has been instantiated in various forms in congestion control [4, 22, 30]. For examples, delay-based congestion control algorithms like TIMELY and Swift [22, 30] adapt the TCP window size dynamically based on the end-to-end delay, and ECN-based algorithms like DCTCP [4] adapt the window size in proportional to the number of packets with the ECN flag. These instantiations all put the control in the end hosts. In comparison, AIFO places the dynamic, proportional adaption in the network, and it serves a different purpose, i.e., programmable packet scheduling. This context brings stringent requirements to the algorithm design: the algorithm should not only achieve optimality, but also be carefully designed to be implemented at line rate.

For readers familiar with the packet scheduling literature, AIFO can be considered as an AQM solution. Traditional AQM solutions consider a specific objective, and drop packets using threshold comparisons against average queue length [12, 34, 35] or delay estimations [32]. In comparison, AIFO is designed to be a general solution that can be programmed to support different objectives, and it drops packets with a combination of threshold comparisons (i.e., the temporal component) and relative packet rank estimations (i.e., the spatial component).

4.2 Algorithm

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

Next, we explain the admission control part in detail. For the temporal component, it uses the difference between the current

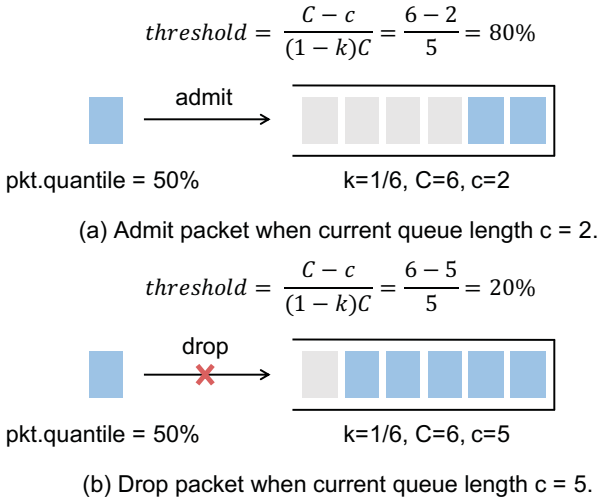


Figure 4: Examples of admission control in AIFO.

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

It is important to note that C is not necessarily the physical size of the FIFO queue. The physical size of a queue in a commodity switch varies in a large range from tens of packets to hundreds or even thousands of packets, depending on the switch ASIC. Despite this capability, production networks tend to use shallow buffers and limit the queue size in deployment for low latency. As such, C can be configured to a smaller number than the physical queue size, and thus we term it as the target (not physical) queue size in the algorithm description.

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

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

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

We provide two examples to illustrate different cases in the admission control. The examples are shown in Figure 4. The target queue length C is 6 and the headroom parameter k is 1/6 (i.e., a headroom of $6\times1/6 = 1$ packet). Suppose the quantile of the arriving packet's rank is 50%.

- **Case 1: admit packet below quantile threshold.** When the current queue length c is 2, the quantile threshold is $\frac{1}{1-k}\frac{C-c}{C} = 80\%$. This means a packet can be admitted if the quantile of the packet's rank is no bigger than 80%. Since the quantile of the arriving packet's rank is 50%, which is smaller than 80%, the packet is admitted.
- **Case 2: drop packet above quantile threshold.** When the current queue length c is 5, the quantile threshold is $\frac{1}{1-k}\frac{C-c}{C} = 20\%$. This means a packet can be admitted if the quantile of the packet's rank is no bigger than 20%. Since the quantile of the arriving packet's rank is 50%, which is bigger than 20%, the packet is dropped.

4.3 Theoretical Guarantee

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

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

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

(1) AIFO and PIFO has the same departure rate for each rank:
- *for rank $i < n^*$, its departure rate is λ_i;*
- *for rank $i = n^*$, its averaged departure rate is $\gamma - \sum_{i<n_*}\lambda_i$;*
- *for rank $i > n^*$, its departure rate is zero.*

(2) FIFO does not perform as the same as PIFO:
- *for rank $i \in \{1, \ldots, n\}$, its departure rate is $\frac{\lambda_i}{\sum_{i=1}^{n}\lambda_i}\gamma$.*

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

(1) The queue length for PIFO and FIFO is C.

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

This theorem means that at the stationary state, the departure rate of each packet rank with AIFO is the same as that with PIFO.

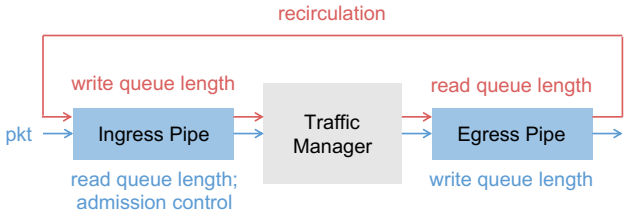


Figure 5: Worker packets carry queue length information from egress pipe to ingress pipe via recirculation. Normal packets read queue lengths and make admission control decisions at ingress pipe. Normal packets also write queue lengths at egress pipe.

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

Admitted packet set. Consider a time interval from 0 to T and n packet ranks. Let $a_i(t)$ be the departure rate of rank i with AIFO, and $p_i(t)$ be the departure rate of rank i with PIFO. We can prove properties for the difference between the departure packets with AIFO and those with PIFO.

Theorem 2. *Adopt the assumption in Theorem 1. Suppose the systems are initialized at time 0, and after time t_0 both PIFO and AIFO reach and stay at their stationary states. Then for the gap measure defined in Eq. (1), we have*

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

Theorem 1 already provides a strong guarantee on the departure rate of each rank. This theorem goes further to show the gap on the difference of the dequeued packets. It proves that Δ defined in Section 3 is close to 0, meaning that AIFO and PIFO dequeue the same set of packets.

4.4 Data Plane Design and Implementation

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

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

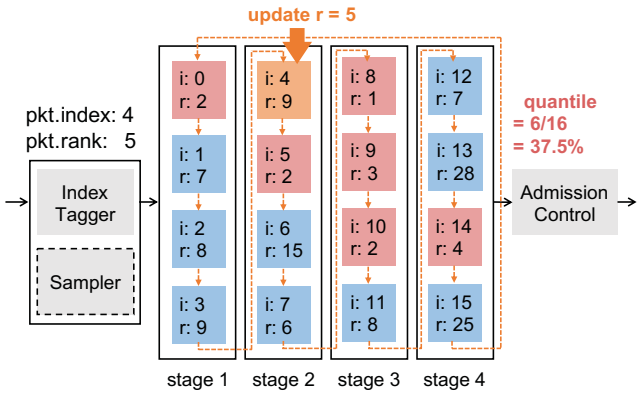


Figure 6: Compute quantile with a sliding window.

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

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

As the worker packets make the queue lengths ready in the ingress pipe, a normal arriving packet can then access the queue length information in the ingress pipe. After the routing decision is made for the packet (i.e., the egress port is known), it can read the queue length of its egress port from *q_len_ingress*. Then the threshold $\frac{1}{1-k}\frac{C-c}{C}$ can be calculated with the queue length to decide whether to admit or drop the packet. If the packet is admitted, it also writes the current queue length to the egress pipe. Figure 5 illustrates how the solution works.

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

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

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

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

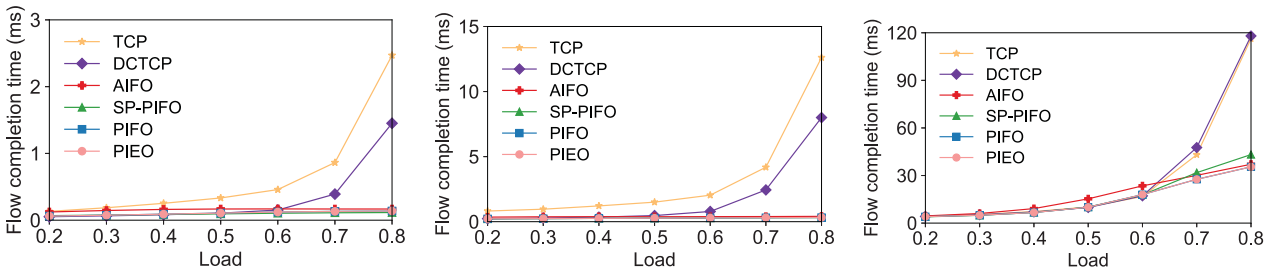
In Figure 6, the rank of the arriving packet is 5. The rank is smaller than the values of 6 registers, which are marked with red in the figure. As the size of the sliding window n is 16, the quantile is $6/16 = 37.5\%$.

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

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

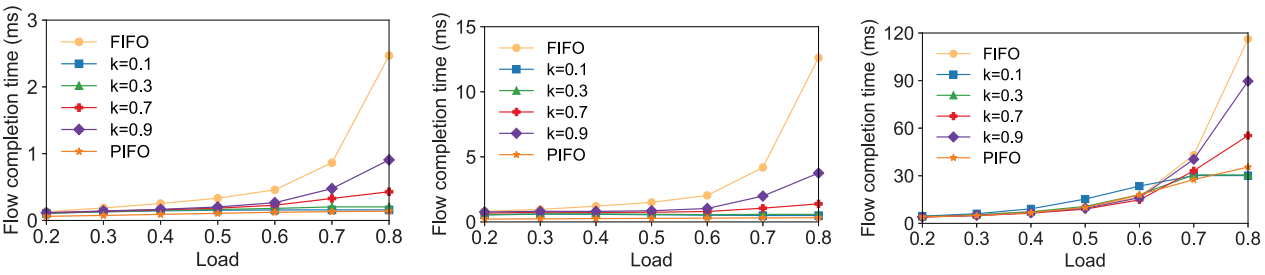
5 EVALUATION

In this section, we provide experimental results to demonstrate the performance of AIFO. We first evaluate AIFO using simulations to show that AIFO can achieve high performance in a large-scale datacenter environment. In the simulations, we benchmark AIFO with state-of-the-art solutions to demonstrate its end-to-end performance. Besides, we also evaluate the effect of different parameters, and the admitted packet set of AIFO. At last, we evaluate our prototype for AIFO on a Barefoot Tofino switch in a hardware testbed.



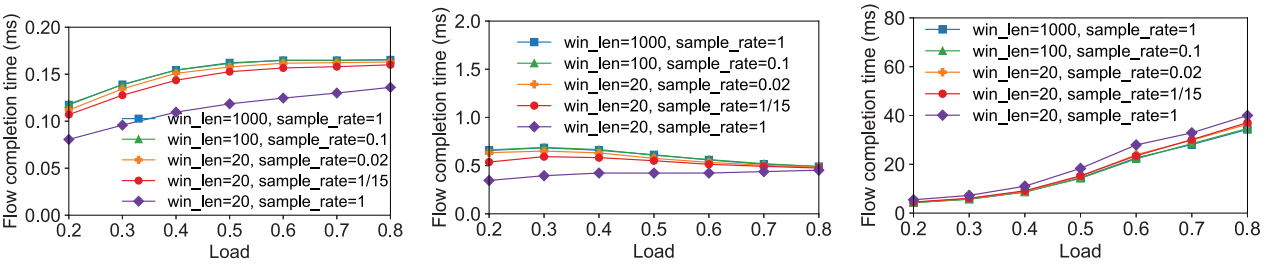
(a) Average FCT for small flows. (b) 99th FCT for small flows. (c) Avearage FCT for large flows.

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



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

Figure 8: The effect of parameter k.



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

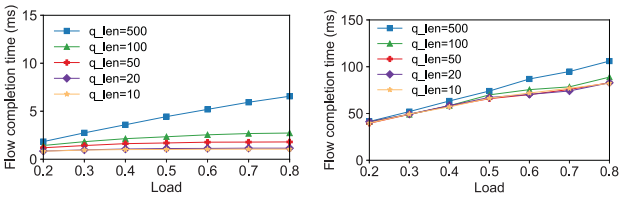
Figure 9: The effect of window length and sampling rate.

5.1 Packet-Level Simulations

We use packet-level simulations to evaluate AIFO in a large-scale datacenter environment. We use a similar setting as recent works on packet scheduling [3, 6]: a leaf-spine topology which contains 9 leaf switches, 4 spine switches and 144 servers, and the bandwidth of the access and leaf-spine links is set at 10Gbps and 40Gbps, respectively. The simulations are conducted with Netbench [1], a packet-level simulator.

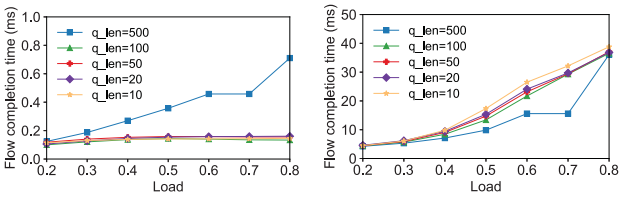
We evaluate two use cases of programmable packet scheduling: minimizing FCT and providing fairness. (*i*) We use AIFO to implement pFabric, and compare it with TCP, DCTCP as well as state-of-the-art approaches PIFO [47], SP-PIFO [3], and PIEO [45] under a realistic traffic workload: web search workload [6]. We also conduct a sensitivity analysis to evaluate and analyze the effect of different parameters (i.e., queue length, scaling parameter k, window length and sampling rate) on AIFO and the admitted packet set of AIFO. (*ii*) We implement Start-Time Fair Queueing (STFQ) on top of AIFO and compare it with other state-of-the-art solutions. For AIFO, we set the target queue length as 20, $k = 0.1$, window length as 20, and sampling rate as $\frac{1}{15}$ by default.

Minimizing FCT with AIFO. We first show the performance of AIFO when implementing SRPT for pFabric [6] to minimize FCT under the web search workload. The traffic starts according to a Poisson distribution. For comparison, we also implement SRPT



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

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

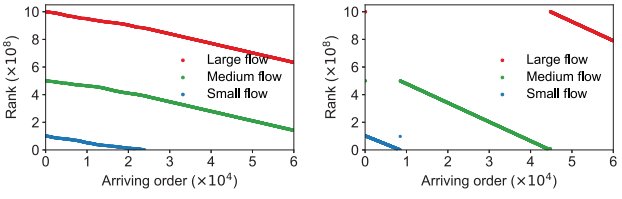


(a) Average FCT for small flows. (b) Average FCT for large flows.

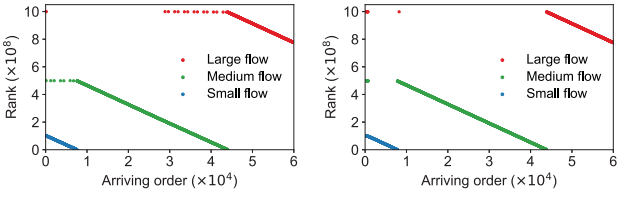
Figure 11: The effect of queue length on 10G/40G network.

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

The effect of parameter k. The headroom parameter k controls how aggressively AIFO drops high-rank packets. We set k among $0.1 \sim 0.9$ and compare the results with FIFO and PIFO. Figure 8 shows the results. AIFO with smaller k always delivers better performance than larger k for small flows, and it also delivers better performance for large flows when the traffic load is big (e.g., 0.7, 0.8). The reason is that with a small k, AIFO drops packets aggresively and keeps the buffer shallow so that the admitted packets get low latency. When k is small, AIFO delivers a close performance compared with PIFO. As we increase k, AIFO admits more packets and it becomes closer to FIFO. When the traffic load grows, the queue buffer accumulates quickly, and it leads to a large delay. While

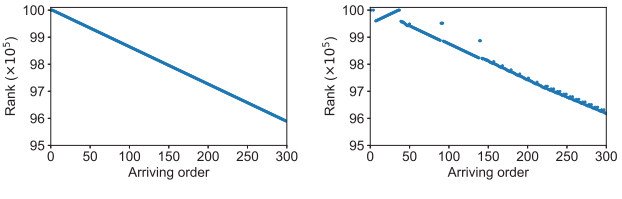


(a) FIFO. (b) PIFO.

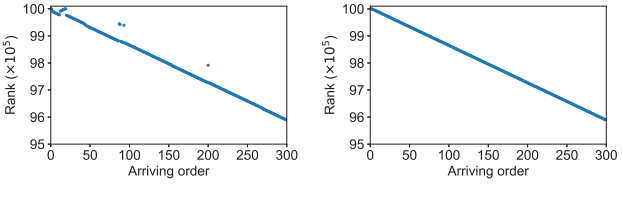


(c) SP-PIFO. (d) AIFO.

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



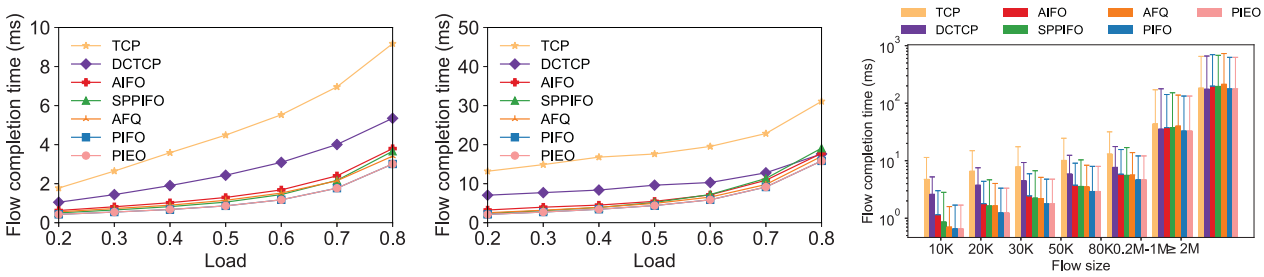
(a) FIFO. (b) PIFO.



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

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

dropping packets aggressively harms large flows especially when the traffic load is not big and the network capacity is underutilized, we show that the harm is slight compared with the benefit it brings to small flows. When $k = 0.1$, the average and 99th percentile FCT for small flows is about 9× lower than that of $k = 0.9$, and the FCT for large flows is only slightly higher than the lowest.



(a) Average FCT for small flows.

(b) 99th FCT for small flows.

(c) FCT breakdown for 70% load.

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

The effect of window length and sampling rate. We also evaluate how the sliding window length affects the performance of AIFO and how well a small sliding window approximates a large sliding window with sampling for AIFO. As shown in Figure 9, when the window length is 20, the performance is better than that when the window length is 1000 for small flows, but worse for large flows. It is because that a large window records packets for a longer time, and the possibility for packets from large flows (high-rank packets) to be admitted is more stable. As a result, there are more high-rank packets admitted into the queue in the long run, which makes the FCT for small flows higher and FCT for large flows lower. It is interesting to see in Figure 9(b) that the 99th percentile FCT is decreasing as the traffic load grows when win_len = 1000. The reason is that when the window is large and the traffic load is low, the quantile is less accurate. The flows that experience deep buffer and inaccurate quantile estimation would have larger FCTs, and these flows would normally contribute to the 99th FCT.

By comparing lines <win_len=20, $sample\_rate$=0.02>, <win_len =100, $sample\_rate$=0.1> and <win_len=1000, $sample\_rate$=1>, we can see that AIFO does not require a very precise quantile and a small window can approximate a large window with sampling. This is important to the practicability of AIFO as a window with 20 slots can be implemented in programmable switches with tiny resource consumption.

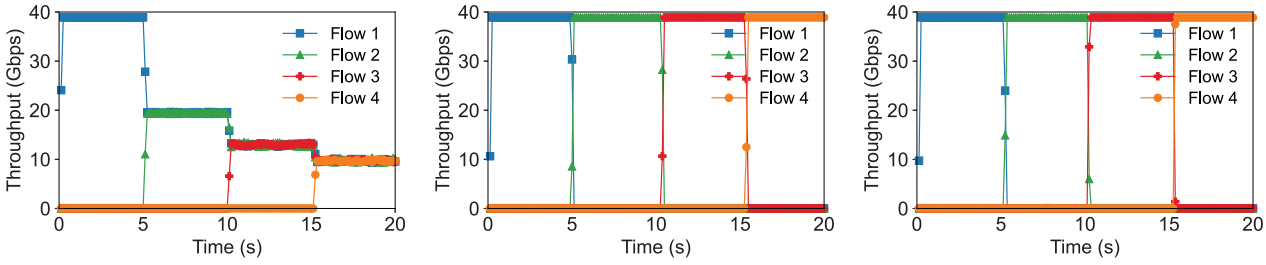
The effect of queue length. To evaluate the impact of queue length on the performance of AIFO, we use different queue lengths and run simulations on both a 1G/4G network (access link: 1Gbps, leaf-spine link: 4Gbps) and a 10G/40G network. As shown in Figure 10 and Figure 11, AIFO is more sensitive to the change of queue length when the bandwidth is low or when the traffic load is high. Figure 11 shows that FCT achieved by AIFO when q_len = 20 is close to q_len = 100 on the 10G/40G network. However, FCT achieved by AIFO when q_len = 20 is much smaller than q_len = 100 in Figure 10 on the 1G/4G network. This is because it takes a while for a long queue to drain when the bandwidth is low, which leads to a considerable queueing delay. A relatively small queue benefits the FCT.

Admitted packet sets. Recall that we indicate in Theorem 2 that the sets of packets dequeued by AIFO and PIFO are similar. This experiment examines the gap between AIFO and PIFO in terms of the difference between the packets dequeued by AIFO and those dequeued by PIFO. Here we use four servers and the servers are connected with a Top-of-Rack switch. The bandwidth of the links between the servers and the switch are set as 1Gbps. We let three servers serve as senders, and the other server serves as a receiver. Each sender sends one flow to the receiver at the same time, and the sizes of the flows are 100MB (large), 50MB(medium), and 10MB(small), respectively. The servers run pFabric as the transport and the flows are tagged with the remaining flow size as their rank. The switch is programmed to support SRPT with AIFO, PIFO or SP-PIFO.

We log the ranks of the first 60000 packets received by the receiver, and plot the log in Figure 12. The x-axis is the arriving order of the packets, and the y-axis is the rank of the packets. As shown in Figure 12, when the network is running FIFO without admission control and packet scheduling, the three flows share the bandwidth and the small flow (blue) finishes late. For the other three solutions (AIFO, PIFO, SP-PIFO), the small flow finishes at about the same time, and it finishes much earlier than it does with FIFO. Besides, it is shown that AIFO is closer to PIFO than SP-PIFO in terms of the admitted packets set: there are larger overlaps on the arriving order (x-axis) between the small flow (blue) and the medium flow (green), as well as between the medium flow and the large flow (red) in Figure 12(c), than those in Figure 12(d) and Figure 12(b).

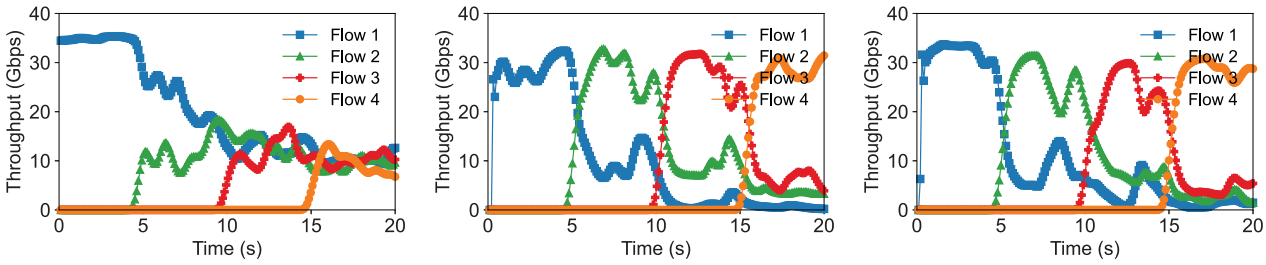
Packet reordering. Besides the admitted set, another interesting metric is the dequeued order of the packets. PIFO always dequeues the packet with the lowest rank in the queue, which may cause out-of-order and harm the end-to-end performance. However, as AIFO only enforces admission control on a FIFO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

We run the same setting as in Figure 12 and we only log the first 300 packets of the small flow in order to show the packet out-of-order clearly. As shown in Figure 13(a) and Figure 13(d), when we enable AIFO, the packet order of one flow is the same as that with FIFO and there is no packet out-of-order. It is because that AIFO only uses a FIFO queue and does not do packet scheduling inside the queue. However, both PIFO and SP-PIFO get some out-of-order packets, as shown in Figure 13(b) and Figure 13(c). The reason is that PIFO and SP-PIFO always dequeue the packet with the lowest rank, while the packets with higher rank will be left in the queue and be scheduled later. With pFabric, the rank is based on the remaining



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

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



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

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

flow size, so a later packet has a lower rank compared with an earlier packet. As a result, these two methods lead to a number of out-of-order packets. As SP-PIFO approximates PIFO with a set of FIFO queues, it causes fewer out-of-orders compared with PIFO.

Fair queueing with AIFO. Programmable packet schedulers like PIFO can be used to implement different kinds of packet scheduling algorithms by changing the rank computation function. Besides implementing SRPT to minimize FCTs, here we show how AIFO performs when we implement Start-Time Fair Queueing (STFQ) [13] on top of it for fair queueing. We also implement STFQ on top of PIFO, SP-PIFO and PIEO to compare them with AIFO. Besides, we include TCP, DCTCP, and the state-of-the-art fair queueing solution AFQ for comparison. We run the web search workload and show the average FCT for small flows (Figure 14(a)), the $99th$ percentile FCT for small flows (Figure 14(b)), and the breakdown of FCT for different flow sizes (Figure 14(c)). AIFO achieves a similar performance compared to the state-of-the-art approaches AFQ, SP-PIFO, PIFO and PIEO for both average FCT and tail FCT, and is significantly better than TCP and DCTCP. The FCT of AIFO for small flows is only 9.7% higher than AFQ and 3.6% higher than SP-PIFO, despite AIFO using only a single queue.

5.2 Testbed Experiments

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

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

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

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

Table 1: Resource consumption of AIFO and SP-PIFO prototypes on Intel Barefoot Tofino. Each number indicates the percentage of resources consumed for the corresponding type.

4 occupies most of the bandwidth. The result is almost identical to SP-PIFO (Figure 15(b)) as the flow with the lowest rank always occupies most of the bandwidth.

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

Resource consumption. AIFO uses only one queue and achieves similar performance as SP-PIFO with eight queues. Table 1 lists the consumption of other switch resources. It shows that AIFO has a higher demand on Match Crossbars, Gateway, Hash Bits, ALUs and Logical TableIDs, while SP-PIFO has a higher demand on SRAM and TCAM.

6 RELATED WORK

Programmable networking. The emergence of programmable networking has triggered many novel applications in network data plane [14, 16, 18–20, 24–26, 28, 31, 36, 39, 49–51]. Among them, programmable packet scheduling [29, 47] is an attracting direction. Programmable packet scheduling in the data plane as opposed to traditional fixed-function packet scheduling [11, 21, 27, 44, 44, 48] is a relatively new concept. After PIFO [47] and UPS [29], several solutions for enabling programmable scheduling have proposed a combination of new abstractions, new algorithms, and new queue structures [38, 42, 43, 45]. However, many of these rely on new hardware designs. SP-PIFO [3] recently shows that efficient programmable packet scheduling can be approximated by using existing devices with as few as eight queues. In this paper, we show that AIFO can closely approximate PIFO with just one queue.

Priority-based scheduling. Priority-based scheduling is a classic scheduling discipline [41] that is often used in the networking context to minimize the average completion time of flows [6, 7, 15, 17] and coflows [8, 9] in both clairvoyant (size is known a priori) and non-clairvoyant (unknown size) scenarios. In the latter case, most of the solutions boil down classic solutions such as Multi-level Feedback Queues (MLFQ) [10] and its continuous approximations [33, 37]. Programmable packet scheduling uses the notion of ranks, which is similar to priorities, but more general in the sense that the definition of ranks can be programmed based on the requirements of users. This general notion has been shown to be able to support a wide variety of different packet scheduling algorithms for different objectives [47].

Active queue management (AQM). Working in conjunction with packet scheduling algorithms, AQM performs admission control by probabilistically dropping packets to prevent congestion. AQM is simple and implemented widely in most switches (e.g., RED [12]). There are many variations: e.g., to improve fairness [34, 35] and to provide bounded worst-case packet queuing delay [32] to name a few. Unlike traditional AQM proposals, AIFO proactively drops packets based on their relative ranks instead of randomly dropping them.

7 CONCLUSION

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

Ethics. This work does not raise any ethical issues.

Acknowledgments. We thank our shepherd George Varghese and the anonymous reviewers for their valuable feedback on this paper. Xin Jin (xinjinpku@pku.edu.cn) is the corresponding author. Xin Jin is with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education. This work is supported in part by NSF grants CCF-1652257, CNS-1813487, CNS-1845853, and CCF-1918757, Project 2020BD007 from PKU-Baidu Fund, ONR Award N00014-18-1-2364, and the Lifelong Learning Machines program from DARPA/MTO.

REFERENCES

[1] 2017. Netbench. http://github.com/ndal-eth/.
[2] 2018. Intel Data Plane Development Kit (DPDK). http://dpdk.org/.
[3] Albert Gran Alcoz, Alexander Dietmüller, and Laurent Vanbever. 2020. SP-PIFO: Approximating Push-In First-Out Behaviors using Strict-Priority Queues. In *USENIX NSDI*.
[4] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2011. Data Center TCP (DCTCP). In *ACM SIGCOMM*.
[5] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. 2012. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *USENIX NSDI*.
[6] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. pfabric: Minimal near-optimal datacenter transport. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 435–446.
[7] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. 2015. Information-Agnostic Flow Scheduling for Commodity Data Centers. In *USENIX NSDI*.
[8] Mosharaf Chowdhury and Ion Stoica. 2015. Efficient Coflow Scheduling Without Prior Knowledge. In *ACM SIGCOMM*.
[9] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. 2014. Efficient coflow scheduling with Varys. In *ACM SIGCOMM*.
[10] Fernando J Corbató, Marjorie Merwin-Daggett, and Robert C Daley. 1962. An experimental time-sharing system. In *Spring Joint Computer Conference*. 335–344.
[11] Alan Demers, Srinivasan Keshav, and Scott Shenker. 1989. Analysis and Simulation of a Fair Queueing Algorithm. *SIGCOMM CCR* (August 1989).
[12] Sally Floyd and Van Jacobson. 1993. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking* 1, 4 (1993), 397–1413.
[13] Pawan Goyal, Harrick M Vin, and Haichen Cheng. 1997. Start-time fair queueing: A scheduling algorithm for integrated services packet switching networks. *IEEE/ACM Transactions on Networking* 5, 5 (1997), 690–704.
[14] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2018. Sonata: Query-driven streaming network telemetry. In *ACM SIGCOMM*.
[15] Chi-Yao Hong, Matthew Caesar, and P Godfrey. 2012. Finishing flows quickly with preemptive scheduling. In *ACM SIGCOMM*.
[16] Kuo-Feng Hsu, Ryan Beckett, Ang Chen, Jennifer Rexford, Praveen Tammana, and David Walker. 2020. Contra: A programmable system for performance-aware routing. In *USENIX NSDI*.
[17] Shuihai Hu, Wei Bai, Gaoxiong Zeng, Zilong Wang, Baochen Qiao, Kai Chen, Kun Tan, and Yi Wang. 2020. Aeolus: A building block for proactive transport in datacenters. In *ACM SIGCOMM*.
[18] Nikita Ivkin, Zhuolong Yu, Vladimir Braverman, and Xin Jin. 2019. Qpipe: Quantiles sketch fully in the data plane. In *ACM CoNEXT*.
[19] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. NetChain: Scale-Free Sub-RTT Coordination. In *USENIX NSDI*.
[20] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *ACM SOSP*.
[21] Srinivasan Keshav. 1991. On the efficient implementation of fair queueing. *Internetworking: Research and Experience* (September 1991).
[22] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan MG Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. 2020. Swift: Delay is simple and effective for congestion control in the datacenter. In *ACM SIGCOMM*.
[23] Joseph Y-T Leung. 1989. A new algorithm for scheduling periodic, real-time tasks. *Algorithmica* 4, 1 (1989), 209–219.
[24] Jialin Li, Ellis Michael, and Dan R. K. Ports. 2017. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control. In *ACM SOSP*.
[25] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R.K. Ports. 2016. Just say NO to Paxos overhead: Replacing consensus with network ordering. In *USENIX OSDI*.
[26] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. 2019.. DistCache: Provable Load Balancing for Large-Scale Storage Systems with Distributed Caching. In *USENIX FAST*.
[27] Paul E McKenney. 1990. Stochastic Fairness Queueing.. In *IEEE INFOCOM*.
[28] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching ASICs. In *ACM SIGCOMM*.
[29] Radhika Mittal, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Universal packet scheduling. In *USENIX NSDI*.
[30] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. 2015. TIMELY: RTT-based congestion control for the datacenter. *SIGCOMM CCR* (August 2015).
[31] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. 2017. Language-Directed Hardware Design for Network Performance Monitoring. In *ACM SIGCOMM*.
[32] Kathleen Nichols and Van Jacobson. 2012. Controlling Queue Delay: A modern AQM is just one piece of the solution to bufferbloat. In *ACM Queue*.
[33] Misja Nuyens and Adam Wierman. 2008. The Foreground–Background queue: A survey. *Performance Evaluation* 65, 3 (2008), 286–307.
[34] Rong Pan, Lee Breslau, Balaji Prabhakar, and Scott Shenker. 2003. Approximate Fairness through Differential Dropping. In *ACM SIGCOMM*.
[35] Rong Pan, Balaji Prabhakar, and Konstantinos Psounis. 2000. CHOKe: A stateless active queue management scheme for approximating fair bandwidth allocation. In *IEEE INFOCOM*.
[36] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. 2015. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *USENIX NSDI*.
[37] Idris A Rai, Guillaume Urvoy-Keller, and Ernst W Biersack. 2003. Analysis of LAS scheduling for job size distributions with high variance. *ACM SIGMETRICS Performance Evaluation Review* 31, 1 (2003), 218–228.
[38] Ahmed Saeed, Yimeng Zhao, Nandita Dukkipati, Ellen Zegura, Mostafa Ammar, Khaled Harras, and Amin Vahdat. 2019. Eiffel: Efficient and flexible software packet scheduling. In *USENIX NSDI*.
[39] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. 2021. Scaling Distributed Machine Learning with In-Network Aggregation. In *USENIX NSDI*.
[40] Hanrijanto Sariowan, Rene L Cruz, and George C Polyzos. 1999. SCED: A generalized scheduling policy for guaranteeing quality-of-service. *IEEE/ACM Transactions on Networking* 7, 5 (1999), 669–684.
[41] Linus E Schrage and Louis W Miller. 1966. The queue M/G/1 with the shortest remaining processing time discipline. *Operations Research* 14, 4 (1966), 670–684.
[42] Naveen Kr Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. 2018. Approximating fair queueing on reconfigurable switches. In *USENIX NSDI*.
[43] Naveen Kr Sharma, Chenxingyu Zhao, Ming Liu, Pravein G Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. 2020. Programmable calendar queues for high-speed packet scheduling. In *USENIX NSDI*.
[44] Madhavapeddi Shreedhar and George Varghese. 1995. Efficient fair queueing using deficit round robin. In *ACM SIGCOMM*.
[45] Vishal Shrivastav. 2019. Fast, scalable, and programmable packet scheduler in hardware. In *ACM SIGCOMM*.
[46] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. 2016. Packet transactions: High-level programming for line-rate switches. In *ACM SIGCOMM*.
[47] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. 2016. Programmable packet scheduling at line rate. In *ACM SIGCOMM*.
[48] Ion Stoica, Scott Shenker, and Hui Zhang. 1998. Core-stateless fair queueing: Achieving approximately fair bandwidth allocations in high speed networks. In *ACM SIGCOMM*.
[49] Zhuolong Yu, Jingfeng Wu, Vladimir Braverman, Ion Stoica, and Xin Jin. 2021. Twenty Years After: Hierarchical Core-Stateless Fair Queueing. In *USENIX NSDI*.
[50] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. 2020. Netlock: Fast, centralized lock management using programmable switches. In *ACM SIGCOMM*.
[51] Hang Zhu, Zhihao Bai, Jialin Li, Ellis Michael, Dan Ports, Ion Stoica, and Xin Jin. 2019. Harmonia: Near-Linear Scalability for Replicated Storage with In-Network Conflict Detection. In *Proceedings of the VLDB Endowment*.

Appendices are supporting material that has not been peer-reviewed.

A PROOF OF THEOREM 1

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

For FIFO, every packets are treated equally based on first arrival first admission principle, thus their incoming rate is proportional to their sending rate. Let us assume the incoming rates of each type of packet to be $k\lambda_1, k\lambda_2, \ldots, k\lambda_n$. On the other hand, at the stationary state, the total incoming rate of the queue equals to its

total outcoming rate γ, i.e., $k\lambda_1 + k\lambda_2 + \cdots + k\lambda_n = \gamma$, which implies $k = \frac{\gamma}{\sum_{j=1}^{n}\lambda_j}$. Thus for packet i, its incoming and outcoming rate is $k\lambda_i = \frac{\gamma\lambda_i}{\sum_{j=1}^{n}\lambda_j}$.

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

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

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

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

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

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

- if $c \leq c^-$, all packets $i \leq n^* + 1$ will be admitted. By the choice of n^* we have $\sum_{j\leq n^*+1}\lambda_j > \gamma$, i.e., the total incoming rate is strictly greater than the total outcoming rate, thus the queue length increases;
- if $c > c^*$, all packets $i \geq n^*$ cannot be admitted. By the choice of n^* we have $\sum_{j\leq n^*-1}\lambda_j < \gamma$, i.e., the total incoming rate is strictly less than the total outcoming rate, thus the queue length decreases;
- if $c^- < c \leq c^*$, all packets $i \leq n^*$ will be admitted, and all packets $i > n^*$ will not be admitted. Note that $\sum_{j\leq n^*}\lambda_j \geq \gamma$. We discuss two cases:
  - if $\sum_{j\leq n^*}\lambda_j = \gamma$, then the system reaches its stationary state at the first time when the queue length satisfies $c^- < c \leq c^*$;
  - if $\sum_{j\leq n^*}\lambda_j > \gamma$, then the queue length keeps increases until it becomes larger than c^*, and falls into the previous category hence the length decreases then. In sum the system reaches its stationary state with queue length being c^*. Moreover, due to the negative feedback principle, in the stationary state, the incoming rate/outcoming rate of packet n^* would be $\gamma - \sum_{j<n^*}\lambda_j$.

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

Remark 1. As an extension to the above setting, we can consider a setting with T time interval, where within each time interval t, the packets are sent with constant sending rate $\lambda_1(t), \ldots, \lambda_n(t)$, but across different time interval, the sending rate of each packet can vary, i.e., $\lambda_i(t) \neq \lambda_i(t')$ for $t \neq t'$. Suppose each time interval is sufficiently long such that the system can reach its stationary state, then we can apply the above theorem within each time interval to characterize the behavior of the algorithms at the stationary state.

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

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

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

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

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

B PROOF OF THEOREM 2

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

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

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

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

which implies

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

Note that (1) the nominator is a constant that is independent of T; and (2) the denominator keeps cumulating with constant non-zero rate at its stationary state. Therefore for any small tolerance $\epsilon > 0$, let the running time T be

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

we have

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

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