Enhancing Fairness for Approximate Weighted Fair Queueing with a Single Queue

Wei Chen, Ye Tian, *Member, IEEE,* Xin Yu, Bowen Zheng, and Xinming Zhang, *Senior Member, IEEE*

***Abstract*—Weighted fair queueing (WFQ) is an essential strategy for enforcing bandwidth guarantee and isolation in high-speed networks. Unfortunately, implementing the original WFQ packet scheduling algorithm on today's commodity switch hardware is challenging due to the prohibitive complexity. Approximate WFQ packet schedulers, which work with the cheap and widely available First-In First-Out (FIFO) queues, have been proposed as an alternative in recent years. In this paper, we show that both the ideal and the approximate WFQ packet schedulers are unable to allocate bandwidths to TCP flows fairly, because of the bursty nature of the TCP traffic. Furthermore, we find that the representative approximate WFQ schedulers further degrade the scheduling fairness, due to their excessive packet drops. To address these issues, we present novel approximate WFQ packet scheduling algorithms in this paper. Our initial design, namely SQ-WFQ, imposes the minimum hardware requirement by using one single FIFO queue, and effectively reduces the excessive packet drops. Extended from SQ-WFQ, we propose the SQ-EWFQ packet scheduling algorithm. SQ-EWFQ inherits all the merits of SQ-WFQ, and is adaptive to the bursty TCP traffic by tolerating short-term packet bursts, while enforcing a long-term fairness among the TCP flows. We have implemented our proposed schedulers on commodity hardware programmable switches, and achieve line rate packet scheduling with them. Experiment results from a real-world testbed and large-scale simulations show that SQ-WFQ and SQ-EWFQ outperform the state-of-the-art approximate schedulers regarding the scheduling fairness, and SQ-EWFQ allocates bandwidths to TCP flows more fairly than SQ-WFQ and other existing solutions.**

***Index Terms*—Weighted fair queueing (WFQ); packet scheduling; TCP; programmable switch**

I. Introduction

Weighted fair queueing (WFQ) is a network scheduling strategy that enforces the generalized processor sharing (GPS) principle among network flows [1]. In WFQ, each backlogged flow is assigned with a weight, and obtains a proportional bandwidth from the port that it shares with other flows. By providing bandwidth guarantees and performance isolation, WFQ is essential for network virtualization in data centers and other high-speed networks.

Unfortunately, realizing WFQ on high-speed networks is challenging. Although a few hardware schedulers have been proposed to sort packets based on their departure times [2], [3], they do not scale due to the prohibitive packet processing overhead. Novel queue abstractions such as Push-In First-Out (PIFO) [4] and Push-In Extract-Out (PIEO) [5] are proposed recently to provide a universal programmability for packet scheduling, but their hardware designs were realized only on experimental platforms such as FPGA (field programmable gate array) because of the complexity.

Instead of faithfully implementing the original WFQ algorithm, a number of approximate packet schedulers based on FIFO (First-In First-Out) queues have been proposed in recent years, as FIFO is the simplest queue that can run at line rate and is available in almost all switches. Among the approximate schedulers, Admission-In First-Out (AIFO) [6] is a representative example that approximates PIFO with one single FIFO queue, and proactively rejects packets based on local comparisons on packet ranks. Programmable Calendar Queue (PCQ) [7] is another exemplary scheduler based on calendar queues, which is composed of a set of priority FIFO queues with a fixed rotation order. However, as we will see in this paper, existing approximate schedulers incur considerable scheduling errors, and drop packets excessively.

All the WFQ packet schedulers make an implicit assumption that packets of a network flow are independent, and dropping a packet will not influence the arrivals of the flow's subsequent packets. With such an assumption, the schedulers enforce their scheduling decisions by dropping packets of the flows whose bandwidth usages exceed their fair shares. However, the assumption does not hold for TCP, as a TCP sender transmits packets in bursts and adjusts its packet sending behavior according to the feedbacks including packet losses. Our analysis and experiments in this paper show that both the ideal and the approximate WFQ schedulers have unexpected impacts on TCP, and allocate bandwidths unfairly; moreover, the AIFO and PCQ-based approximate schedulers further degrade the fairness due to their excessive packet drops.

For enhancing fairness in bandwidth allocation, in this paper, we propose novel WFQ approximate packet scheduling algorithms. Our initial design, namely *SQ-WFQ*, reduces the excessive packet drops comparing with the AIFO and PCQ-based schedulers. Extended from SQ-WFQ, we propose *SQ-EWFQ*, a WFQ approximate scheduler dedicatedly designed for the bursty TCP traffic. In addition to preserving all the merits of SQ-WFQ, SQ-EWFQ adapts to TCP by tolerating short-term packet bursts in TCP flows, while enforcing a long-term fairness in packet scheduling. Unlike many schedulers using multiple priority queues per egress port [7]–[10], SQ-WFQ and SQ-EWFQ require one single FIFO queue per port, thus greatly save the precious physical queue resources in hardware switches.

We have implemented SQ-WFQ and SQ-EWFQ on com-

This work was supported in part by the National Natural Science Foundation of China (NSFC) under Grants 61672486 and 62072425. (*Corresponding author: Ye Tian (yetian@ustc.edu.cn)*).

The authors are with Anhui Key Laboratory on High-Performance Computing, School of Computer Science and Technology, University of Science and Technology of China, Hefei, Anhui, China, 230026. (email: {szcw33, yuxin5836, bwzheng}@mail.ustc.edu.cn, {yetian, xinming}@ustc.edu.cn)

modity hardware programmable switches based on the Intel Tofino chip [11], and evaluate our proposed algorithms with a real-world testbed and large-scale simulations. The experiment results show that both SQ-WFQ and SQ-EWFQ schedule packets at line rate and are work-conserving, and both schedulers improve fairness in scheduling TCP flows by reducing the excessive packet drops. In particular, SQ-EWFQ is capable to allocate bandwidths to small-weighted TCP flows, TCP flows with longer round-trip times (RTTs), TCP flows applying less aggressive congestion controls, and small-to-medium sized TCP flows more fairly than other solutions.

In summary, our key contributions in this paper are:

- We analyze the representative approximate WFQ packet schedulers based on AIFO [6] and PCQ [7], and show that they make excessive packet drops. We further reveal that the ideal WFQ algorithm is unfair when scheduling TCP flows, and the approximate schedulers further degrade the fairness due to their excessive packet drops.
- We propose novel approximate packet scheduling algorithms. Our initial design, namely SQ-WFQ, schedules packets with one single FIFO queue and reduces excessive packet drops. Extended from SQ-WFQ, we propose the SQ-EWFQ algorithm that inherits all the merits of SQ-WFQ, and is adaptive to the bursty TCP traffic. SQ-EWFQ provides burst tolerance to TCP flows by temporally increasing a flow's weight when detecting a packet burst, while enforces a long-term fairness by constraining the total amount of bytes a flow can enqueue over an interval that is sufficiently long for covering a burst duration.
- We evaluate our proposed SQ-WFQ and SQ-EWFQ schedulers with a real-world testbed and large-scale simulations. We find that SQ-WFQ outperforms existing approximate WFQ schedulers by reducing excessive packet drops, and SQ-EWFQ allocates bandwidth to TCP flows more fairly than all the existing approaches.
- We implemented both SQ-WFQ and SQ-EWFQ with the P4 language [12] on Tofino-based commodity hardware switches, and achieve line rate packet scheduling with them. We make our implementations publicly available at https://github.com/HPCC724/SingeQueue_WFQ.

To our best knowledge, we are the first to address the unfairness issue of WFQ packet scheduling on TCP flows, and present solutions for enhancing the scheduling fairness. The remainder part of this paper is organized as follows. We discuss the related works in Sec. II. Sec. III analyzes the existing WFQ schedulers and presents our motivation; We describe the design and implementation of our proposed WFQ approximate schedulers in Sec. IV; Sec. V presents the evaluation results and we conclude in Sec. VI.

II. Related Work

A. Programmable Packet Scheduling

There is a rich literature of packet scheduling, and representative algorithms such as WFQ [13], pFabric [14]–[16], STFQ [17], SRPT [18], [19], EDF [20] provide various guarantees in network services. On the other hand, conventional hardware switches have limited and fixed functionalities, and can not support the scheduling algorithms flexibly.

To provide programmability for packet scheduling, novel queueing abstractions were proposed in recent years. For example, PIFO [4] assigns each packet a rank, and sorts packets in an ascending order according to their ranks and dequeues from the head. PIEO [5] provides a better expressiveness than PIFO by supporting a programmable filtering at dequeue. Although a wide range of packet scheduling algorithms including fair queueing can be realized with PIFO and PIEO, however, they require sorting packets' priority ranks at line rate, which is challenging for today's commodity switch hardware.

To approximate PIFO with the existing FIFO-based switch hardware, SP-PIFO [8] employs multiple strict-priority queues, maps packet ranks to queue priorities, and dynamically adjusts the queue priority bounds to reduce the scheduling errors. AIFO [6] approximates PIFO with one single FIFO queue, and proactively rejects a packet if its rank is relatively high among a number of recently received packets. PCQ [7] imposes a fixed rotation order on a set of priority queues, and transforms a packet's rank to the index of the queue it is enqueued to. Although many packet scheduling algorithms can be approximated, however, the approximation itself comes at a cost of scheduling errors, as we will show in this paper.

B. Fair Queueing

For enforcing (weighted) fair queueing among network flows, the bit-by-bit round robin (BR) algorithm [13] computes a bid number to estimate the departure time for each packet, and transmits the packet that departs earliest. By assigning packet rank as its departure time, the BR algorithm can be emulated with PIFO [4] and its approximations such as AIFO [6]. To avoid employing a massive number of priority queues, the SFQ [21] algorithm maps network flows to a small set of priority queues using a hash function, and perturbs the mapping periodically. The DRR [22] algorithm improves SFQ's fairness by assigning a quantum to each queue for controlling the bytes a queue can send in each round. The AFQ [9] algorithm approximates fair queueing with a scheduler that reuses a set of priority queues by periodically rotating them, and the authors extend the scheduler to the more generalized PCQ [7] for approximating a wider range of packet scheduling algorithms including WFQ. EFQ [23] enhances AFQ by improving the queue unitization and reducing unnecessary packet drops. GearBox [10] improves PCQ with a logical FIFO queue hierarchy that can support a wider range of packet departure times and reduce the departure time discrepancy. Note that most approximate packet schedulers require multiple queues per egress port, while physical queues are precious in hardware switches. Similar to AIFO [6], the schedulers we propose in this paper require only one single queue per port.

Besides packet scheduling, another approach for fair bandwidth allocation is probabilistic packet dropping, in which packets of a flow that exceeds its fair share bandwidth are dropped at a certain probability. Early examples include RED-PD [24], AFD [25], and CSFQ [26]. By carrying per-flow rate and hierarchy information within packet header, HCSFQ [27] approximates hierarchical fair queueing with a fluid algorithm. AHAB [28] employs control plane to collect bandwidth demands and compute bandwidth allocations. Cebinae [29] is a mechanism that augments fairness with packet dropping penalties on flows exceeding their fair share bandwidth. Unlike the solutions that punish bursty TCP flows, our proposed SQ-EWFQ scheduler tolerates packet bursts in TCP flows, therefore avoids excessive packet drops and converges quickly.

Beside fair queueing on switch, novel queue structures and operations are proposed for efficiently scheduling packets in software [30]–[32], however, these designs can not be applied on today's hardware switches yet.

III. Background and Motivation

In this section, we briefly review the WFQ packet scheduling algorithm and the representative approximate schedulers based on AIFO and PCQ. We show that both approximate schedulers drop packet excessively, and analyze how WFQ packet scheduling impacts TCP flows.

A. Weighted Fair Queueing

Weighted fair queueing (WFQ) is a packet scheduling policy that enforces the generalized processor sharing (GPS) principle among network flows [1]. Specifically, consider a set $\mathbf{F}$ of backlogged flows sharing a switch egress port, where each flow $f \in \mathbf{F}$ is assigned with a weight w_f. If the port has a bandwidth of R, then WFQ allocates a bandwidth of $R_f = R \times \frac{w_f}{\sum_{f \in \mathbf{F}} w_f}$ to flow f.

```
1: on a $pkt$ of flow $f$ arrival:
       //compute $pkt$'s virtual finish time
2:     $f.finish\_time = \max(f.finish\_time, r) + pkt.size/R_f$ ;
3:     $pkt.finish\_time = f.finish\_time$ ; enqueue( $pkt$ );

4: while port is idle and buffer is not empty:
5:     select $pkt$ in buffer with smallest virtual finish time;
6:     dequeue( $pkt$ ); $r = r + pkt.size/R$ ;
```

Fig. 1. The WFQ packet scheduling algorithm.

Fig. 1 presents the WFQ packet scheduling algorithm [13]. The algorithm maintains a *virtual time* r, and for each arrived packet, the algorithm computes its *virtual finish time* as the maximum of the current virtual time r and the virtual finish time of the flow's last enqueued packet, plus the time required to transmit the arrived packet at rate R_f (line 2). When the port is idle, the algorithm selects the packet with the earliest virtual finish time in the buffer to dequeue, and increments the virtual time by $\frac{pkt.size}{R}$ (line 4-6). The algorithm achieves *weighed max-min fairness* and is *work-conserving*, i.e., unused bandwidth share of a flow can be allocated to other flows.

The WFQ algorithm can be faithfully realized on novel queue abstractions such as Push-In First-Out (PIFO) [4] by assigning a packet's rank as its virtual finish time, and we refer to the resulting scheduler as *PIFO-WFQ* in this paper. Unfortunately, PIFO requires arbitrarily sorting packets at line rate, which is challenging to implement with today's switch hardware. To overcome this problem, a number of approximate schedulers built on FIFO (First-In First-Out) queues, which are cheap and widely available in commodity hardware switches, were proposed in recent years. In the following, we briefly introduce two representative schedulers that approximate WFQ based on Admission-In First-Out (AIFO) [6] and Programmable Calender Queues (PCQ) [7] respectively.

B. AIFO-based Approximate WFQ Scheduler

AIFO [6] approximates PIFO with one single FIFO queue. Unlike PIFO that evicts high-ranked packets from the queue, AIFO proactively rejects an arrived packet if its rank is relatively high among a number of recently received packets. More specifically, AIFO maintains a dynamic threshold as

$$\frac{1}{1-k}\frac{Q-D}{Q} \tag{1}$$

where Q is the queue length, D is the queue depth (i.e., amount of data buffered in the queue), and k is a parameter for controlling the aggressiveness of packet rejection. For an arrived packet pkt, it is enqueued only when its rank quantile in a window of W recent received packets, i.e., $W.quantile(pkt.rank)$, does not exceed the threshold. Obviously, by using a packet's virtual finish time as its rank, WFQ can be approximated with AIFO, and we refer to the corresponding scheduler as *AIFO-WFQ*.

C. PCQ-based Approximate WFQ Scheduler

PCQ [7] is a practical mechanism for approximating packet scheduling algorithms with $M (M \geq 2)$ strict-priority FIFO queues. In PCQ, when the head queue with the highest priority is drained by dequeueing all its packets, it is paused and assigned with the lowest priority; while the previous second-highest-priority queue becomes the new head queue, and starts to dequeue packets. Such an operation is called a *rotation*.

As described in [7], PCQ can be used to approximate WFQ packet scheduling, and we refer to such a scheduler as *PCQ-WFQ*. When receiving a packet of a flow f, the PCQ-WFQ scheduler computes the index n of the queue to which the packet should be enqueued as

$$n = \lfloor (C_f + pkt.size)/(Q \times w_f) - r \rfloor \tag{2}$$

where Q is the queue length, w_f is the flow weight, r is the current round, and $C_f = \max(B_f, r \times Q \times w_f)$ is the maximum of the bytes the flow has ever enqueued and the bytes it is eligible to enqueue in the past r rounds. Note that in PCQ, queues are indexed between 0 and $M-1$ with queue 0 as the head queue, and when the queue index n computed by (2) is greater than $M-1$, the packet is dropped. Round r is incremented by 1 each time a rotation happens.

D. Excessive Packet Drops

From the above introduction one can see that both the ideal and the approximate WFQ schedulers enforce bandwidth

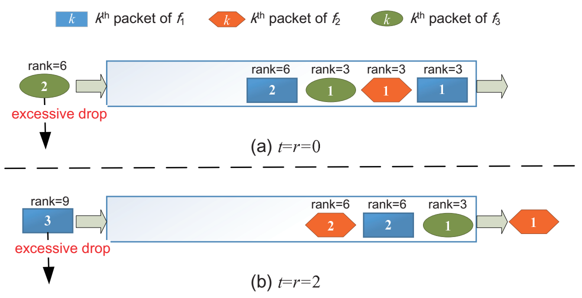


Fig. 2. An example of excessive packet drops made by AIFO-WFQ.

allocation by selectively dropping packets. More specifically, in PIFO-WFQ, higher-ranked packets would be evicted from the queue to make room for a lower-ranked packet; an arrived packet would be proactively rejected by AIFO-WFQ; and in PCQ-WFQ, if an arrived packet can not find a queue to be enqueued to, it will be dropped.

Moreover, after carefully examining AIFO-WFQ and PCQ-WFQ, we find that they make *excessive packet drops*. To show this, consider an example as in Fig. 2, in which 3 flows, i.e., f_1, f_2, and f_3, share an egress port with their weights as $w_1 = w_2 = w_3 = \frac{1}{3}$. Packets are equal-sized and the port can transmit one packet per unit time, i.e., $R = 1$. The queue has a length of $Q = 6$ packets, and is managed by AIFO with the parameters of $W = 4$ and $k = 0.2$. From Fig. 2(a), one can see that for the 2^{nd} packet of f_3, its rank quantile among the 4 recent arrived packets is $W.quantile(6) = 0.75$, which is greater than the threshold of $\frac{1}{1-k}\frac{Q-D}{Q} = 0.417$ according to (1), and the packet is dropped. The 3^{rd} packet of f_1, which arrives at $t = 2$ as shown in Fig. 2(b), is also dropped, as its rank is greater than the ranks of all the $W = 4$ recently arrived packets, and $W.quantile(9) = 1$ is greater than the threshold of 0.625.

However, both packets are dropped excessively. If we replace the AIFO queue with PIFO, then when the 2^{nd} packet of f_3 arrives, the queue has spare space to buffer it, and all the packets with smaller ranks, which would evict this packet potentially, have already been enqueued. Similarly, the 3^{rd} packet of f_1 will not be evicted either, as when it arrives, the queue has enough space to buffer it, and all the packets with smaller ranks have already arrived.

PCQ-WFQ also makes excessive packet drops, to show this, consider an example in Fig. 3, in which a PCQ is composed of $M = 2$ strict-priority queues, and each queue has a length of $Q = 4$ packets. Consider a flow f with a weight of $w_f = \frac{1}{3}$, and the flow sends a packet to the switch in every 3 time units. According to the WFQ policy, all the packets of f should be enqueued as the flow demands exactly $\frac{1}{3}$ of the port bandwidth. In Fig. 3(a), when $t = 0$ and $r = 0$, the 2^{nd} packet of flow f is enqueued to the queue indexed at 1, as $n = \lfloor (C_f + pkt.size)/(Q \times w_f) - r \rfloor = \lfloor \frac{6}{4} - 0 \rfloor = 1$ according to (2). After 3 time units when the 3^{rd} packet of the flow arrives, r is still 0 as no rotation happens, but the packet has to be dropped, because at this time $n = \lfloor (C_f + pkt.size)/(Q \times w_f) - r \rfloor = \lfloor \frac{9}{4} - 0 \rfloor = 2$, which is greater than $M - 1 = 1$, as we can see in Fig. 3(b).

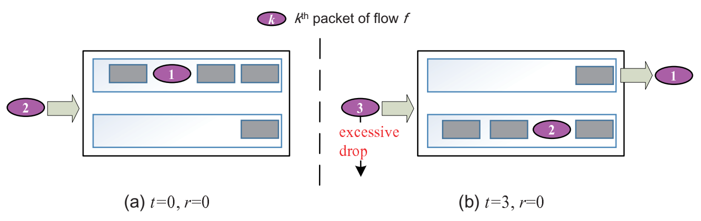


Fig. 3. An example of excessive packet drop made by PCQ-WFQ.

E. Impact of WFQ Packet Scheduling on TCP

When scheduling network flows by dropping packets selectively, an implicit assumption made by WFQ schedulers is that packets are independent to each other, and dropping a packet will not impact arrivals of the flow's subsequent packets. Such an assumption is reasonable for UDP flows without end-host rate control, but for TCP flows, which have end-to-end congestion controls, the assumption no longer holds.

In TCP, a sender maintains a congestion window (cwnd), and typically sends an entire window of packets in a burst. After sending a burst of packets, the sender waits for feedbacks, which could be either acknowledgments (ACKs) from the receiver or timeouts, adjusts its cwnd accordingly, and sends (resends) packets in another burst.

When scheduling TCP flows, a WFQ scheduler may drop a flow's packet despite that the flow has not achieved its fare share bandwidth. This is because when a burst of packets from a TCP flow arrive to the switch, the "extra" packets in the burst that exceed the flow's fair share buffering space in the queue are likely to be dropped. To show this, consider an example in which a TCP flow f has a weight of $w_f = 0.05$ and shares a port with a bandwidth of R =10 Gbit/s with other flows. The queue size is $Q = 100$ packets, so the flow's fair share buffering space in the queue is $Q \times w_f = 5$ packets. If the flow has an RTT (round trip time) of 600 µs, its cwnd must be no smaller than 25 packets for ensuring that its data rate, which is $\frac{\text{cwnd}}{\text{RTT}}$, can reach to its fair share bandwidth of $R \times w_f$ =500 Mbit/s. But on the other hand, the WFQ scheduler may start to drop the flow's packet when the burst size exceeds its fair share buffering space in the queue, which is 5 packets in this example.

To make things worse, packet loss is considered as a signal of congestion by TCP, and can significantly impact a TCP sender's subsequent behavior. For example, in the congestion avoidance state, after a packet loss, if the sender receives sufficiently number of duplicate or selective ACKs, it enters into the fast retransmit state and halves its cwnd, but when a packet is lost without triggering fast retransmit, the sender must wait until its retransmission timer expires, which would be much longer than RTT [33], without sending any data. Moreover, the sender will reduce its cwnd to as small as 1 MSS after an retransmission timeout (RTO). Note that in both cases, the sender will take many RTTs to recover its original window size, during which the flow can not maintain its original throughput.

How a TCP sender reacts to packet losses depends on many factors, such as the sender's congestion control algorithm, end-to-end RTT, etc., therefore, different TCP flows are impacted by WFQ packet scheduling differently, which arises an unfairness issue. In Sec. V, we learn from experiments that WFQ packet scheduling is unfair against TCP flows with small weights, TCP flows with longer RTTs, TCP flows applying less aggressive congestion control algorithms, and small-to-medium sized TCP flows. Moreover, when employing AIFO-WFQ or PCQ-WFQ, fairness is further sabotaged, due to the excessive packet drops made by the two schedulers. As far as we know, we are the first to address the unfairness issue of WFQ packet scheduling on TCP flows.

IV. Our Proposed WFQ Packet Schedulers

A. Design Objective

Motivated by our analysis in Sec. III, in this section, we seek to develop approximate WFQ packet scheduling algorithm that fulfills the following objectives:

- (**O1**): When scheduling network flows without rate control, the scheduler should achieve weighted max-min fairness and be work-conserving.
- (**O2**): The scheduler should reduce the excessive packet drops made by AIFO-WFQ and PCQ-WFQ.
- (**O3**): The scheduler should be adaptive to the bursty TCP traffic and fairly allocate bandwidths to TCP flows.
- (**O4**): The scheduler should be practical to implement on existing commodity hardware switches, schedule packets at line rate, and use as few as one single FIFO queue per port.

We achieve our goals in two steps. In the first step, we present SQ-WFQ, an approximate WFQ scheduler that realizes weighted max-min fairness and is work-conserving (**O1**). SQ-WFQ also reduces excessive packet drops (**O2**) and employs one single FIFO queue in packet scheduling (**O4**). Extended from SQ-WFQ, we then propose an algorithm named SQ-EWFQ that inherits all the merits of SQ-WFQ, and enhances the fairness when scheduling TCP flows (**O3**).

B. SQ-WFQ: WFQ Packet Scheduler with Single Queue

In this section, we propose an approximate WFQ packet scheduling algorithm named *SQ-WFQ*, and present it in Fig. 4.

```
1: on a $pkt$ of flow $f$ arrival:
2:     $C_f = \max(B_f, r \times R \times w_f)$;
       //enqueueing condition
3:     if $(C_f + pkt.size)/(R \times w_f) - r \le Q/R$
4:         enqueue( $pkt$ ); $B_f = C_f + pkt.size$;
5:     else
6:         drop( $pkt$ );

7: on a $pkt$ dequeued:
8:     $r = r + \frac{pkt.size \times Q/D}{R}$;
```

Fig. 4. The SQ-WFQ packet scheduling algorithm.

As its name suggests, SQ-WFQ employs one single FIFO queue, and only makes decisions on whether a packet should be enqueued or not on its arrival. For each backlogged flow f, the algorithm traces the amount of bytes the flow has ever enqueued with a per-flow state B_f. When a packet pkt of the flow arrives, the algorithm computes a variable C_f as the maximum of B_f and $r \times R \times w_f$ (line 2), where R is the port rate and w_f is the flow's weight, for ensuring that C_f is no smaller than the amount of bytes that flow f is eligible to enqueue in the past r rounds. The algorithm then compares $\frac{C_f + pkt.size}{w_f \times R} - r$ and $\frac{Q}{R}$, where the former is the expected time to dequeue the arrived packet if it is enqueued, and the latter is the time to drain a full queue. If the former is no greater than the latter, the packet is enqueued, otherwise, it is dropped (line 3-6).

Unlike PCQ-WFQ that increments the round value r by one per rotation, SQ-WFQ updates r more frequently on each packet dequeue (line 7). Moreover, when a packet pkt is dequeued, we increment the round value r by an amount of $\frac{pkt.size \times Q/D}{R}$, where Q is the queue length, and D is the queue depth defined as the amount of bytes buffered in the queue (line 8). Note that by scaling the packet size with $\frac{Q}{D}$, we adaptively adjust the increasing speed of r according to the queue status.

We explain the adaptive increasing of the round value r in more detail: When the queue has few packets buffered with $D \ll Q$, $\frac{Q}{D} \gg 1$, the round value r will be increased at a rate much faster than $\frac{pkt.size}{R}$. In this case, the enqueueing condition in line 3 of the algorithm, which we re-present as

$$\frac{C_f + pkt.size}{R \times w_f} - r \le \frac{Q}{R} \tag{3}$$

is easy to meet, and packets are more likely to be enqueued to prevent the port from starving.

Consider an extreme case that $D \rightarrow 0$, which means that the queue is close to empty[1]. In this case, the enqueueing condition in (3) is boiled down to enqueueing an arrived packet pkt as long as $pkt.size \le w_f \times Q$, which means that if a flow's fair share buffering space in the queue is no smaller than a packet size, a packet from the flow should be enqueued. If we consider a flow as schedulable as long as $w_f \times Q \ge MTU$, then the condition states that when the queue is close to empty, SQ-WFQ will enqueue any arrived packet from a schedulable flow, so as to prevent the port from starving.

On the other hand, when the queue is almost full with $D \rightarrow Q$, the round value r is increased at a rate of $\frac{pkt.size}{R}$, which makes the enqueueing condition in (3) difficult to meet, and the switch tends to drop packets to avoid the queue overflow.

Consider an other extreme case that $D \rightarrow Q$ and a flow f has C_f such that

$$\begin{aligned} &\left(\frac{C_f + pkt.size}{R \times w_f} - r \rightarrow \frac{Q}{R}\right) \Rightarrow \\ &\left(C_f + pkt.size \rightarrow \left(\frac{Q}{R} + r\right) \times R \times w_f\right) \end{aligned} \tag{4}$$

[1] D will never be zero as it is the queue depth at the moment a packet is dequeued, thus should be at least one packet size.

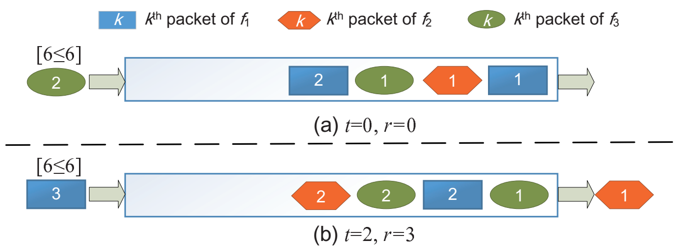


Fig. 5. Excessive packet drops avoided by SQ-WFQ comparing with the example of AIFO-WFQ in Fig. 2.

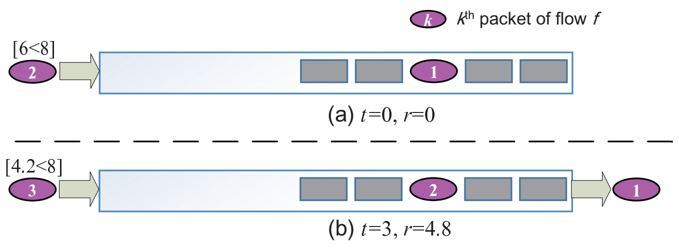


Fig. 6. Excessive packet drop avoided by SQ-WFQ comparing with the example of PCQ-WFQ in Fig. 3.

After a packet is dequeued, r is incremented by $\frac{pkt.size}{R}$, and the right side of "→" in (4) is increased by $w_f \times pkt.size$. To keep the condition in (3) hold, C_f at the left side of "→" can be incremented by no more than $w_f \times pkt.size$. In other words, the scheduler has to wait for the switch to dequeue over $\frac{1}{w_f}$ packets before it can enqueue a new packet from flow f, while any earlier arrived packet must be dropped. From the analysis, we can see that the adaptive increasing of the round value r can automatically tune the difficulty of the enqueueing condition, and prevent the queue from either starvation or overflow.

SQ-WFQ reduces the excessive packet drops that we have seen in AIFO-WFQ and PCQ-WFQ. To show this, reconsider the example in Fig. 2, in which we replace AIFO-WFQ with SQ-WFQ to manage the FIFO queue, and re-present the example in Fig. 5. As we can see in Fig. 5(a), when the 2nd packet of f_3, which is excessively dropped by AIFO-WFQ, arrives to the switch, we have $r = 0$, and the left side of the enqueueing condition in (3) is $\frac{C_f + pkt.size}{w_f \times R} - r = 6$, which is no greater than the right side of $\frac{Q}{R} = 6$. The packet is enqueued. Similarly, when the 3rd packet f_1 arrives at $t = 2$, r is increased to 3, as two packets have been dequeued with $D = 4$, and each packet dequeue increments r by $\frac{Q}{D} = 1.5$. As we can see in Fig. 5(b), the enqueueing condition still holds with $\frac{C_f + pkt.size}{w_f \times R} - r = 6 \le \frac{Q}{R} = 6$, and the packet is also enqueued despite that it is excessively dropped by AIFO-WFQ.

To compare SQ-WFQ with PCQ-WFQ, we reconsider the example in Fig. 3, but replace PCQ with a single FIFO queue of equal size (i.e., $Q = 8$) managed by the SQ-WFQ algorithm. As shown in Fig. 6(a), at $t = 0$ when the 2nd packet of flow f arrives, since no packet has been dequeued, we have $r = 0$, and the enqueueing condition in (3) is satisfied with $\frac{C_f + pkt.size}{w_f \times R} - r = 6 < \frac{Q}{R} = 8$. The packet is enqueued. After 3 time units when the 3rd packet of flow f arrives, the queue depth D remains 5, but the round value r is increased to $3 \times \frac{Q}{D} = 4.8$, as 3 packets have been dequeued, and each packet dequeue increments r by $\frac{Q}{D} = 1.6$. Note that for this packet, the enqueueing condition in (3) still holds with $\frac{C_f + pkt.size}{w_f \times R} - r = 9 - 4.8 < \frac{Q}{R} = 8$, and as shown in Fig. 6(b), the packet is enqueued by the SQ-WFQ algorithm, even though it is excessively dropped by PCQ-WFQ.

From the above analysis and comparisons, we can see that unlike AIFO-WFQ that drops high-ranked packets based on local comparisons, SQ-WFQ carefully maintains two variables of C_f and r, which accumulate all the previous enqueueing events of the flow and dequeueing events of the queue, and compares $(C_f + pkt.size)$ with $(r \times R \times w_f)$ to make the scheduling decisions more precisely. Unlike PCQ-WFQ, which updates the round value r per rotation, SQ-WFQ updates r more frequently per packet dequeue, thus can make more accurate decisions with up-to-date information. As we will see in Sec. V, SQ-WFQ reduces the excessive packet drops that happen in AIFO-WFQ and PCQ-WFQ, and is capable to schedule packets more fairly than them.

C. SQ-EWFQ: Enhancing Fairness with Burst Tolerance

```
on a pkt of flow f arrival:
    $s_f = (n_f \times s_f + pkt.size)/(n_f + 1)$; $n_f = n_f + 1$;
    $\tau_f = \theta \times \tau_f + (1-\theta) \times (t_{now} - t_f)$; $t_f = t_{now}$;
    $\alpha_f = \max(\rho \times (s_f/\tau_f)/(R \times w_f), 1)$;
    $C_f = \max(B_f, r \times R \times w_f)$;
    //enqueueing condition
    if $(C_f + pkt.size - r \times w_f \times R)/(R \times \min(1, \alpha_f \times w_f)) \le Q/R$
        enqueue( pkt ); $B_f = C_f + pkt.size$;
    else
        drop( pkt );

on a pkt dequeued:
    $r = r + \frac{pkt.size \times Q/D}{R}$;
```

Fig. 7. The SQ-EWFQ packet scheduling algorithm.

As we have discussed in Sec. III-E, the SQ-WFQ scheduling algorithm, as well as the other WFQ schedulers, has an unfairness issue on TCP. In this section, we extended SQ-WFQ and propose an approximate WFQ packet scheduling algorithm named *SQ-EWFQ*. SQ-EWFQ inherits all the merits of SQ-WFQ by requiring only one single FIFO queue and reducing excessive packet drops. Moreover, comparing with SQ-WFQ, SQ-EWFQ is more adaptive to the bursty TCP traffic and allocates bandwidths more fairly to TCP flows.

The basic idea of SQ-EWFQ is simple: When a TCP flow is detected to start sending a burst of packets, the scheduler temporarily increases the flow's weight, so as to enable it to grow its cwnd to an extent larger than the one under the SQ-WFQ algorithm. The larger cwnd will allow the flow to transmit a burst of packets at a higher chance of success without being interrupted by packet drops. SQ-EWFQ enforces the weighted fairness by constraining the total amount of bytes a flow can enqueue over an interval that is sufficiently long for covering a burst duration.

More specifically, for each backlogged flow f, SQ-EWFQ maintains five per-flow states, denoted as B_f, n_f, s_f, τ_f, and t_f, which we describe as the following.

- B_f is the state that traces the bytes that flow f has ever enqueued, as we have seen in the SQ-WFQ algorithm.
- n_f is the number of flow f's packets that have arrived to this switch.
- s_f is the averaged packet size of flow f.
- τ_f is the exponential moving averaged (EMA) packet inter-arrival time of flow f.
- t_f is the timestamp of flow f's last arrived packet.

Besides B_f, the other four states actually trace a flow's packet arrival pattern, and are used to detect packet bursts.

For each flow f, the scheduler computes its data arrival rate as $\frac{s_f}{\tau_f}$, and compares it with the flow's fair share bandwidth, i.e., $R \times w_f$. If the former is greater than the latter, which suggests that the flow is sending a burst of packets and demands a higher instant rate than its fair share bandwidth, the scheduler temporarily increases f's weight to $\alpha_f \times w_f$. Here $\alpha_f = \max(\rho \times \frac{s_f/\tau_f}{R \times w_f}, 1)$ is a scaling factor computed by dividing f's data arrival rate s_f/τ_f with $R \times w_f$, and we use a parameter ρ $(0 < \rho \leq 1)$ to control the aggressiveness. Note that when $\rho = 1$, the algorithm would increase f's rate to s_f/τ_f, which means that the switch will temporarily send the flow's packets at the rate that they arrive, regardless of the flow weight.

SQ-EWFQ employs the following packet enqueueing condition

$$\frac{C_f + pkt.size - r \times w_f \times R}{R \times \min(1, \alpha_f \times w_f)} \leq \frac{Q}{R} \tag{5}$$

The condition constrains the difference between $C_f + pkt.size$ and $r \times w_f \times R$, where the former is the amount of data enqueued by f (plus the arrived pkt), and the latter is the amount of data that f is eligible to enqueue in the past r rounds. Note that the difference is divided by $R \times \min(1, \alpha_f \times w_f)$, which is temporarily increased during a burst, but falls back to $R \times w_f$ afterwards. As a consequence, the condition ensures that over a period of time that is sufficiently long to cover a burst, flow f can enqueue more data than its weight w_f allows, but after the burst, the total bytes it has enqueued still needs to be constrained by $r \times w_f \times R$. In other words, SQ-EWFQ enforces a *long-term weighted fairness* rather than a short-term one among the TCP flows.

We present the SQ-EWFQ algorithm in Fig. 7. In the algorithm, when a packet of flow f arrives, the per-flow states of s_f, n_f, τ_f, and t_f are updated (line 2-3), and the algorithm computes the scaling factor α_f with the updated s_f and τ_f (line 4). Note that when updating τ_f, we use θ as the weight in the exponential moving averaging. The enqueueing condition (line 6) enforces a long-term weighted fairness. Finally, as in SQ-WFQ, the round value r is incremented by $\frac{pkt.size \times Q/D}{R}$ on each packet dequeue (line 10-11).

D. Implementation

We use Intel's Barefoot Tofino Ethernet switch [11] as the hardware platform, and implement the SQ-WFQ and SQ-EWFQ schedulers with the P4$_{16}$ programming language [12]. Implementing SQ-WFQ and SQ-EWFQ on Tofino-based switch is non-trivial, and in this section, we describe how we overcome the challenges in the implementations.

1) Handling Multiplication and Division Operations: In a Tofino-based switch, an incoming packet first goes through an ingress pipeline, where a scheduling decision is made by the SQ-WFQ or SQ-EWFQ algorithm on whether the packet should be enqueued or not. When making the decision, both SQ-WFQ and SQ-EWFQ incur multiplication and division operations. Unfortunately, the Tofino-based switch does not support multiplication and division on arbitrary values. To overcome this problem, we apply the following methods.

Our first method is to replace divisions with multiplications. For example, in SQ-WFQ, when evaluating the enqueueing condition

$$\frac{\max(B_f, r \times R \times w_f) + pkt.size}{R \times w_f} - r \leq \frac{Q}{R} \tag{6}$$

We multiply both sides of the inequality with $R \times w_f$, so that the condition becomes

$$\max(B_f, r \times R \times w_f) + pkt.size - r \times R \times w_f \leq Q \times w_f \tag{7}$$

One can see that when evaluating (7), no division is required.

Our second method is to replace multiplications with arithmetic bit-shift operations. To this end, we require flow weight to be power of 2 (or sum of powers of 2). For example, in the experiment in Sec. V-A1, there are four flows with weights as $w_1 = 8$, $w_2 = 4$, $w_3 = 2$, and $w_4 = 1$, but within the switch, the actual weights are $w_1 = \frac{1}{2}$, $w_2 = \frac{1}{4}$, $w_3 = \frac{1}{8}$, and $w_4 = \frac{1}{16}$[2]. In addition, the constant of port bandwidth R is also rounded as power of 2. As a result, we can use left or right bit-shift operations, which are supported by Tofino, to replace the multiplications. For example, to evaluate the condition in (7), since w_f and R are powers of 2, only arithmetic bit-shift operations are required.

Our third method is to use lookup table to approximate multiplication. For example, in SQ-EWFQ, to compute

$$\alpha_f \times w_f \times R = \max(\rho \times \frac{s_f}{\tau_f}, w_f \times R) \tag{8}$$

we make two observations: First, for a flow f to be schedulable, its weight w_f should be no smaller than $w_{f,\min} = \frac{MTU}{Q}$, otherwise, the fair share buffering space of f in the queue is smaller than MTU. Our second observation is that, for a given specific value of s_f, there exists an upper bound of τ_f for satisfying $\rho \times \frac{s_f}{\tau_f} \geq w_{f,\min} \times R$.

With the observations, we compute the right side of (8) as the following. We first find the upper and lower bounds of s_f as $s_f^+ = MTU$ and $s_f^- = 0$, and equally divide the range $[s_f^-, s_f^+]$ into M sub-ranges, with the i^{th} sub-range as $[s_f^- + i \times \Delta s_f, s_f^- + (i+1) \times \Delta s_f]$, where $\Delta s_f = \frac{s_f^+ - s_f^-}{M}$. For the

[2]To efficiently utilize the queue buffer space, the sum of all the flows' weights should close to 1.

s_f	τ_f	Pre-computed value
...	...	...
$[s_f^- + i \times \Delta s_f, s_f^- + (i+1) \times \Delta s_f]$	$[\tau_{f,i}^- + j \times \Delta\tau_f, \tau_{f,i}^- + (j+1) \times \Delta\tau_f]$	$\max\left(\rho \times \frac{s_f^- + (i+0.5) \times \Delta s_f}{\tau_{f,i}^- + (j+0.5) \times \Delta\tau_f}, w_f \times R\right)$
...	...	...

Fig. 8. Using range-match lookup table to approximate multiplication.

central value $s_{f,i} = s_f^- + (i + 0.5) \times \Delta s_f$ of the i^{th} sub-range, we find the upper and lower bounds of $\tau_{f,i}$ as $\tau_{f,i}^+$ and $\tau_{f,i}^-$, where $\tau_{f,i}^- = 0$ and $\tau_{f,i}^+$ is obtained by applying our second observation as above described. Similarly, we divide the range $[\tau_{f,i}^-, \tau_{f,i}^+]$ into N_i sub-ranges, with the j^{th} sub-range as $[\tau_{f,i}^- + j \times \Delta\tau_f, \tau_{f,i}^- + (j+1) \times \Delta\tau_f]$, where $\Delta\tau_f = \frac{\tau_{f,i}^+ - \tau_{f,i}^-}{N_i}$. As shown in Fig. 8, we place a range-match lookup table in the ingress pipeline to enumerate all the combinations of the sub-ranges. After obtaining s_f and τ_f from a new packet, the two values are range-matched in the lookup table, and the action of the matched table entry writes the pre-computed value of (8) to the packet's metadata as the computation result.

2) Computing Round Value r: When making decision on whether to enqueue or drop a packet, both algorithms of SQ-WFQ and SQ-EWFQ require the knowledge of the round value r, which is computed with the queue depth D that dynamically changes over time. Unfortunately, queue depth is maintained by a module called traffic manager (TM) between the ingress and egress pipelines, and a packet can record the queue depth D in its metadata only after its has been dequeued and enters into the egress pipeline.

In our implementation, for each switch port, we use a register in its egress pipeline for maintaining the round value r, which is updated each time a packet is dequeued. We use a lookup table as above described to perform the $\frac{pkt.size \times Q}{R \times D}$ computation. Meanwhile, we maintain a copy for the register in the ingress pipeline, and execute the packet scheduling algorithms with the value in the copy register.

To synchronize the ingress register to the egress one, we exploit the *recirculation* mechanism of the Tofino-based switch. More specifically, we send a special packet called *carrier packet*, and recirculate it within the switch. When the carrier packet arrives to the egress pipeline, it reads the register and carry the round value r in its payload. The egress pipeline recirculates the carrier packet to the ingress pipeline, which writes the the round value r that the carrier packet carries to the copy register.

Using carrier packet to compute r is efficient. Since carrier packet does not queue with data packets and goes through a dedicated port, it is recirculated very fast within the switch. In our implementation, a recirculation takes shorter time than transmitting an MTU-sized packet, or in other words, r is updated more frequently than its value changes, and such a capability guarantees the accuracies of the scheduling decisions made by the SQ-WFQ and SQ-EWFQ algorithms.

3) Computing (Moving) Average: The SQ-EWFQ algorithm uses s_f and τ_f to detect whether a flow is sending a burst of packets, where s_f is the averaged packet size and τ_f is the exponential moving average (EMA) of the packet inter-arrival time. We place a ring buffer as in [6] in the ingress pipeline to compute s_f and τ_f.

TABLE I
SUMMARY OF RESOURCE USAGES OF OUR SQ-WFQ AND SQ-EWFQ IMPLEMENTATIONS ON TOFINO-BASED SWITCH, AND THE RESOURCE USAGE OF PCQ-WFQ REPORTED BY [7].

Resource Type	SQ-WFQ	SQ-EWFQ	PCQ-WFQ
Pipeline Stages	6	12	12
Match Crossbars	61	107	63
Hash Bits	263	420	140
SRAM	32	57	46
TCAM	6	12	2
ALU Instructions	3	13	13

The ring buffer is composed of 5 records for keeping the sizes and arrival times of the 5 recent packets. We also maintain an index to indicate the current head position of the ring buffer. When a new packet arrives, it reads and updates the index, and updates the ring buffer with its own size and arrival time by over-writing the earliest record.

The averaged packet size s_f and the EMA of the inter-arrival time τ_f are computed by the arriving packet from the 5 recent packets. For example, with $\theta = 0.5$, τ_f is computed as

$$\begin{aligned}\tau_f &= 0.5 \times (t_{f,1} - t_{f,2}) + 0.25 \times (t_{f,2} - t_{f,3}) \\ &+ 0.125 \times (t_{f,3} - t_{f,4}) + 0.125 \times (t_{f,4} - t_{f,5})\end{aligned} \tag{9}$$

where $t_{f,i}$ is arrival time of the i^{th} recent packet.

4) Resource Overhead: Table I shows the overheads of implementing SQ-WFQ and SQ-EWFQ on the Tofino-based switch reported by the P4 compiler. We also consult [7] and list the implementation overhead of PCQ-WFQ in the table as well. From the table one can see that SQ-EWFQ requires more resources than SQ-WFQ as it involves more states and incurs more computations.

Finally, we stress that our proposed SQ-WFQ and SW-EWFQ algorithms are independent of the hardware architecture, and believe that the implementation presented here is valuable for implementing them on other platforms.

V. EVALUATION

In this section, we evaluate our proposed SQ-WFQ and SQ-EWFQ packet scheduling algorithms and compare them with the state-of-the-art solutions. We carry out the experiments on both a real-world testbed based on commodity hardware programmable switches and `Netbench` [34], [35], a packet-level simulator.

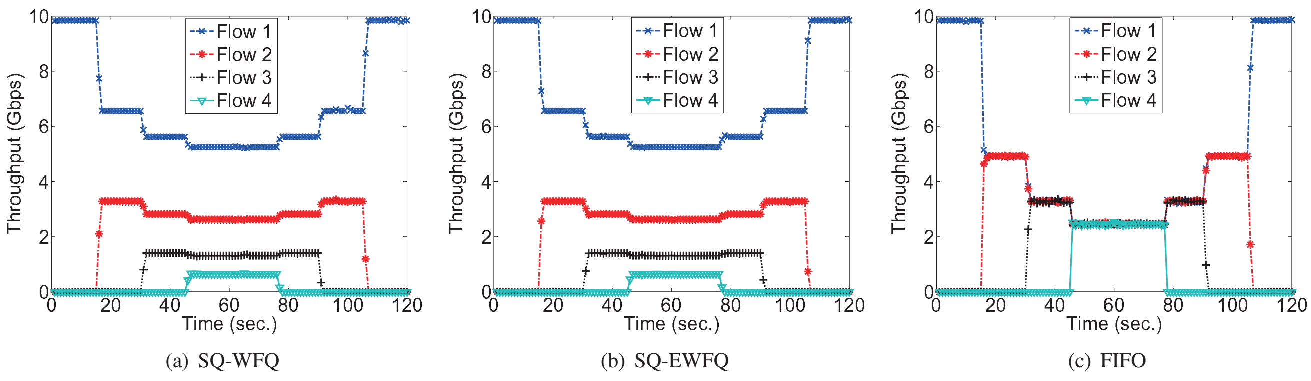


Fig. 9. Evolvements of UPD flows' throughputs over time when the flows are scheduled by (a) SQ-WFQ, (b) SQ-EWFQ, and (c) FIFO. The four flows have their weights set as $w_1 = 8$, $w_2 = 4$, $w_3 = 2$, and $w_4 = 1$.

A. Evaluation on Hardware Testbed

Our real-world testbed is composed of one Edgecore Wedge 100BF-32X Tofino-based hardware switch and 4 servers connecting to the switch. Each server is equipped with a 8-core Intel i7-8700 CPU and an Intel X710-DA2 10GbE dual-port Ethernet adapter, and runs Ubuntu 16.04 LTS with Linux kernel 4.14.24. The Tofino-based switch runs our P4 implementation of the SQ-WFQ or SQ-WFQ packet scheduler, and we configure the line rate of the switch port as 10 Gbit/s. We set $\theta = 0.5$ and $\rho = 0.9$ for the SQ-EWFQ algorithm, if not otherwise specified.

1) Weighted Fairness and Work-Conserving: We first examining the SQ-WFQ and SQ-EWFQ algorithms on scheduling UDP flows. For this purpose, we send 4 UDP flows using `iPerf3` [36] at a rate of 9.8 Gbit/s to the Tofino-based switch, and set the weights of the flows as $w_1 = 8$, $w_2 = 4$, $w_3 = 2$, and $w_4 = 1$. Moreover, different flows start and end at different times: flow f_1 starts at time 0 and lasts for 120 seconds, flow f_2 starts at the 15^{th} second and lasts for 90 seconds, flow f_3 starts at the 30^{th} second and lasts for 60 seconds, and flow f_4 starts and ends at the 45^{th} and 75^{th} seconds respectively.

Fig. 9(a) and (b) present the evolvements of the flows' throughputs over time when they are scheduled by SQ-WFQ and SQ-EWFQ respectively, and we also present the throughputs when the switch employs a FIFO queue without applying any packet scheduling algorithm in Fig. 9(c). From Fig. 9(a) and (b), we can see that both SQ-WFQ and SQ-EWFQ schedule packets at line rate and achieve a weighted max-min fairness, as each of the concurrent flows achieves a throughput proportional to its weight, and their total throughput saturates the port bandwidth without starving the port. On the contrary, from Fig. 9(c), one can see that when applying FIFO, the concurrent UDP flows equally divide the port bandwidth regardless of their weights.

Fig. 9(a) and (b) also show that both SQ-WFQ and SQ-EWFQ are work-conserving, as when a UDP flow ends, its unused bandwidth is proportionally allocated among the remaining flows according to their weights.

2) Bandwidth Allocation for TCP Flows:

a) Scheduling TCP flows with different weights: In the remaining experiments on the real-world testbed, we use SQ-WFQ and SQ-EWFQ to schedule TCP flows. We first send 6 TCP flows to the Tofino-based switch, and set the weights for the flows f_1-f_4 as 1, and for the flows f_5 and f_6 as 2. The TCP senders run NewReno [37] for congestion control.

Fig. 10 presents the averaged throughputs of the 6 flows scheduled by SQ-WFQ, SQ-EWFQ, and FIFO in 60 seconds. From the figure we can see that SQ-WFQ is unfair against the smaller-weighted TCP flows, i.e., f_1-f_4, as the throughput achieved by f_5 or f_6 is as much as over 6 times of the ones achieved by f_1-f_4. However, when applying SQ-EWFQ, bandwidth is allocated to the 6 flows approximately proportional to their weights, and the smaller-weighted TCP flows, i.e., f_1-f_4, obtain fairer bandwidth allocations.

We explain the improved fairness of SQ-EWFQ over SQ-WFQ as the following: When scheduled by SQ-WFQ, a TCP flow with a smaller weight is more often to have its packet burst size exceeding its fair share buffering space in the queue, thus experiences more packet drops than a larger-weighted TCP flow, and it is well-known that a higher packet loss rate leads to greater throughput reductions. One the other hand, when scheduled by SQ-EWFQ, both flows can grow their cwnds without being frequently interrupted by packet drops, and the amount of bytes a flow can successfully enqueue is largely decided by its weight rather than the end-host congestion control.

b) Scheduling TCP flows with different RTTs: We also apply SQ-WFQ, SQ-EWFQ, and FIFO to schedule TCP flows with different RTTs. To this end, we send 8 TCP flows to the Tofino-based switch, where the flows have equal weights but different RTTs. In particular, flows f_1-f_4 have an RTT of 3 ms and flows f_5-f_8 have an RTT of 7 ms. We use Linux `tc` [38] to add latency to the TCP flows.

Fig. 11 presents the averaged throughputs of the 8 flows scheduled by SQ-WFQ, SQ-EWFQ, and FIFO in 60 seconds. One can see that under SQ-EWFQ, TCP flows obtain bandwidths proportional to their weights and have similar throughputs regardless of their RTTs. But under SQ-WFQ and FIFO, f_1-f_4 have much higher throughputs than f_5-f_8, and to our surprise, SQ-WFQ is more biased towards f_1-f_4 than FIFO. We explain the observation with the fact that when a TCP flow has a longer RTT, its cwnd grows slowly. Comparing with FIFO, the SQ-WFQ scheduler, which has no tolerance to packet bursts, would be more likely to drop packets that arrive

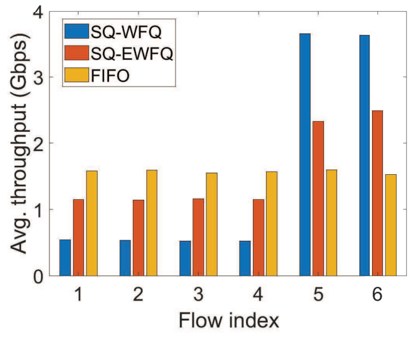


Fig. 10. Averaged throughputs of 6 TCP flows scheduled by SQ-WFQ, SQ-EWFQ, and FIFO, where f_1-f_4 have a weight of 1, and f_5-f_6 have a weight of 2.

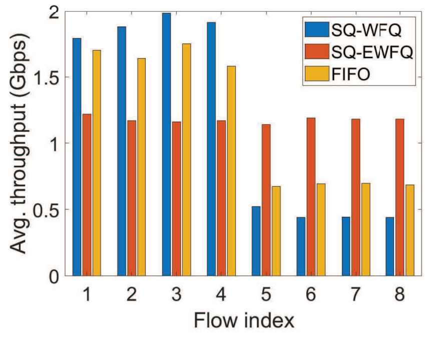


Fig. 11. Averaged throughputs of 8 TCP flows scheduled by SQ-WFQ, SQ-EWFQ, and FIFO, where f_1-f_4 have an RTT of 3 ms, and f_5-f_8 have an RTT of 7 ms. Flows have equal weights.

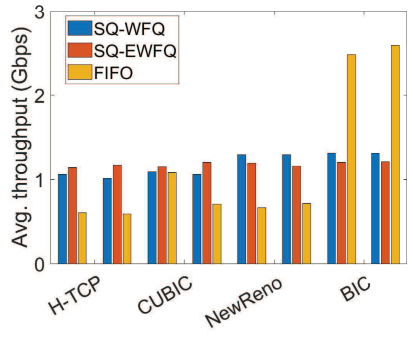
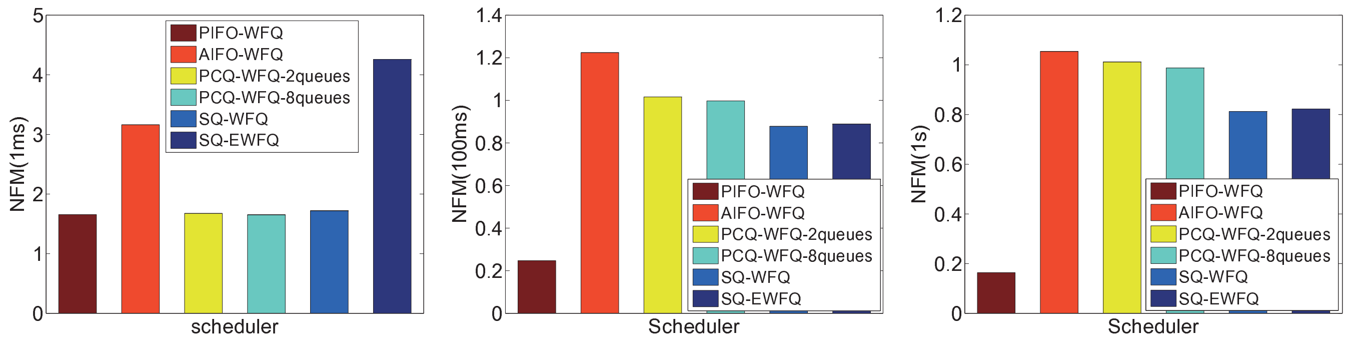


Fig. 12. Averaged throughputs of 8 equal-weighted TCP flows scheduled by SQ-WFQ, SQ-EWFQ, and FIFO, where f_1-f_2 apply H-TCP, f_3-f_4 apply CUBIC, f_5-f_6 apply NewReno, and f_7-f_8 apply BIC as congestion controls.



(a) NFM(1 ms) (b) NFM(100 ms) (c) NFM(1 s)

Fig. 13. Normalized Fairness Metrics (NFMs) of the PIFO-WFQ, AIFO-WFQ, PCQ-WFQ-2queues, PCQ-WFQ-8queues, SQ-WFQ, and SQ-EWFQ schedulers in durations of (a) 1 ms, (b) 100 ms, and (c) 1 s.

in bursts, thus it takes the flow with a longer RTT more time to recover. On the other hand, by temporarily increasing a TCP flow's weight, the SQ-EWFQ scheduler can better handle the bursty TCP traffic and enforce a longer-term weighted fairness, as we have discussed in Sec. IV-C.

c) Scheduling TCP flows with different congestion controls: We study how SQ-WFQ, SQ-EWFQ, and FIFO allocate bandwidths to TCP flows applying different congestion control algorithms. For this purpose, we send 8 equal-weighted TCP flows to the Tofino-based switch, and the flows are divided into 4 groups with each group containing 2 flows. We apply different congestion control algorithms of H-TCP [39], CUBIC [40], NewReno [37], and BIC [41] in different groups.

Fig. 12 presents the flows' averaged throughputs under different schedulers. We can see that when not applying any packet scheduling, i.e., FIFO, the flows using BIC have much higher throughputs than the other flows, because of BIC's aggressive cwnd growth function. When the flows are scheduled by SQ-WFQ, the BIC and NewReno flows still have higher throughputs than the H-TCP and CUBIC flows, as the latter two algorithms make efforts to avoid being overaggressive in growing their cwnds after packet losses. Finally, we can see when the TCP flows are scheduled by SQ-EWFQ, they equally divide the port bandwidth according to their weights, despite that they apply different congestion control algorithms.

B. Packet-level Simulation with a Single-Switch Network

1) Simulation Setup: In this section, we employ `Netbench` [34], [35], a packet-level simulator, to evaluate various WFQ packet scheduling algorithms. In particular, we evaluate and compare the following schedulers:

- **PIFO-WFQ**: We apply the WFQ algorithm as in Fig. 1 to assign ranks to packets, and emulate PIFO [4] with `Netbench` to realize the PIFO-WFQ scheduler. Since PIFO-WFQ faithfully implements the WFQ algorithm, we use it as the benchmark in our evaluation.
- **AIFO-WFQ**: We realize AIFO [6] with `Netbench` and apply it to approximately schedule packets with their ranks assigned by the WFQ algorithm as in Fig. 1. We set $W = 40$ and $k = 0.15$ as suggested by [6].
- **PCQ-WFQ**: This is the approximate WFQ packet scheduler realized over PCQ [7]. In particular, we evaluate two PCQ-WFQ scheduler instances with the PCQ containing 2 queues and 8 queues respectively.
- **SQ-WFQ**: This is the approximate WFQ packet scheduler that we have proposed in Sec. IV-B.
- **SQ-EWFQ**: This is the approximate WFQ scheduler that we have proposed in Sec. IV-C. We set $\theta = 0.5$ and $\rho = 0.9$ as in the testbed experiments by default.

All the packet schedulers are evaluated and compared under same per-port buffer size. That is, suppose the egress port has a buffer of B bytes, then for PIFO-WFQ, AIFO-WFQ, SQ-

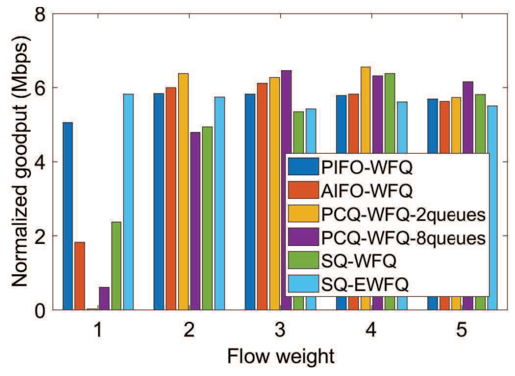


Fig. 14. Averaged normalized goodputs for TCP flows of different weights under various WFQ schedulers.

WFQ, and SQ-EWFQ that employ one single queue, the queue length is $Q = B$ bytes, while for PCQ-WFQ using $M(M \geq 2)$ queues, each queue has a length of $Q = \frac{B}{M}$ bytes.

We first simulate a single-switch topology in which 11 hosts are connected by a switch. Each host-switch link has a bandwidth of 10 Gbit/s and a propagation delay of 3 µs. We set the per-port buffer size as 2.25 MB.

We set up 500 TCP flows to send data from 10 senders to 1 receiver. The flow weights are between 1 and 5, and we choose 100 flows to have a weight of 1, 100 flows to have a weight of 2, ..., and 100 flows to have a weight of 5. We use NewReno [37] as the end-to-end TCP congestion control, and set RTO_{min} as 200 µs.

2) Weighted Fairness: We use the *Normalized Fairness Metric* (*NFM*) as defined in [30] to numerically measure the weighted fairness achieved by different packet schedulers. NFM is based on the *Fairness Metric* (*FM*) [22], which computes the maximum difference of the bytes sent by two flows, normalized by their weights during a time interval τ. NFM is the FM over all intervals of duration τ, and normalized to the data sent by the port. Note that according to its definition, a smaller NFM indicates a fairer bandwidth allocation among the flows.

Fig. 13 presents NFMs of the PIFO-WFQ, AIFO-WFQ, PCQ-WFQ (2 queues), PCQ-WFQ (8 queues), SQ-WFQ, and SQ-EWFQ schedulers over three durations of 1 ms, 100 ms, and 1 s. From the figures one can see that in general, it is easier to achieve a fairer bandwidth allocation over a longer interval. For each individual scheduler, several observations can be made: First, PIFO-WFQ is the fairest scheduler with the lowest NFMs over all the three durations, this conforms to our expectation as PIFO-WFQ faithfully realizes the ideal WFQ algorithm by evicting high-ranked packets; Second, besides PIFO-WFQ, our proposed SQ-WFQ scheduler has the lowest NFM, as it reduces the excessive packet drops made by AIFO-WFQ and PCQ-WFQ, as we have analyzed in Sec. III-E; Third, with a short duration of 1 ms, our proposed SQ-EWFQ has the highest NFM, this is reasonable as SQ-EWFQ would temporarily increase a bursty TCP flow's weight in a short interval, thus harms the short-term fairness; but with longer durations of 100 ms and 1 s, SQ-EWFQ has NFMs lower than AIFO-WFQ and PCQ-WFQ, and is only slightly higher than SQ-WFQ. Such an observation indicates that SQ-EWFQ is capable to maintain a long-term weighted fairness across the flows, as we have discussed in Sec. IV-C.

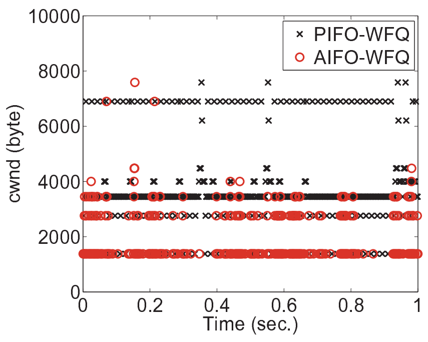
3) Bandwidth Allocation for Small-Weighted TCP Flows: To examine how different WFQ schedulers allocate bandwidths to TCP flows of different weights, we examine a flow's *normalized goodput*, which is the flow's goodput divided by its weight. Fig. 14 presents the mean normalized goodputs of the flows with different weights under various schedulers. From the figure we can see that for the flows of the smallest weight, i.e., weight 1, the schedulers of PIFO-WFQ, AIFO-WFQ, PCQ-WFQ and our proposed SQ-WFQ have obvious lower normalized goodputs than the larger-weighted flows. In particular, the PCQ-WFQ scheduler using 2 queues in PCQ barely sends any data with a goodput close to zero.

The observation in Fig. 14 can be explained with the fact that WFQ packet scheduling is unfavorable to TCP flows of small weights, especially the flows of weight 1 in this experiment. Moreover, the approximate schedulers of AIFO-WFQ and PCQ-WFQ further degrade the scheduling fairness due to their excessive packet drops. In particular, the PCQ-WFQ scheduler instance using 2 queues has a very low goodput for the smallest-weighted flows, as the scheduler updates the round value r infrequently. The stall r value of PCQ-WFQ causes continuous packet drops, which lead to more RTOs than other schedulers.

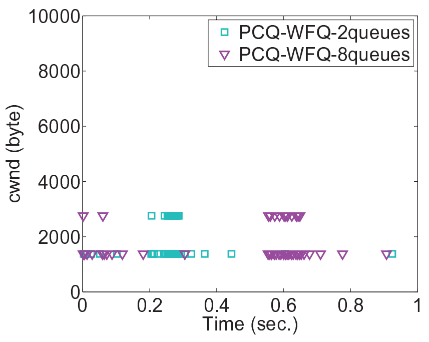
Our proposed SQ-WFQ algorithm has a normalized goodput close to the ideal PIFO-WFQ scheduler and outperforms AIFO-WFQ and PCQ-WFQ, as it substantially reduces the excessive packet drops. When scheduled by SQ-EWFQ, the smallest-weighted TCP flows have a normalized goodput close to the larger-weighted flows, and is higher than the other five schedulers. We explain the observation with the fact that SQ-EWFQ is designed to be adaptive to TCP. By temporarily increasing a bursty flow's weight, SQ-EWFQ enables a TCP flow to transfer packet bursts at a higher chance of success, rather than being frequently interrupted by packet drops as under other WFQ schedulers.

To better understand the impacts of the different schedulers on TCP flows, in Fig. 15 we present cwnd of a smallest-weighted TCP flow under each WFQ scheduler. We can see that the flows under the two PCQ-WFQ scheduler instances employing 2 and 8 queues have their cwnds much smaller than the flows under PIFO-WFQ and our proposed SQ-WFQ and SQ-EWFQ schedulers, and AIFO-WFQ also causes the flow to have a relatively smaller cwnd. On the other hand, our proposed SQ-WFQ and SQ-EWFQ enable the TCP flows to maintain larger cwnds by reducing the excessive packet drops. Moreover, one can see that although fluctuating, the cwnd of the flow under SQ-EWFQ is larger than the one under SQ-WFQ, and is even larger than PIFO-WFQ from time to time, thanks to the burst tolerance of the scheduler.

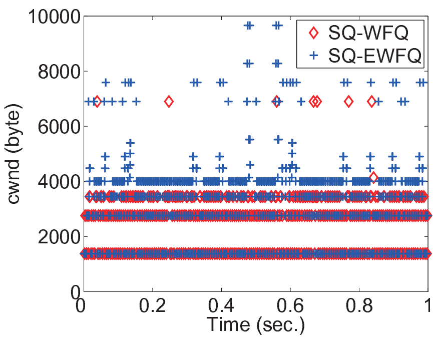
4) Impact of parameter ρ on SQ-EWFQ: As described in Sec. IV-C, our proposed SQ-EWFQ algorithm employs a parameter ρ to control the aggressiveness of the temporary weight increasing. In this experiment, we examine the impact of this parameter by experimenting various ρ values of 0.3, 0.5, 0.7, and 0.9. Note that with a larger ρ, the SQ-EWFQ



(a)

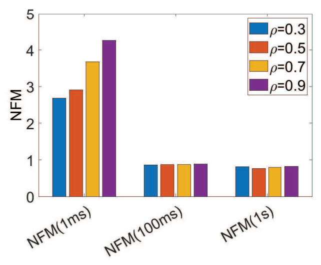


(b)

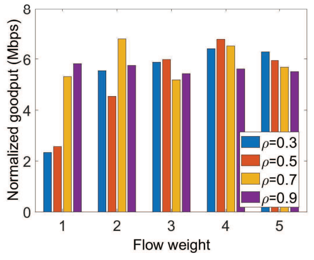


(c)

Fig. 15. Congestion window (cwnd) of TCP flows of weight 1 scheduled by (a) PIFO-WFQ and AIFO-WFQ, (b) PCQ-WFQ-2queues and PCQ-WFQ-8queues, and (c) SQ-WFQ and SQ-EWFQ.



(a) NFM



(b) Normalized goodput

Fig. 16. (a) NFMs and (b) normalized goodput of different-weighted TCP flows scheduled by SQ-EWFQ with ρ = 0.3, 0.5, 0.7, and 0.9.

scheduler increases a flow's weight more aggressively, and can tolerate larger packet bursts.

In Fig. 16(a), we present fairness of the packet scheduling in terms of NFM(1 ms), NFN(100 ms), and NFN(1 s) for SQ-EWFQ under various ρ values. We can see that within a 1 ms duration, when ρ is large, the scheduler behaves less fairly. This is reasonable as tolerating larger packet bursts will surely harm the short-term fairness of the packet scheduling. But under the longer durations of 100 ms and 1 s, SQ-EWFQ has similar NFMs under different ρ values. This is because SQ-EWFQ enforces a long-term weighted fairness with its enqueueing condition, as we have discussed in Sec. IV-C.

We also examine the normalized goodputs of the TCP flows of different weights under SQ-EWFQ when applying various ρ values, and present the results in Fig. 16(b). We find that when ρ is small, SQ-EWFQ behaves more close to SQ-WFQ, which is unfavorable to the small-weighted TCP flows. But with a larger ρ value, the small-weighted TCP flows are allocated bandwidths more fairly, as the SQ-EWFQ algorithm allows a TCP sender's cwnd to temporarily grow more aggressively for transmitting larger bursts of packets. From Fig. 16, one can see that by varying ρ, SQ-EWFQ makes a tradeoff between the short-term and long-term fairness in packet scheduling.

C. Packet-level Simulation on a Large-Scale Network

Besides the single-switch topology, we also evaluate our proposed WFQ schedulers and compare with other solutions using a large-scale leaf-spine data center network as shown in Fig. 17. We simulate the network using `Netbench` [34]. The

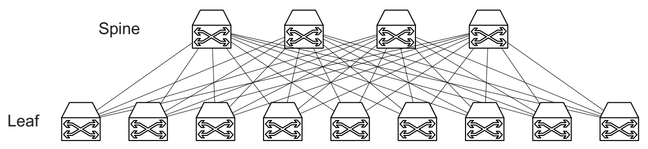


Fig. 17. The leaf-spine data center network topology.

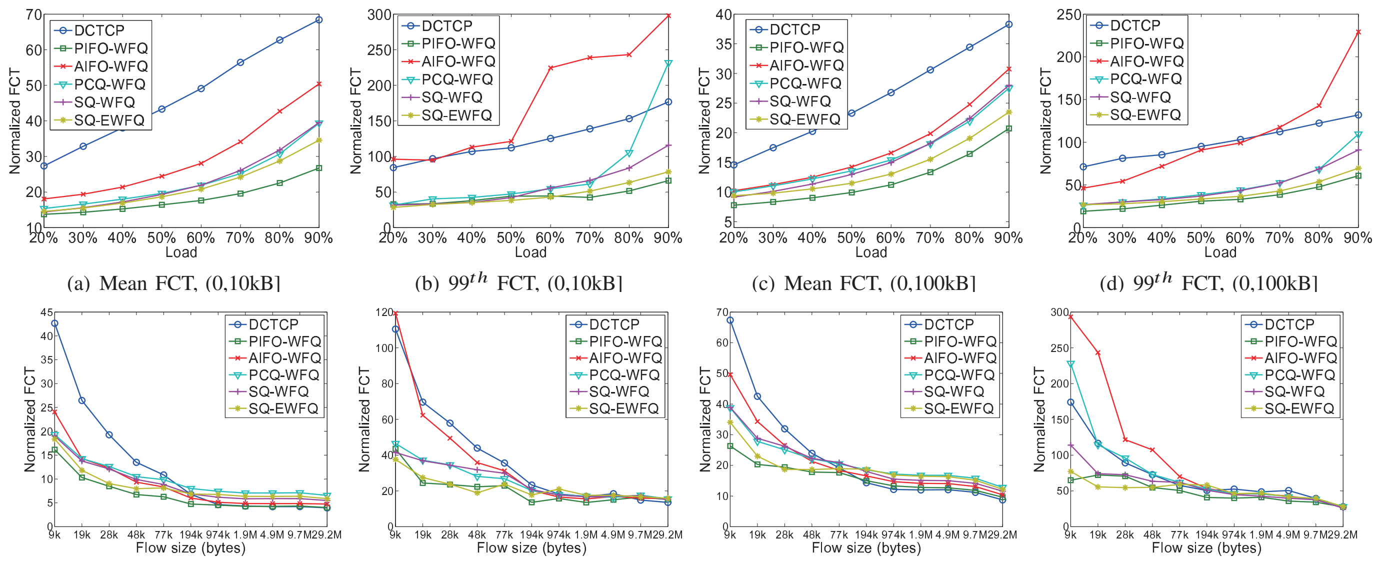
network contains 4 spine switches and 9 leaf switches. Each leaf switch connects to 16 hosts, and a total number of 144 hosts are connected by the network. The host-leaf link has a bandwidth of 10 Gbit/s and the leaf-spine link has a bandwidth of 40 Gbit/s. The per-port buffer size is set as 150 kB.

Our experiments are driven by empirical web search workload in a data center network [42]. In particular, the workload follows a heavy-tailed distribution regarding flow size, and flows arrive in a Poisson process with different inter-arrival intervals resulting in different traffic loads. All flows are over TCP with NewReno [37] as the end-to-end congestion control, and flows are of same weight.

We examine the flow completion time (FCT) of the TCP flows in the data center network, and present the normalized FCTs[3] for the flows of different sizes under various network loads in Fig. 18. In addition to PIFO-WFQ, AIFO-WFQ, PCQ-WFQ and our proposed SQ-WFQ and SQ-EWFQ algorithms, we also evaluate DCTCP [42] with a default ECN marking threshold of 65 packets. For PCQ-WFQ, we only consider the scheduler with the PCQ containing 8 queues. We make the following observations from the experiment results:

Small-sized flows benefit from fewer packet drops. From Fig. 18(a), (b), (e)-(h), we find that small-sized TCP flows have lower mean and 99th percentile FCTs under our proposed SQ-WFQ and SQ-EWFQ schedulers. For example, for the flows with sizes no larger than 10 kB, SQ-WFQ reduces the mean FCTs of the AIFO-WFQ scheduler by 19.5% – 25.5%, and reduces the 99th percentile FCTs by as much as 61.1% – 75.2%, under the network loads varying from 20% to 90%; SQ-EWFQ reduces the mean FCTs of PCQ-WFQ by 4.9% – 12.2%, and reduces the 99th percentile FCTs by 10.0% – 66.2%. The advantages of SQ-WFQ and SQ-EWFQ over AIFO-WFQ and

[3]Normalized FCT means a flow's actual FCT normalized to its ideal FCT when no other flows are active in the network.



(a) Mean FCT, (0,10kB] (b) 99^{th} FCT, (0,10kB] (c) Mean FCT, (0,100kB] (d) 99^{th} FCT, (0,100kB]

(e) Mean FCT breakdown, 50% load (f) 99^{th} FCT breakdown, 50% load (g) Mean FCT breakdown, 90% load (h) 99^{th} FCT breakdown, 90% load

Fig. 18. Normalized FCTs of TCP flows of different sizes when imposing various loads from the web search workload [42] on the leaf-spine data center network.

PCQ-WFQ can be explained with the fact that the small-sized TCP flows, which transfer data with a few bursts, are vulnerable under the excessive packets drops made by AIFO-WFQ and PCQ-WFQ. By reducing the excessive drops, our proposed SQ-WFQ and SQ-EWFQ algorithms enable the small-sized flows to better claim their fair share bandwidths. Moreover, small-sized TCP flows further benefit from the burst tolerance of SQ-EWFQ, and can grow larger cwnds without suffering frequent packet losses.

Heavily-loaded networks benefit from fairer bandwidth allocations. From Fig. 18(a)-(d), (g), and (h), we find that our proposed SQ-WFQ and SQ-EWFQ schedulers benefits small-to-medium sized TCP flows more significantly when the network is heavily loaded. For example, under the 90% network load, for TCP flows no larger than 48 kB, SQ-WFQ reduces the 99^{th} percentile FCTs of AIFO-WFQ and PCQ-WFQ by 40.2% and 13.1% respectively, in contrast to the reductions of 11.2% and 0.5% under the 50% network load; and SQ-EWFQ reduces the 99^{th} percentile FCTs of AIFO-WFQ and PCQ-WFQ by 48.8% and 24.7%, in contrast to the reductions of 47.1% and 19.0% under the 50% network load. The reason behind the significant FCT reductions is that in a heavily-loaded network, more TCP flows compete for bandwidth, and consequently, each flow will have a smaller normalized weight than they would have in a lightly-loaded network. As we have seen in Sec. V-B2, SQ-WFQ and SQ-EWFQ are fairer in allocating bandwidths to the small-weighted TCP flows than the other approximate schedulers, and reduce their FCTs more significantly.

Worst-case performances are improved under SQ-WFQ and SQ-EWFQ. By comparing Fig. 18(e) and (f), (g) and (h), we find that our proposed SQ-WFQ and SQ-EWFQ schedulers improve the worst-case performances, i.e., the 99^{th} percentile FCTs of the TCP flows more significantly than other schedulers. For example, under the 90% network load, for network flows within 10 kB, SQ-WFQ reduces the 99^{th} percentile FCTs of DCTCP, AIFO-WFQ, and PCQ-WFQ by 34.5%, 61.3%, and 50.1% respectively, and SQ-EWFQ further reduces the 99^{th} percentile FCTs of the three schedulers by as much as 55.6%, 73.7%, and 66.2%. The improvements can be explained with the improved fairness provided by SQ-WFQ and SQ-EWFQ. As in DCTCP, AIFO-WFQ, and PCQ-WFQ, if a small-sized TCP flow unfortunately encounters retransmission timeouts in its first a few RTTs, it will have difficulty to grow its cwnd, and as a result, this flow will have a low goodput and long FCT, as we have seen in Fig. 14. On the other hand, as small-sized flows are scheduled more fairly by SQ-WFQ and SQ-EWFQ, their worse-case performances are substantially improved.

SQ-EWFQ is close to and even outperforms PIFO-WFQ. From Fig. 18(a)-(h), one can see that our proposed SQ-EWFQ algorithm has achieved the mean and 99^{th} percentile FCTs very close to the ideal PIFO-WFQ scheduler, and for small-to-medium sized TCP flows, SQ-EWFQ even outperforms PIFO-WFQ regarding the 99^{th} percentile FCT in heavily-loaded networks.

The reason for the good performance of SQ-EWFQ is two-fold: First, as extended from SQ-WFQ, SQ-EWFQ reduces the excessive packet drops made by AIFO-WFQ and PCQ-WFQ, thus is fairer in bandwidth allocation than them. Second, by temporarily increasing a bursty TCP flow's weight in short intervals while enforcing a long-term fairness, SQ-EWFQ allows larger packet bursts than PIFO-WFQ does, and better handles the bursty TCP traffic.

VI. Conclusion

In this paper, we focused on weighted fair queueing (WFQ), and studied both the ideal scheduler realized on the PIFO queue abstraction and the approximate schedulers based on AIFO and PCQ. We found that existing WFQ packet schedulers can not allocate bandwidths to TCP flows fairly, because of the unexpected impacts of the scheduled packet drops on TCP congestion control, and the AIFO and PCQ-based approximate schedulers further degrade the scheduling fairness due to their excessive packet drops.

To address these issues, in this paper we propose approximate WFQ packet scheduling algorithms. Our initial design, namely SQ-WFQ, employs only one single FIFO queue, and is capable to schedule packets at line rate. In addition, SQ-WFQ is work-conserving and achieves the max-min weighted fairness when scheduling UDP flows. By avoiding excessive packet drops, SQ-WFQ improves the fairness in scheduling TCP flows, comparing with the AIFO and PCQ-based approximate WFQ schedulers. Extended from SQ-WFQ, we propose the SQ-EWFQ algorithm. SQ-EWFQ inherits all the merits of SQ-WFQ, and by tolerating packet bursts while enforcing a long-term fairness, SQ-EWFQ outperforms all the existing WFQ packet schedulers regarding fairness in scheduling TCP flows. In particular, SQ-EWFQ is capable to allocate bandwidths more fairly to small-weighted TCP flows, TCP flows of longer RTTs, TCP flows applying less aggressive congestion controls, and small-to-medium sized TCP flows than the existing ideal and approximate WFQ schedulers. We have implemented prototypes of the SQ-WFQ and SQ-EWFQ schedulers on the Tofino-based commodity hardware programmable switches, and shared our P4 implementations with the community.

References

[1] A. K. Parekh and R. G. Gallager, "A generalized processor sharing approach to flow control in integrated services networks: the single-node case," *IEEE/ACM Trans. Networking*, vol. 1, no. 3, 1993.

[2] H. J. Chao, Y.-R. Jenq, X. Guo, and C. H. Lam, "Design of packet-fair queuing schedulers using a RAM-based searching engine," *IEEE J. Sel. Areas Commun.*, vol. 17, no. 6, 1999.

[3] A. Ioannou and M. G. H. Katevenis, "Pipelined heap (priority queue) management for advanced scheduling in high-speed networks," *IEEE/ACM Trans. Networking*, vol. 15, no. 2, 2007.

[4] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown, "Programmable packet scheduling at line rate," in *Proc. of SIGCOMM'16*, Florianopolis, Brazil, Aug. 2016.

[5] V. Shrivastav, "Fast, scalable, and programmable packet scheduler in hardware," in *Proc. of SIGCOMM'19*, Beijing, China, Aug. 2019.

[6] Z. Yu, C. Hu, J. Wu, X. Sun, V. Braverman, M. Chowdhury, Z. Liu, and X. Jin, "Programmable packet scheduling with a single queue," in *Proc. SIGCOMM'21*, Virtual Event, Aug. 2021.

[7] N. K. Sharma, C. Zhao, M. Liu, P. G. Kannan, C. Kim, A. Krishnamurthy, and A. Sivaraman, "Programmable calendar queues for high-speed packet scheduling," in *Proc. of NSDI'20*, Santa Clara, CA, USA, Feb. 2020.

[8] A. G. Alcoz, A. Dietmüller, and L. Vanbever, "SP-PIFO: Approximating push-in first-out behaviors using strict-priority queues," in *Proc. of NSDI'20*, Santa Clara, CA, USA, Feb. 2020.

[9] N. K. Sharma, M. Liu, K. Atreya, A. Krishnamurthy, and A. Sivaraman, "Approximating fair queueing on reconfigurable switches," in *Proc. of NSDI'18*, Renton, WA, USA, Apr. 2018.

[10] P. Gao, A. Dalleggio, Y. Xu, and H. J. Chao, "Gearbox: A hierarchical packet scheduler for approximate weighted fair queuing," in *Proc. of NSDI'22*, Renton, WA, USA, Apr. 2022.

[11] "Intel tofino series," accessed on Nov. 30, 2022. [Online]. Available: https://intel.com/content/www/us/en/products/details/network-io/programmable-ethernet-switch/tofino-series.html

[12] "$P4_{16}$ portable switch architecture (psa)," The P4.org Architecture Working Group, Tech. Rep., 2021.

[13] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queueing algorithm," in *Proc. of SIGCOMM'89*, Austin, Texas, USA, Sep. 1989.

[14] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "pFabric: Minimal near-optimal datacenter transport," in *Proc. of SIGCOMM'13*, Hong Kong, China, Aug. 2013.

[15] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang, "Information-agnostic flow scheduling for commodity data centers," in *Proc. of NSDI'15*, Oakland, CA, USA, May 2015.

[16] V. Dukic, S. A. Jyothi, B. Karlas, M. Owaida, C. Zhang, and A. Singla, "Is advance knowledge of flow sizes a plausible assumption?" in *Proc. of NSDI'19*, Boston, MA, USA, Feb. 2015.

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

[18] L. E. Schrage and L. W. Miller, "The queue M/G/1 with the shortest remaining processing time discipline," *Operations Research*, vol. 14, no. 4, 1997.

[19] Z. Wang, J. Ye, D. Lin, Y. Chen, and J. C. S. Lui, "Approximate and deployable shortest remaining processing time scheduler," *IEEE/ACM Trans. Networking*, vol. 30, no. 3, 2022.

[20] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, 1973.

[21] P. McKenney, "Stochastic fairness queueing," in *Proc. of IEEE INFOCOM'90*, San Francisco, CA, USA, Jun. 1990.

[22] M. Shreedhar and G. Varghese, "Efficient fair queuing using deficit round-robin," *IEEE/ACM Trans. Networking*, vol. 4, no. 3, 1996.

[23] J. Liu, J. Huang, Z. Li, Y. Li, J. Wang, and T. He, "Achieving per-flow fairness and high utilization with limited priority queues in data center," *IEEE/ACM Trans. Networking*, vol. 30, no. 5, 2022.

[24] R. Mahajan, S. Floyd, and D. Wetherall, "Controlling highbandwidth flows at the congested router," in *Proc. of ICNP'01*, Riverside, CA, USA, Nov. 2001.

[25] R. Pan, L. Breslau, B. Prabhakar, and S. Shenker, "Approximate fairness through differential dropping," *SIGCOMM CCR*, vol. 33, no. 2, 2003.

[26] I. Stoica, S. Shenker, and H. Zhang, "Core-stateless fair queueing: a scalable architecture to approximate fair bandwidth allocations in high-speed networks," *IEEE/ACM Trans. Networking*, vol. 11, no. 1, 2003.

[27] Z. Yu, J. Wu, V. Braverman, I. Stoica, and X. Jin, "Twenty years after: Hierarchical core-stateless fair queueing," in *Proc. of NSDI'21*, Boston, MA, USA, Apr. 2021.

[28] R. MacDavid, X. Chen, and J. Rexford, "Scalable real-time bandwidth fairness in switches," *IEEE/ACM Trans. Networking*, 2003, early access.

[29] L. Yu, J. Sonchack, and V. Liu, "Cebinae: Scalable in-network fairness augmentation," in *Proc. of SIGCOMM'22*, Amsterdam, Netherlands, Aug. 2022.

[30] B. Stephens, A. Singhvi, A. Akella, and M. Swift, "Titan: Fair packet scheduling for commodity multiqueue NICs," in *Proc. of ATC'17*, Santa Clara, CA, USA, Jul. 2017.

[31] B. Stephens, A. Akella, and M. Swift, "Loom: Flexible and efficient nic packet scheduling," in *Proc. of NSDI'19*, Boston, MA, USA, Feb. 2019.

[32] A. Saeed, Y. Zhao, N. Dukkipati, E. Zegura, M. Ammar, K. Harras, and A. Vahdat, "Eiffel: Efficient and flexible software packet scheduling," in *Proc. of NSDI'19*, Boston, MA, USA, Feb. 2019.

[33] T. Flach, N. Dukkipati, A. Terzis, B. Raghavan, N. Cardwell, Y. Cheng, A. Jain, S. Hao, E. Katz-Bassett, and R. Govindan, "Reducing web latency: the virtue of gentle aggression," in *Proc. of SIGCOMM'13*, Hong Kong, China, Aug. 2013.

[34] "Netbench," accessed on Feb. 20, 2023. [Online]. Available: https://github.com/ndal-eth/

[35] S. Kassing, A. Valadarsky, G. Shahaf, M. Schapira, and A. Singla, "Beyond fat-trees without antennae, mirrors, and disco-balls," in *Proc. SIGCOMM'17*, Los Angeles, CA, USA, Aug. 2017.

[36] "iperf3," accessed on Feb. 20, 2023. [Online]. Available: https://iperf.fr/

[37] T. Henderson, S. Floyd, A. Gurtov, and Y. Nishida, "The NewReno modification to TCP's fast recovery algorithm," 2004, rfc 6582.

[38] "tc (Linux)," accessed on Feb. 20, 2022. [Online]. Available: https://en.wikipedia.org/wiki/Tc_(Linux)

[39] D. Leith and R. Shorten, "H-TCP: TCP congestion control for high bandwidth-delay product paths," 2005, internet draft draft-leith-tcp-htcp-00.txt.

[40] I. Rhee, L. Xu, S. Ha, A. Zimmermann, L. Eggert, and R. Scheffenegger, "CUBIC for fast long-distance networks," 2018, rfc 8312.

[41] L. Xu, K. Harfoush, and I. Rhee, "Binary increase congestion control (BIC) for fast long-distance networks," in *Proc. INFOCOM'04*, Hong Kong, China, Mar. 2004.

[42] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center TCP (DCTCP)," in *Proc. of SIGCOMM'10*, New Delhi, India, Aug. 2010.

Wei Chen received the bachelor's degree in computer science from University of Science and Technology of China (USTC), Hefei, China, in 2020. He is currently pursuing the Ph.D. degree with the School of Computer Science and Technology, USTC. His research interests include programmable networks and network traffic scheduling.

Ye Tian received the bachelor's degree in electronic engineering and the master's degree in computer science from University of Science and Technology of China (USTC), Hefei, China, in 2001 and 2004, respectively, and the Ph.D. degree from the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, China, in 2007. He joined USTC in 2008 and is currently an Associate Professor with the School of Computer Science and Technology, USTC. His research interests include programmable networks, network traffic scheduling, and network measurement. He has published over 80 papers and co-authored a research monograph published by Springer. He is the winner of the Wilkes Best Paper Award of Oxford The Computer Journal in 2016. He is a member of the IEEE.

Xin Yu received the bachelor's degree in computer science from University of Science and Technology of China (USTC), Hefei, China, in 2021. He is currently working toward the master's degree with the School of Computer Science and Technology, USTC. His research interest is focused on network traffic scheduling.

Bowen Zheng received the bachelor's degree in software engineering from University of Electronic Science and Technology of China, Chengdu, China, in 2022. He is currently pursuing the master's degree with the School of Computer Science and Technology, USTC. His research interest is focused on programmable networks.

Xinming Zhang received the BE and ME degrees in electrical engineering from China University of Mining and Technology, Xuzhou, China, in 1985 and 1988, respectively, and the PhD degree in computer science and technology from the University of Science and Technology of China (USTC), Hefei, China, in 2001. Since 2002, he has been with the faculty of USTC, where he is currently a Professor with the School of Computer Science and Technology. From September 2005 to August 2006, he was a visiting Professor with the Department of Electrical Engineering and Computer Science, Korea Advanced Institute of Science and Technology, Daejeon, Korea. His research interest includes wireless networks, deep learning, and intelligent transportation. He has published more than 100 papers. He won the second prize of Science and Technology Award of Anhui Province of China in Natural Sciences in 2017. He won the awards of Top reviewers (1%) in Computer Science & Cross Field by Publons in 2019. He is a senior member of the IEEE.