Sifter: An Inversion-Free and Large-Capacity Programmable Packet Scheduler

Peixuan Gao
New York University

Anthony Dalleggio
New York University

Jiajin Liu
New York University

Chen Peng
New York University

Yang Xu [*]
Fudan University

H. Jonathan Chao
New York University

Abstract

Packet schedulers play a crucial role in determining the order in which packets are served. They achieve this by assigning a rank to each packet and sorting them based on these ranks. However, when dealing with a large number of flows at high packet rates, sorting functions can become extremely complex and time-consuming. To address this issue, fast-approximating packet schedulers have been proposed, but they come with the risk of producing scheduling errors, or packet inversions, which can lead to undesirable consequences. We present Sifter, a programmable packet scheduler that offers high accuracy and large capacity while ensuring inversion-free operation. Sifter employs a unique sorting technique called "Sift Sorting" to coarsely sort packets with larger ranks into buckets, while accurately and finely sorting those with smaller ranks using a small Push-In-First-Out (PIFO) queue in parallel. The sorting process takes advantage of the "Speed-up Factor", which is a function of the memory bandwidth to output link bandwidth ratio, to achieve Sift Sorting and ensure accurate scheduling with low resource consumption. Sifter combines the benefits of PIFO's accuracy and FIFO-based schedulers' large capacity, resulting in guaranteed delivery of packets in an accurate scheduling order. Our simulation results demonstrate Sifter's efficiency in achieving inversion-free scheduling, while the FPGA-based hardware prototype validates that Sifter supports a throughput of 100Gbps without packet inversion errors.

1 Introduction

The Programmable Data Plane has seen increased interest in both academia and industry to enhance the flexibility and the programmability of ultra-high-speed networks without compromising the throughput and latency performance [9] [27] [56] [53] [55] [23]. The academic and industry communities have built a sophisticated ecosystem for the programmable data plane including the P4 programming language (P4) [8], cross-platform high-level languages and their compilers [42] [38], programmable switch devices [11], [17], and a variety of re-configurable network chip architectures [18] [10].

Recent research has focused on enhancing the programmability of packet scheduling in the data plane. Various packet scheduling disciplines sort packets based on a "rank" value assigned by the scheduler to represent their transmission order. [48] [49] [35].

However, implementing a priority queue that sorts packets in an ultra-high-speed data plane is challenging. The scheduler needs to serve each packet within a very limited time budget. For a 64-byte packet on a 100 Gbps Ethernet link, the packet processing time is less than 7 ns. Packet schedulers such as Sequencer [13] [14] [15] and PIFO[48] [49] manage to sort packets with ultra-low time complexity by sacrificing buffer capacity. To accommodate larger buffer sizes and reduce the implementation complexity, the community proposed a number of approximating packet schedulers such as PCQ [45], SP-PIFO [2], Gearbox [24] and AIFO [54] that trade-off scheduling accuracy for simplicity and scalability.

Scheduling errors, also known as packet inversions [2], are introduced by approximating packet schedulers and may impact network performance. These packet inversions cause throughput fluctuation, affect fairness, introduce delay, and slow down the flow completion time (FCT). Failure to rigorously adhere to the scheduling order of bandwidth allocation algorithms can negatively affect performance and fairness, which are essential for network isolation [31] [6] [46] [30] [41] [50]. Moreover, emerging applications, such as self-driving vehicles and remote surgery, exhibit a high sensitivity to packet inversions. As a result, there is a pressing requirement for precise packet scheduling to maintains strict packet order, not only for delay-sensitive applications but also for algorithms that react adversely to packet inversions.

Accuracy and scalability of the scheduler are often conflicting requirements. We want a scheduler that can accurately serve packets by the strict order of their ranks (e.g., PIFO) and also support a large buffer size with low hardware resource consumption.

We present Sifter, an accurate and large-capacity programmable packet scheduler that operates free of packet inversions and supports a large buffer size with low implementation overhead. Sifter sorts packets by "Sift Sorting", a sorting method consisting of two parallel processes. Sifter sorts packets with larger ranks coarsely into a FIFO-based "Rotating Calendar Queue" (RCQ) and packets with smaller

[*]Corresponding author

ranks accurately using a mini PIFO. Sifter is intended for next-generation programmable switches and smart NICs. By taking advantage of the "Speed-up Factor" provided by the hardware architecture, Sifter combines the advantages of accurate scheduling of a PIFO with the large capacity provided by FIFO-based approximate schedulers. Based on our simulation results and our hardware prototype, Sifter eliminates packet inversions[1] while supporting a large buffer capacity and low implementation complexity. The contributions of this paper are summarized as follows:

- **Analysis of the impact of packet inversions on network performance.** Starting from the different scheduling goals of packet scheduling algorithms, this paper discusses the impact of packet inversion errors on network performance. Various scheduling algorithms have different sensitivities to packet inversions, and delay-guarantee scheduling algorithms are the most sensitive to the accuracy of the packet scheduling order.

- **Design of an accurate and large-capacity packet scheduler with inversion-free operation.** This paper presents Sifter, an accurate and large-capacity packet scheduler that operates free of packet inversions without sacrificing scalability or increasing implementation complexity using "Sift Sorting". By exploiting the "Speed-up Factor", Sifter combines the accuracy of PIFO and the large capacity of RCQ, a FIFO-based scheduler.

- **Definition of the conditions for achieving inversion-free operation.** This paper presents a quantitative analysis to determine the conditions under which Sifter guarantees operation with no packet inversions.

- **Comprehensive simulation of Sifter on NS3 and experiments on an FPGA-based Sifter prototype.** We implemented Sifter in the NS3 simulator and in VHDL on an AMD/Xilinx Alveo U250 FPGA card [4] with a mid-speed grade XCVU13P FPGA. In the NS3 [37] simulator, we conduct comprehensive evaluations of Sifter using multiple metrics, demonstrating that it provides packet scheduling performance that is close to the ideal PIFO. Our hardware testbed results show the FPGA-based Sifter prototype operates at 322 MHz and achieves a line rate of 100 Gbps for packets larger than 370 bytes[2] without any inversions.

The rest of the paper is organized as follows. *Section* 2 provides the background and motivation of our work. *Section* 3 presents the detailed architecture, scheduling processes of Sifter and the conditions for inversion-free scheduling. *Section* 4 provides a detailed Sifter prototype implementation and *Section* 5 presents the packet-based NS3 and the hardware testbed evaluations. We discuss related works in *Section* 6 and conclude the paper in *Section* 7.

[1]See *Section* 3.5 for the conditions for inversion-free scheduling.

[2]See *Appendix* H for more details.

2 Background and Motivation

2.1 Programmable Packet Scheduling

The concept of programmable packet scheduling is to enable network administrators to schedule packets using different packet scheduling algorithms. According to previous literature [48] [49] [35], different packet scheduling algorithms determine the packet scheduling order to achieve certain goals, such as max-min fairness in bandwidth allocation [20] [39] [40] [25] [26] [57], minimizing FCT [43] [3] as well as delay guarantee [34] [35]. When a generalized scheduler serves packets in ascending order of their rank values, it effectively implements the scheduling order dictated by a specific scheduling algorithm. Consequently, the abstraction of programmable packet scheduling involves two key steps: (1) establishing the sequence for scheduling each packet (*Rank Calculation*) and (2) enforcing this packet scheduling sequence (*Rank Sorting*).

The calculation of packet *rank* can be programmed at the end-hosts before the packet is sent out, or even in the programmable ingress and egress switch pipeline [9][8]. When the packet arrives at the scheduler, the switch needs to enforce the scheduling order by sorting the packets by their *rank* values. Based on this theory, multiple works have implemented generic programmable packet schedulers [45] [2] [24] [54].

2.2 Packet Inversions

To deliver a correct packet scheduling order, an ideal packet scheduler needs to serve packets strictly according to the increasing order of the *rank* of each packet. However, due to limited packet processing time and implementation complexity, many generic packet scheduler designs cannot serve packets in such an ideal order and instead, schedule packets in an approximate increasing order, which may introduce packet inversions. We formally define packet inversions[3].

Packet Inversions: When a packet with rank r departs from the scheduler, there exists packet(s) with a smaller rank r' (where smaller rank has higher priority, $r' < r$) in the scheduler.

2.2.1 Impact of Packet Inversions

Packet inversions have various impacts on different packet scheduling algorithms according to their logic and goals.

Fairness The impact of packet inversions on fairness-based scheduling algorithms varies from fluctuation in short-term bandwidth allocation to fairness impairment. Most of the fairness-based algorithms [20] [39] [40] [25] [47] [26] [57] are derived from GPS, which serves flows based on round-robin logic by assigning a departure time to each packet. Packet inversions can have several negative effects,

[3]We further extend *Packet Inversions* by *Inversion Magnitude* to quantify the severity of packet inversions in AppendixA.

including short-term fairness issues and fluctuations in both throughput and delay.

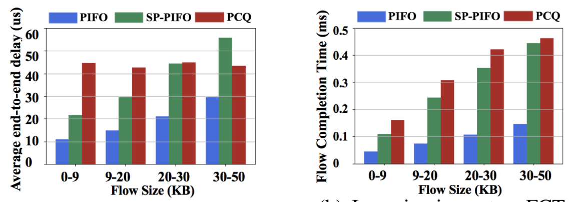
However, for the fairness-based scheduling algorithms that update the virtual clock according to the departure times of packets [25] [26], the consequence of packet inversions could be much more severe. When there is a packet inversion and a packet with a very large timestamp (rank) t' departs early, the scheduler updates its current virtual clock from t to a much larger value t' in advance. The skew on the virtual clock can prevent any newly arrived packet from getting a departure time between t and t'. As a consequence, existing packets with departure times smaller than t' would take up the bandwidth and lead to starvation of the newly arrived flows. The accumulation of such packet inversions can lead to performance degradation and impairment of network isolation [31] [6] [46] [30] [41] [50].

Minimizing FCT Other packet scheduling algorithms, such as those designed to minimize average FCT [43] [3] [28], can be significantly impacted by packet order inversions. The core logic of minimizing FCT is to always serve the flow with the minimum remaining flow size. However, if packet inversions occur, the scheduler may serve other flows with larger remaining sizes and may possibly serve multiple flows at the same time. For example, when packet inversions occur, the packet scheduler may fail to differentiate between multiple flows that have similar remaining sizes. As a result, the scheduler may start serving multiple flows in a round-robin fashion instead of serving the flow with the minimum size, which can dramatically increase the average FCT.

Delay Guarantee (Tail Packet Delay) Scheduling disciplines that are most affected by packet inversions are those that provide a delay guarantee [34] [52]. Typically, these scheduling algorithms cater to delay-sensitive applications that impose strict end-to-end delay requirements, such as Vehicle to Everything (V2X) and remote surgery. These algorithms usually assign a delay budget to each packet ("slack time"), which represents the end-to-end delay requirement. Upon the arrival of each packet, the packet scheduler sorts packets according to their urgency and always schedules the packet with the least "slack time". In this case, the accuracy of the packet scheduler is critical. A minor inversion in scheduling could result in a series of packets missing their deadlines and having a major negative impact on the performance of the application.

Evaluating Impacts of Packet Inversions We evaluated the impacts of packet inversions by running Start-Time Fair Queueing (SFQ) [26] on approximate schedulers and a PIFO. The evaluation setup is described in detail in *Appendix* F.

Figure 1(a) shows the impact of packet inversion on packet delay. When packet inversions occur, packets with larger ranks are scheduled prior to other packets with smaller ranks, which leads to extra queuing delay for the packets that should



(a) Inversion impact on delay (b) Inversion impact on FCT

Figure 1: Impacts of Packet Inversions

have been scheduled earlier. Furthermore, Figure 1(b) shows the impact of packet inversion on FCT. Approximate packet schedulers introduce extra delay and unfairness in throughput due to packet inversions, which impairs the FCT of small flows.

We have also noted that the introduction of randomness through packet inversion amplifies both throughput and delay uncertainties. Certain applications like autonomous vehicles and remote surgery heavily depend on consistent connection performance.

2.3 Achieving accurate packet scheduling

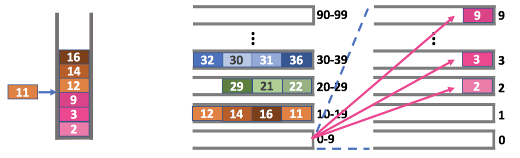
2.3.1 Causes of Packet Inversions

As discussed in *Section* 2.2, existing implementations of approximating packet schedulers [44] [45] [2] [24] [54] are subject to packet inversions. These approximating schedulers are implemented with one or more FIFO queues and packets from each FIFO queue are served on a first-come-first-served basis, which can lead to packet inversions between the packets from the same FIFO queue. Although such a FIFO-based structure has the advantages of low time-complexity, high scalability, and implementation simplicity, the potential packet inversions may result in various consequences as discussed in *Section* 2.2.1. The key to eliminating those packet inversions is to resolve any packet misordering within each FIFO queue.

2.3.2 Proactive and Reactive Packet Sorting

An intuitive solution to resolve packet misordering is to sort the packets in each FIFO queue before they are dequeued. The existing solutions of packet sorting can be categorized into two classes, proactive and reactive. The proactive approach sorts each packet upon arrival and maintains a sorted priority queue [19]. As Figure 2(a) shows, each packet is inserted into the correct place upon arrival. Such sorting algorithms are subject to a fairly large time complexity of $O(logN)$ for each packet enqueue [4]. This sorting is difficult to complete within the limited packet processing time in ultra-high-speed data planes. A number of works [13] [14] [15] [48] [49] overcome this challenge and provide proactive packet sorters operating at line rate, however, they usually cannot support a large buffer capacity. The reactive approach

[4]N is the total number of packets inside the packet sorter



(a) Proactive Sorter (b) Reactive Sorter

Figure 2: Proactive and Reactive Sorters

sorts packets as part of the dequeue process [45]. As shown in Figure 2(b), reactive solutions enqueue packets into a set of strict priority FIFO queues and sort them with finer granularity when the scheduler is about to serve these packets. Although these reactive sorters provide a fast enqueue process and large capacity, the sorting process is time-consuming and might lead to blocking in the dequeue process.

2.3.3 Speed-up Factor

We can achieve a perfect packet sorting solution that combines the advantages of non-blocking dequeue from proactive sorters and large capacity from reactive sorters. The key to our approach to accomplish this is sorting packets in parallel with the dequeue process.

Memory bandwidth is usually higher than the output links bandwidth on hardware switches and network interface cards (NIC). In addition, packet schedulers usually schedule descriptors[5] that represent packets. Packet descriptors are much smaller than the average size of a packet. Combining the above facts, hardware switches can access multiple packet descriptors during the time of dequeuing a single packet. We adopt the concept of "Speed-up Factor" K, a number that measures the relative speed between the descriptor access speed and packet dequeue speed.

$$K = \lfloor \frac{R_M}{R_o} \cdot \frac{\min L_P}{L_d} \rfloor \tag{1}$$

where K is the minimum Speed-up factor[6], R_M is the memory bandwidth, R_o is the output link line rate, $\min L_P$ is the minimum packet size and L_d is the byte-length of the packet descriptor. For illustration purposes, let us consider a system with R_M = 64 Gbps (64-bit memory running at 1 GHz), L_d = 64 bits, $\min L_P$ = 64 bytes, and R_o = 100 Gb/s, then

$$K = \lfloor \frac{64 \cdot 10^9}{100 \cdot 10^9} \cdot \frac{64 \cdot 8}{64} \rfloor = \lfloor 5.12 \rfloor = 5 \tag{2}$$

A speed-up factor K means the scheduler can access at least K packet descriptors during the time of dequeuing a packet. By taking advantage of the speed-up factor, the reactive FIFO-based sorter can sort packets with a proactive priority queue (PIFO) in parallel without causing blocking.

[5]A packet descriptor is also known as metadata.

[6]When packet sizes are larger than the minimum packet size $\min L_P$, the speed-up factor would be larger accordingly.

3 Sifter: An Accurate and Large-Capacity Programmable Packet Scheduler

3.1 Concept and Architecture

The previous sections establish a need for a programmable packet scheduler with accuracy and scalability, combining the advantages of both a perfect and an approximating PIFO.

We introduce Sifter, a scalable PIFO to support large-capacity programmable packet scheduling without packet inversions. As shown in Figure 3, Sifter consists of two major components: (1) A Mini-PIFO and (2) An RCQ. The Mini-PIFO stores the packets with relatively smaller ranks. It performs a strict sorting of packets and always outputs the packet with the smallest rank. The RCQ holds the rest of the packets with relatively larger ranks. It stores and sorts packets with coarse granularity[7] similarly to [12] [51] [44] [45] [2] [24] by a set of strict-priority FIFO queues and is therefore widely scalable in rank range and capacity.

Sifter uses an algorithm, "Sift Sorting", which consists of two parallel processes to schedule packets in strict order: The first process is Enqueue, in which packets are stored in different components according to their ranks. The packets with larger ranks enter the RCQs and are sorted with coarse granularity while the packets with smaller ranks enter the Mini-PIFO and are strictly sorted. The second process called "Sifting" operates in parallel. When packets dequeue from the Mini-PIFO and there is available space, the RCQ keeps migrating (sifting) the packets from FIFOs with smaller ranks to the Mini-PIFO. The Mini-PIFO therefore eliminates the potential misordering of ranks in a FIFO in the RCQ using only minimal-sized sifting registers. By combining the RCQ and Mini-PIFO, Sifter schedules packets in the correct order while supporting a large capacity.

The name Sifter is derived from its sorting method. If each packet is a particle in the sifter, the rank of each packet would represent the size of the particle. Sifter "filters" the larger particles from the upper layers and passes through the smaller particles, as shown in Figure 3. Since the Mini-PIFO always holds the packets with the smallest ranks in the scheduler, Sifter dequeues packets from the head of the Mini-PIFO. For a more in-depth look at Sifter's operation, we introduce its three major processes: Enqueue, Dequeue, and Sifting in *Section* 3.2.

Sifter only schedules packet descriptors[8]. In this paper, "packets" refer to the packet descriptor in the scheduler. To

[7]Granularity: the rank range that a FIFO in the RCQ accommodates.

[8]Packets are stored in a shared-memory buffer [1] [16] and the scheduler only deals with their descriptors. An example of packet descriptor is provided in Appendix E

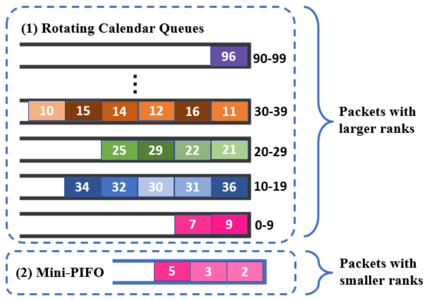


Figure 3: Sifter Architecture

Table 1: TERMS AND NOTATIONS

Notation	Description
$P_{(i,k)}$	k^{th} packet in flow i
$R_{(i,k)}$	Rank of packet $P_{(i,k)}$
K	Speed-up factor of Sifter
R_M	Memory bandwidth
R_o	Line rate of the output link
$\min L_P$	Minimal packet size
L_d	Byte length of a packet descriptor
S_F	Max size of a FIFO in the RCQ
F	Total number of FIFOs in the RCQ
g	Granularity of a FIFO in the RCQ
O_P	Current occupancy of the Mini-PIFO
S_P	Max size of the Mini-PIFO
Th_S	Sifting threshold of the Mini-PIFO
s	Rank value of the Sentinel

better illustrate the detailed schemes in Sifter, we summarize the related concepts and notations in Table 1.

3.2 Enqueue, Dequeue and Sifting Processes

Sifter has three major processes: Enqueue, Dequeue, and Sifting. Figure 4 shows the workflow of Sifter. When packets arrive at Sifter, those with larger ranks enter the RCQ and those with smaller ranks enqueue directly into the Mini-PIFO. When the Mini-PIFO's occupancy is below a given threshold described below, Sifter moves packets from the RCQ to the Mini-PIFO through "Sifting". If the Mini-PIFO becomes full, the packet with the largest rank is evicted from the tail of the Mini-PIFO back to the RCQ as it is replaced with a smaller-rank packet. In the dequeue process, Sifter dequeues the packet with the smallest rank from the head of the Mini-PIFO.

Enqueue The enqueue process decides whether the packet should be placed in the RCQ or the Mini-PIFO. As described in *Section* 3.1, the Mini-PIFO stores the packets with the smallest ranks in the scheduler. To preserve this property, if a newly arriving packet has a rank smaller than the smallest rank in the RCQ, it is stored in the Mini-PIFO. Here we introduce the concept of Sentinel, which represents the maximum rank value of a packet that can enter the Mini-PIFO. In most cases[9], Sentinel is equal to the smallest rank among the packets in the RCQ.

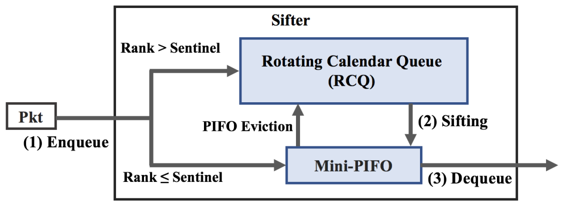


Figure 4: Sifter Workflow

The enqueue process consists of two steps: (1) Determine the target storage element (RCQ or Mini-PIFO) and (2) Enqueue into that storage element. The target storage element of a packet depends on the comparison between its rank and the Sentinel value. If a packet has a rank smaller than or equal to the Sentinel value, it is stored in the Mini-PIFO, otherwise, it is stored in the RCQ. For the second (enqueue) step, packets are enqueued in the Mini-PIFO and the RCQ differently. Enqueuing in the RCQ is straightforward: the packet is stored in the FIFO that accommodates its rank based on the FIFOs' granularity g. To enqueue in the Mini-PIFO, the packet's position in the PIFO is determined by its rank. If the Mini-PIFO is full when a new packet is enqueued, the packet with the largest rank is evicted back to the RCQ. Sifter updates the Sentinel to the evicted packet's rank and stores that packet in the RCQ as described above.

Dequeue Based on the property that the Mini-PIFO always holds the packets with the smallest rank in the scheduler, Sifter always dequeues the packet at the head of the Mini-PIFO. As long as Sifter maintains the above property and keeps sifting the packets with smaller ranks from the RCQ into the Mini-PIFO, the dequeue process will always output the packet with the smallest rank in the scheduler.

Sifting The sifting process is triggered by a "Sifting Threshold", denoted as Th_S. When the occupancy of the Mini-PIFO is lower than Th_S and the RCQ is not empty, Sifter finds the earliest[10] non-empty FIFO in the RCQ and transfers packets from it to the Mini-PIFO. The function and configuration of the Sifting Threshold Th_S are discussed in further detail in *Section* 3.4

The process of sifting the packets from the RCQ into the Mini-PIFO results in a strict ordering of the smallest ranks in the Mini-PIFO. Since the RCQ sorts packets with granularity g in each FIFO queue, the earliest non-empty FIFO queue contains the packets with the smallest rank in RCQ. Sifter needs to traverse all the packets in this FIFO to ensure that the packets it migrates to the Mini-PIFO have ranks smaller

[9]When outside of the Sifting process.

[10]Earliest FIFO: the FIFO that covers the lowest-valued rank range.

than all other packets remaining in the RCQ. This is accomplished by transferring all the packets from this FIFO to the Mini-PIFO such that the packets with the smaller ranks will remain in the Mini-PIFO while the others are evicted back to the RCQ if the Mini-PIFO overflows.

To start the sifting process, Sifter first updates the Sentinel value s to the maximum rank that the FIFO queue accommodates to guarantee that the Sentinel does not block that FIFO's packets from entering the Mini-PIFO. If the Mini-PIFO becomes full, as new packets enter the Mini-PIFO and the packets with the largest ranks are evicted back to the RCQ, Sifter updates the Sentinel s to the rank of the last evicted packet, as in the enqueue process. We further explain the Sentinel update mechanism in *Section* 3.3. When all packets from the earliest non-empty FIFO are transferred, the sifting process is completed and, as a result, the packets with the smallest ranks are held in the Mini-PIFO. If, following a sifting round, the Mini-PIFO occupancy is still lower than the sifting threshold Th_S, Sifter again finds the next earliest non-empty FIFO and repeats the sifting process until the occupancy of the Mini-PIFO is higher than Th_S.

We describe the Enqueue, Dequeue and Sifting processes in pseudo-code in Appendix I.

3.3 Sentinel Updates

The Sentinel s is a key element that controls the selection of storage elements (Mini-PIFO vs. RCQ) when a packet is enqueued. As described in *Section* 3.2, the sentinel represents the smallest rank in the RCQ and therefore guarantees that the packets that enqueue into the Mini-PIFO have ranks smaller than all packets in the RCQ. To maintain this property, the Sentinel needs to be updated with each eviction from the Mini-PIFO. Since the most recently evicted packet has the smallest rank among the evicted packets, it is always correct to update the Sentinel with the rank of the recently evicted packet[11].

Figure 5 illustrates an example of a sifting process with Sentinel updates. At stage (1), the occupancy of the Mini-PIFO is lower than the "Sifting Threshold" Th_S, which means Sifter needs to sift packets from the earliest non-empty FIFO in the RCQ into the Mini-PIFO. Sifter determines the FIFO that it needs to sift from, which covers the rank range from 20 to 29. To make sure all packets have the opportunity to enter the Mini-PIFO, the Sentinel value s is updated to 29, the highest possible rank in the FIFO. As Sifter transfers packets from the RCQ to the Mini-PIFO, the Mini-PIFO fills up and evicts the packet with the largest rank as shown in stage (2). In this example, the packet with a rank of 28 is evicted and sent back to the RCQ. Note that this packet with a rank of 28 will stay in the RCQ until the next sifting round, which means that other packets with ranks larger than 28 will not enter the Mini-PIFO until then. To achieve this, Sifter updates the Sentinel value s to 28 as described in *Section* 3.2. In stage (3), although the Mini-PIFO is not full, the packet with rank 29 cannot enqueue into the Mini-PIFO since its rank is larger than the Sentinel value s (28). The Sentinel guards the entrance of the Mini-PIFO to keep the packets with larger ranks out and thus guarantees all the packets in the Mini-PIFO have ranks smaller than any packet in the RCQ after each sifting round [12].

3.4 Sifting Threshold

The configuration of the Sifting Threshold Th_S is critical to the performance of Sifter. It serves as a watermark below which the Mini-PIFO is guaranteed to hold the packets with the smallest ranks. As stated in *Section* 3.2, Sifter replenishes the Mini-PIFO when its occupancy is below this threshold using the sifting process that must traverse *all* the packets in the earliest non-empty FIFO of the RCQ. Therefore, this threshold Th_S must be high enough, with enough packets below it to guarantee that the Mini-PIFO will not underrun before the sifting process is complete.

Although it would be reasonable to propose that the sifting threshold Th_S be large, at the limit as large as the size of the Mini-PIFO S_P, which would keep the Mini-PIFO fully utilized, a large sifting threshold would result in too-frequent sifting operations. If Th_S is set as large as S_P, Sifter would trigger the sifting process after each packet dequeue. Since the sifting process requires the transfer of all packets from the earliest non-empty FIFO, frequent sifting rounds would be unnecessarily wasteful, consuming a lot of energy with no performance gain. Therefore, the sifting threshold Th_S must be the smallest value that guarantees that the Mini-PIFO does not underrun.

Figure 6 shows the Mini-PIFO of size S_P and a sifting threshold, Th_S. According to Figure 6, the Mini-PIFO is bisected by the sifting threshold Th_S. The segment below (to the right of) Th_S is guaranteed to be occupied as long as packets are available in the RCQ. As mentioned above, this segment holds enough packets to dequeue before the sifting process is completed. The segment above (to the left of) Th_S determines the interval between two sifting rounds. Typically, the sizes of the FIFOs in the RCQ S_F are larger than the Mini-PIFO S_P, which means the Mini-PIFO would be filled up after each Sifting process and it would take $S_P - Th_S$ dequeues until the Mini-PIFO occupancy triggers the next Sifting. In other words, a larger segment above Th_S leads to less frequent sifting.

We provide the quantitative relationship between the FIFO size S_F, the Mini-PIFO size S_P and the sifting threshold Th_S that guarantees the condition for strict-order scheduling in *Section* 3.5. In *Appendix* C, we evaluate the average num-

[11]Before updating the Sentinel, Sifter compares the current Sentinel s with the rank of the evicted packet and updates s to the smaller rank

[12]Sentinel is initialized to ∞ and will be reset to ∞ if RCQ is empty.

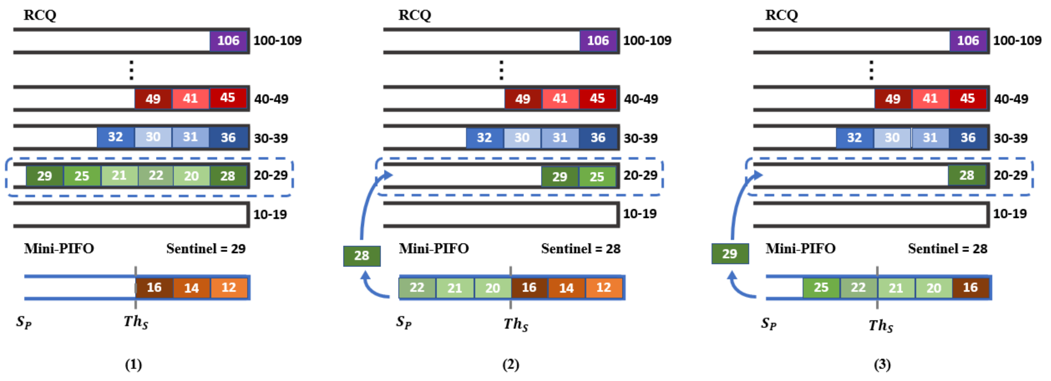


Figure 5: Sifting and Sentinel Updates

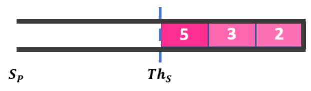


Figure 6: The Mini-PIFO and the Sifting Threshold

ber of extra memory accesses[13] per packet to show that a larger space above the sifting threshold Th_S would decrease the number of extra memory accesses introduced by the sifting process.

3.5 Condition for Inversion-free Scheduling

To guarantee there are no packet inversions in Sifter, we must ensure that each dequeue will output the packet with the smallest rank in the scheduler. According to the dequeue process described in *Section* 3.2, Sifter always dequeues the packet at the head of the Mini-PIFO. Therefore, Sifter needs to guarantee that the Mini-PIFO always holds the packet with the smallest rank at its head. We introduce **Property 1** of Sifter:

Property 1: The Mini PIFO always holds the packet with the smallest rank at its head.

To maintain **Property 1**, Sifter needs to guarantee that all the packets sifted into the Mini-PIFO have smaller ranks than the ones that remain in the RCQ after each sifting process. Thus Sifter needs to traverse all the packets in the earliest FIFO before the original packets in the Mini-PIFO drain out. That is the packets below the sifting threshold Th_S multiplied by the speed-up factor K should be larger than the maximum size of a FIFO S_F in the RCQ. We have:

$$Th_S * K \geq S_F \tag{3}$$

In addition to condition (3), Sifter must ensure that the Mini-PIFO holds at least Th_S packets before traversing through the earliest FIFO in the RCQ.

The size of the Mini-PIFO region between S_P and Th_S determines the number of packets transferred to the Mini-PIFO in each sifting process, assuming the source FIFO has enough packets. When Sifter sifts from a FIFO that has more than $(S_P - Th_S)$ packets, the Mini-PIFO will be loaded with $(S_P - Th_S)$ newly sifted elements at the end of the sifting process. If the earliest FIFO in the RCQ has fewer than $(S_P - Th_S)$ packets, Sifter will keep transferring from the next FIFOs until the Mini-PIFO fill level reaches Th_S as described in *Section* 3.2. Accordingly, Sifter guarantees a minimal Mini-PIFO fill level after each sifting as follows:

$$\min\{(S_P - Th_S), Th_S\} \tag{4}$$

Since Sifter needs to ensure there are at least Th_S packets at the beginning of the sifting process, we have the following additional condition:

$$S_P - Th_S \geq Th_S \tag{5}$$

Rearranging terms:

$$S_P \geq 2 * Th_S \tag{6}$$

Combining conditions (3) and (6), Sifter always guarantees **Property 1**, regardless of the state of the sifting process. Therefore, Sifter guarantees there are no packet inversions.

4 Sifter Hardware Prototype

Figure 7 shows an abstraction of the system-level application of a programmable packet scheduler.

Usually, the size of the packet descriptor[14] is much smaller than the packet itself. Thus, packet schedulers in high-speed data planes need to process only the descriptors rather than the packets themselves due to the limited packet processing time. According to Figure 7, there are two paths in the system: the *Packet Path* and the *Descriptor Path*. In the *Packet*

[13] Average number of extra memory accesses: the average number of extra memory access of a packet descriptor from the time it enters and until it leaves the packet scheduler when compared with a simple FIFO queue.

[14] In general, a packet descriptor in the data plane includes metadata related to the packet, including length, flow ID, memory pointer, etc.)

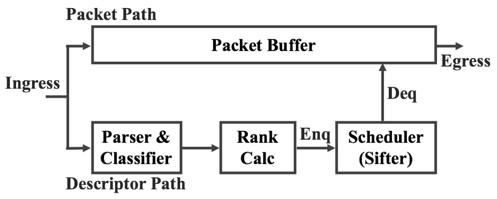


Figure 7: System-Level Packet Scheduler Application

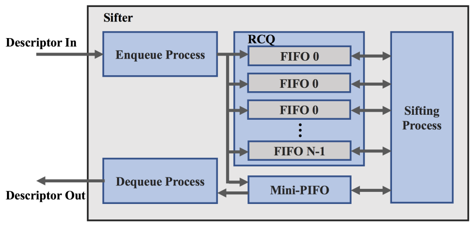


Figure 8: Sifter Block Diagram, VHDL Implementation

Path, the ingress packets are stored in the packet buffer and wait to be dequeued or dropped. The packet scheduling takes place in the *Descriptor Path*. The packet classifier and parser extract the packet information from the packet header and generate a packet descriptor by combining it with the memory address of the packet in the buffer. Next, the packet rank calculator computes the rank of the packet according to the applicable scheduling algorithm and updates the descriptor. Finally, the system enqueues the descriptor into the packet scheduler[15], which drops or dequeues descriptors according to their ranks. When the scheduler dequeues or drops a packet descriptor, the system accesses the associated packet in the buffer, referenced by its memory address, and dequeues or drops the corresponding packet. Since this paper focuses on programmable packet scheduling, the hardware prototype design section also targets the *Descriptor Path* in the system.

Hardware Prototype Architecture We implemented the Sifter hardware prototype in VHDL according to the algorithms in *Section* 3. Figure 8 shows a block diagram of the Sifter hardware prototype. The RCQ and Mini-PIFO hold descriptors during the enqueue and dequeue operations. The Enqueue block stores the packet in one of the FIFOs in the RCQ or in the Mini-PIFO according to the descriptor's rank. The Sifting process transfers the descriptors between the RCQ and the Mini-PIFO to ensure that those with the smallest ranks are held in the Mini-PIFO. Finally, the Dequeue block outputs the descriptors with the smallest rank.

There are multiple scaling parameters in the VHDL implementation of Sifter, including: (1) the number and size of FIFOs in the RCQ, (2) the size and sifting threshold of the Mini-PIFO, and (3) other sizing parameters for memories and logic. These parameters improve the flexibility and scalability of our design[16].

We evaluate the performance impact and the resource overhead of adjusting different design parameters in *Appendix* D.

We designed a 72-bit packet descriptor[17] for the Sifter hardware prototype, and Sifter schedules packets according to a 20-bit *Packet Rank* field.

[15]Sifter is a specific implementation of such a packet scheduler.

[16]The VHDL code, test bench, and FPGA implementation files are available at *https://github.com/Sifter-NSDI24/Sifter-NSDI24*

[17]The details of the packet descriptor are revealed in *Appendix* E.

Rank Computation We programmed SFQ[26], a virtual-clock-based fairness packet scheduling scheme, on the Sifter hardware prototype. The transmission time is pre-calculated by the packet classifier and stored in the *Packet Rank* field in the packet descriptor. The *Enqueue Process* module updates the *Packet Rank* by the system virtual clock and the last rank of the corresponding flow.

According to the definition of programmable packet scheduling in *Section* 2.1, most packet scheduling schemes compute the ranks in the end-hosts or flow tables prior to entering the packet scheduler[35] [48] [49]. Therefore, it would be straightforward to extend the Sifter prototype to support a wide variety of packet scheduling schemes with the pre-calculated ranks in the packet descriptors.

Fast RCQ FIFO Indexing For each packet descriptor that enqueues into the RCQ module, Sifter needs to find the FIFO associated with its rank. We call this operation FIFO indexing. RCQ performs FIFO indexing when a new packet descriptor arrives or when a packet descriptor is evicted from the Mini-PIFO, which occurs whenever the Mini-PIFO becomes full and additional descriptors need to be stored in the RCQ. Thus, FIFO indexing must be extremely fast in the hardware implementation.

Rather than scanning through the rank bounds of each FIFO, RCQ directly finds the corresponding FIFO index by extracting a range of bits in the *Packet Rank*. We configured the number of FIFOs F and the granularity g as powers of two to be able to use bit slice selection and shifting operations in the hardware implementation rather than slower math operations, such as multiplication[18].

Mini-PIFO The Mini-PIFO is implemented using a shift register structure and performs a push operation in two clock cycles. In the first clock cycle when a new descriptor is pushed, its rank is compared in parallel to those of all descriptors stored in the Mini-PIFO and an insertion position is determined. In the second clock cycle, the descriptors with higher ranks are shifted one position towards the tail, and the new descriptor is inserted in the newly created “hole”. In the process of shifting towards the tail, one descriptor may be evicted and stored in the RCQ if the Mini-PIFO was previously full. The pop operation simply outputs the descriptor

[18]Additional details of fast FIFO indexing are provided in *Appendix* E

at the head of the Mini-PIFO and shifts all other descriptors toward the head. Once the Mini-PIFO fill level reaches the reload threshold, the sifting process starts a reload operation to replenish the PIFO from the RCQ.

The Mini-PIFO can also handle simultaneous push and pop operations, whether the arriving descriptor needs to be popped immediately or inserted in a Mini-PIFO location.

In addition, the Mini-PIFO has a logic "wrapper" that handles the resolution of race conditions that may occur if enqueue and reload operations generate simultaneous push operations toward the PIFO.

Search for The First Non-empty FIFO When the Mini-PIFO triggers a sifting process, RCQ finds the earliest non-empty FIFO queue to replenish the Mini-PIFO with packet descriptors. The FIFOs in the RCQ are organized as a circular list with an index indicating the current FIFO (earliest time range) corresponding to the current value of the virtual clock. The search for the first non-empty FIFO is performed in parallel on two groups of FIFOs: (1) First Group: From the current FIFO to the highest numbered FIFO in the range; (2) Second Group: From FIFO zero to the FIFO preceding the current FIFO.

Because of the circular structure, FIFOs in the First Group hold descriptors that have smaller ranks than those in the Second Group. Therefore, if the first non-empty FIFO is found in the First Group, it is selected as the next non-empty FIFO. Otherwise, the first non-empty FIFO in the Second Group is selected.

5 Evaluation

In *Section* 5.1 and 5.2, we evaluate Sifter using the NS3 [37] packet-based simulator and show the advantages of Sifter in fairness, FCT, and delay due to the elimination of packet scheduling inversions. Furthermore, we demonstrate the flexibility of Sifter with two use cases. Finally, in *Section* 5.3 we implement Sifter on an AMD/Xilinx Alveo U250 FPGA board and evaluate its functionality, performance as well as resource utilization.

5.1 Micro-benchmark Evaluation

We first set up a micro-benchmark to evaluate the performance of Sifter in detail at the packet level. The evaluated performance metrics include fairness in short-term bandwidth allocation shown in *Section* 5.1.2, the FCT and end-to-end delay given in *Appendix* F, as well as the average extra number of memory accesses shown in *Appendix* C.

5.1.1 Micro-benchmark setup

Topology Given that the micro-benchmark aims to assess the intricate packet-level performance of schedulers, it is advisable to maintain a compact topology scale. This approach facilitates a more straightforward observation of detailed metrics. In the micro-benchmark, we set up a single-node star topology with multiple end hosts connected via one switch with programmable packet schedulers. All links have a bandwidth of 10 Gbps and a delay of $3\mu s$.

Traffic We prepared two sets of traffic for different evaluation proposes: (1) A set with small-scale traffic that contains 8 flows to evaluate the convergence of fair bandwidth allocation and (2) A set with large-scale traffic that contains approximately $1k$ flows to evaluate the FCT and end-to-end delay. To create network congestion on the bottleneck link, we generate the TCP flows[19] in an in-cast traffic pattern.

Schedulers Setup We applied packet schedulers on the bottleneck link to evaluate their performance under congestion conditions. The evaluated schedulers include Sifter, Programmable Calendar Queues (PCQ) [45] and SP-PIFO [2]. Each of the schedulers has 16 FIFO queues with a depth of 64[20]. Sifter has a mini-PIFO with a capacity of 32 descriptors. In addition, we also introduce an ideal PIFO[48][49] with a capacity of 1024 and a standalone Mini-PIFO with a capacity of 32 to serve as benchmarks.

5.1.2 Convergence to fair bandwidth allocation

There are 8 TCP flows in the micro-benchmark test with the small-scale traffic, with each flow becoming active at successive 2 *ms* intervals. Flow 8 stops 2 *ms* after its arrival, and other flows stop progressively in reverse order of their arrival at intervals of 2 *ms*. To evaluate the fairness of short-term bandwidth allocation, we measure the throughput of each TCP flow in a window size of 120 μs, which is ten times the RTT. The evaluation results are shown in Figure 9.

Sifter converges to fairness in bandwidth allocation Figure 9 shows that Sifter has a fair bandwidth allocation close to that of an ideal PIFO. The 8 TCP connections in Sifter converge to the fair share of bandwidth very fast and with minimum fluctuations. On the other hand, PCQ and SP-PIFO experience packet inversions, which result in fluctuations in the short-term throughput as shown in Figure 9. The fluctuation is due to the fact that PCQ and SP-FIFO store packets in FIFOs. Packets in the same FIFO are scheduled on a first-come-first-served basis, which means that multiple packets from a flow could continuously enter the same FIFO in a short burst, and later leave the scheduler continuously in a burst. Such burstiness impairs the short-term fairness of bandwidth allocation and the steadiness of the throughput.

Sifter eliminates packet inversions In the simulation results, we observe no inversions in Sifter while PCQ and SP-PIFO exhibit a significant number of packet inversions with large magnitudes due to their FIFO-based architectures. The

[19]We use TCP New Reno in our micro-benchmark simulation

[20]An Insufficient FIFO size can result in sub-optimal performance due to overflows for both FIFO-based schedulers and Sifter. We will discuss the overflows in Sifter in Appendix B.

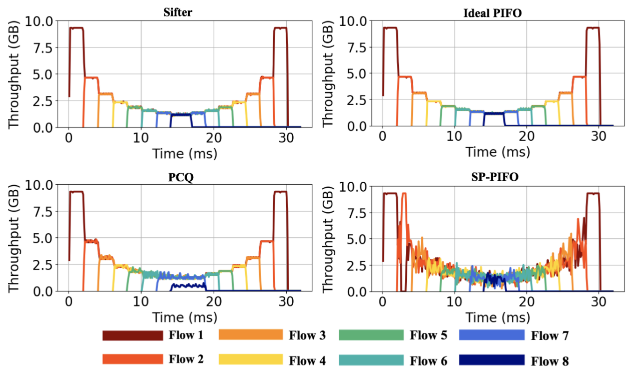


Figure 9: Convergence for fair bandwidth allocation

consequence of inversions is reflected in the fluctuations of flow throughput as shown in Figure 9.

5.2 Large-scale Packet-based Simulation

To evaluate the performance of packet schedulers in data-center environments, we use large-scale packet-based simulations with a fat-tree topology with a variety of larger-scaled empirical traffic loads.

We evaluate two use cases of programmable packet schedulers: (1) Weighted max-min fairness in bandwidth allocation with SFQ [26] in *Section* 5.2.2 and (2) Minimizing FCT with modified SRPT[3] in *Appendix* G.

5.2.1 Simulation Setup

Network topology We extend the topology to a 3-tier fat-tree consisting of 4 core switches, 8 aggregation switches, and 8 ToR switches. All the switch nodes are connected with $40 Gbps$ links with $1 \mu s$ delay. A total of 256 hosts are connected by the fat-tree topology. Each ToR switch connects with 32 hosts and the links between each host and the ToR switch each have a bandwidth of $10 Gbps$ with $10 ns$ delay.

Simulation Traffic We generate two types of empirical traffic with different patterns: (1) *random pattern* and (2) *incast pattern*. For the *random pattern*, each generated TCP flow randomly selects a source node and a destination node among the 256 hosts in the topology, while the flows in the *incast pattern* randomly select a source node but always have a fixed destination node to represent an in-cast traffic pattern. The arrival time of each TCP flow follows a Poisson distribution, and the flow size follows the Web-search distribution in data centers [3].

5.2.2 Use Case: Start-time Fair Queuing for weighted max-min fairness in bandwidth allocation

Weighted max-min fairness in bandwidth allocation is one of the most important goals of packet scheduling [46] [6] [41] [32] [36]. Among the algorithms derived from Weighted Fair Queueing (WFQ), Start-time Fair Queueing (SFQ) [26] is the most popular one due to its simplicity and accuracy. Therefore, we applied SFQ to all the schedulers in the simulation. We configured all three schedulers (Sifter, PCQ, and SP-PIFO) with 32 FIFO queues with a depth of 256 entries. In addition, Sifter has a Mini-PIFO with 64 entries[21]. Furthermore, we set two PIFO benchmarks with different sizes: an ideal PIFO with a large capacity of 1024 entries and a standalone Mini-PIFO with 64 entries to match the Mini-PIFO in Sifter.

Sifter is a close approximation to the ideal PIFO Figure 10 $(a)-(d)$ shows the Normalized FCT[22] performance of the different packet schedulers under *random pattern* and *in-cast pattern* respectively. As shown in Figure 10, Sifter has an FCT performance that is nearly identical to the ideal PIFO. Similarly, the end-to-end delay performance of Sifter matches closely that of the ideal PIFO. Figure 10 $(e)-(h)$ shows the 95th percentile end-to-end delay of the small flows ($\leq$50MB) with different setups. By providing an accurate scheduling order, Sifter guarantees that packets with smaller ranks are scheduled without the extra delay that would be caused by scheduling inversions.

Compared with the standalone Mini-PIFO, the RCQ of Sifter extends the buffer capacity and thus reduces the packet loss rate for mid-sized flows. As Figure 10 $(a)-(d)$ shows, Sifter outperforms the Mini-PIFO among the mid-sized flows by reducing packet loss and re-transmissions.

Sifter is stable under different traffic patterns Approximate packet schedulers are subject to the impacts of certain traffic patterns while Sifter achieves a stable performance close to that of the ideal PIFO.

As shown in Figure 10, PCQ has poor FCT and tail-latency performance for small-sized flows for both *random* and *in-cast* patterns. As illustrated in *Section* 2.3.1, the FIFO-based calendar queue structure of PCQ introduces inversions and extra delays for packets within one FIFO queue. The packets with small ranks from the short flows are significantly affected by the extra delay, which results in significant performance degradation for small flows between PCQ and other schedulers.

Although SP-PIFO provides decent performance with the *random pattern*, its performance degrades as the congestion level increases for the *in-cast pattern*. Queue-bound adjustments in SP-PIFO tend to put packets with larger ranks into the queues with lower priorities. Since SFQ is a virtual-clock-based scheduling algorithm where the ranks of incoming packets keep increasing as the congestion persists, SP-PIFO stores most of the packets in the last few FIFO queues. We observe that SP-PIFO does not fully utilize all of its

[21]In the Mini-PIFO of Sifter, we set the Sift threshold $Th_S = 32$ (half of the PIFO size), and used a speed-up factor $K = 4$ to fulfill the conditions in *Section* 3.5.

[22]Normalized FCT: The measured FCT normalized to the ideal FCT.

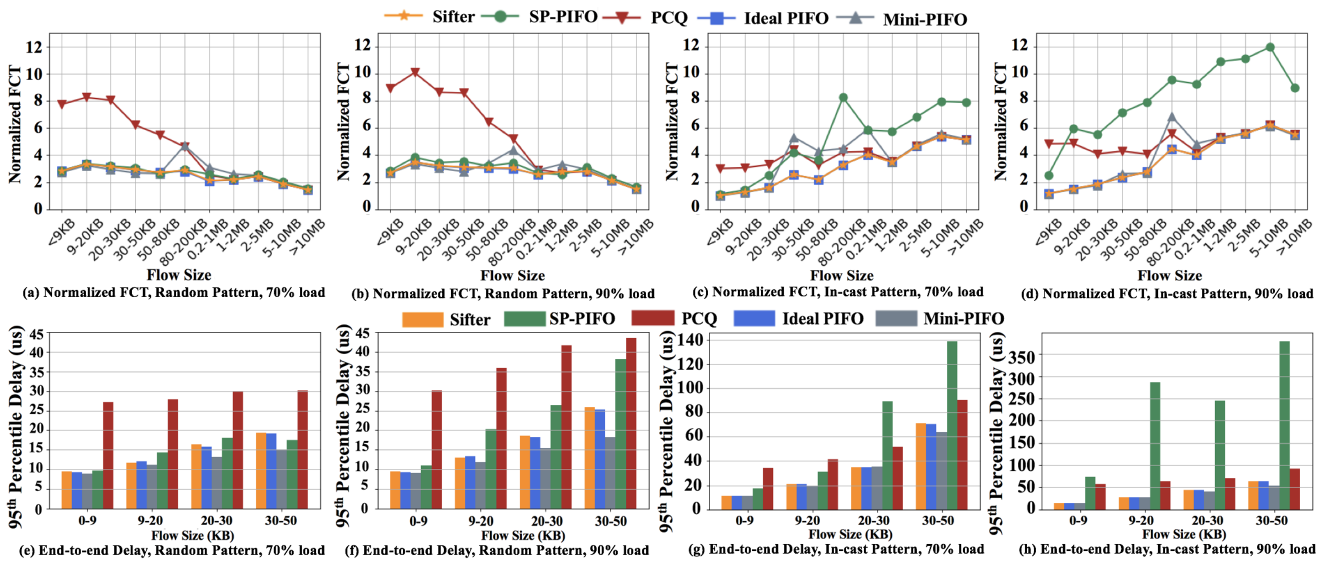


Figure 10: Normalized FCT and End-to-End Delay with SFQ

available FIFO queues, and that the last few FIFOs become particularly crowded under heavy congestion. This queue congestion increases the end-to-end delay as Figure 10 (g) and (h) shows and degrades the FCT of large flows as shown in Figure 10 (c) and (d).

5.3 FPGA Hardware Testbed

We built a hardware testbed for the Sifter prototype on an AMD/Xilinx Alveo U250 board [4], which uses an FPGA similar in size to an UltraScale+ VU13P with mid-speed grade. The hardware evaluation of the Sifter prototype covers three aspects: (1) Line rate measurement; (2) Verification of no inversions in the output of Sifter to validate error-free scheduling; (3) Evaluation of the hardware resources for different configurations by varying the design parameters.

Hardware Testbed Structure According to *Section* 4, we focus on the descriptor path in the system. Therefore, in our hardware testbed evaluation, we feed packet descriptors to our hardware prototype and scale their input and output rates to achieve a line rate of 100*Gbps*.

Figure 11 shows the structure of the hardware testbed. We use two sets of packet descriptors parsed from a PCAP file.

For simulation purposes, we implemented the testbed in Cocotb [21], a Python-based HDL simulation environment. For the hardware testing, we implemented the same testbed using Exegy's nxFramework [22], which provides the PCIe connectivity and C++ device drivers to access registers and memories within Sifter.

The input packet descriptors are originally generated from a PCAP packet capture [33]. We load the input packet descriptors in the *Input Buffer* module on the FPGA, along with time gaps between packets to set a line rate equivalent to 100*Gbps*. The *Enq Rate Ctl* module reads the descriptors from the input buffer and enqueues them into Sifter at the specified rate. After a software-programmable delay, the *Deq Rate Ctl* module starts dequeuing packet descriptor from Sifter at a rate equivalent to 100*Gbps*, by using the packet length information to control the descriptor dequeue rate. Dequeued descriptors are written to the Output Recorder block along with timestamps and read out and written to an output file by software at the end of the test. We analyze the packet descriptor output file from the *Output Recorder* to measure the line rate and validate that there are no packet inversions or missing descriptors.

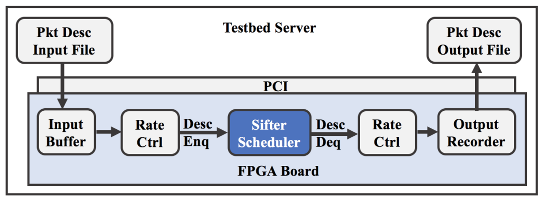
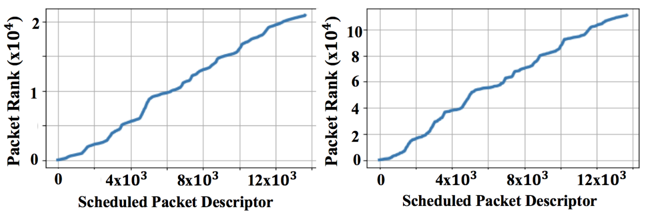


Figure 11: Sifter Hardware Testbed

Running Sifter on an FPGA We configured Sifter with 32 FIFOs with 32 locations each and a Mini-PIFO with 32 locations. We used Vivado 2021.2 to generate a bitstream of the test bed including the Exegy PCIe memory-mapped I/O (MMIO) logic and loaded it on an AMD/Xilinx Alveo U250 board [4]. With the above design parameters, the Sifter prototype achieved a frequency of 322MHz and used less than 1.25% of the FPGA. [23] We used C++ code and the Exegy MMIO device drivers to write the input data into the *Input Buffer* and read the output data from the *Output Recorder*.

[23] Additional information about the trade-off of different design parameters with FPGA resource overhead is given in *Appendix* D.2.



(a) Scheduling Order, fixed packet size (b) Scheduling Order, varying packet size

Figure 12: Inversion-free Scheduling Order of Sifter Prototype

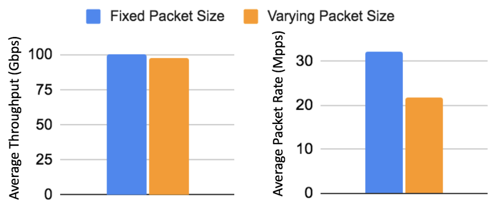


Figure 13: Sifter Hardware Throughput and Packet Rate

Traffic Trace We generated the packet descriptor inputs by parsing a PCAP packet capture, which includes 13708 packets and 941 flows. The average packet size of the input trace is 823 Bytes. With the max frequency of 322 MHz of its FPGA prototype, Sifter can achieve a line rate of 100Gbps with packets larger than 370 Bytes[24].Therefore, we modified the packet size of the PCAP file and generated two sets of packet descriptor inputs: (1) A fixed packet size input where all the packets have the same size of 370 Bytes and (2) A varying packet size input by modifying the small packets size to 370 Bytes while preserving the original packet sizes of other packets that are larger than 370 Bytes

Inversion-free Packet Scheduling Figure 12 shows the packet descriptor scheduling order from the output of our testbed. For both fixed and varying packet size traces, there are no inversions in the packet scheduling order and the packet loss rate[25] of both input traces is lower than 0.6%.

Scaling to 100Gbps Line Rate Figure 13 shows the throughput and the packet rate of Sifter's hardware prototype under fixed and varying packet size inputs. From the output of our hardware testbed, Sifter reaches the line rate of 100 Gbps for both types of input. For the fixed packet size input, the Sifter prototype operates at a constant packet rate of 32.2 Mpps, since the packet processing time of Sifter is 10 clock cycles. On the other hand, for the varying packet size input, the line rate is also $100Gbps$ with an average packet rate of 21.6 Mpps because the average packet size is larger.

[24]This packet length allows Sifter to dequeue at $100Gbps$ without inversions on our prototype. With a higher clock frequency, Sifter supports a $100Gbps$ line rate with 64-byte packets. See *Appendix* E and H for details

[25]Packet loss may occur if FIFOs become full and is not related to Sifter's operation. See *Appendix* B for more information on overflows in Sifter.

6 Related Work

In the 1990s, early implementations of general packet schedulers such as Sequencer [13] [14] [15] sort packets accurately by their ranks based on a shift-register design. However, the architecture of these ASIC-based packet schedulers limits their capacity. In the 2000s, pipeline-Heap (pHeap) [7][29] was proposed, which serves as an accurate packet scheduler that supports a large buffer capacity. However, the high implementation overhead of pHeap takes up a significant area on the switch chip. This issue would be compounded if a pHeap is used for each port on the switch.

In the 2010s, the advent of the programmable data plane brought to light the topic of implementing general packet schedulers in academia and industry. PIFO [48] [49] is a fundamental building block for programmable packet schedulers. However, while its design achieves accurate programmable packet scheduling with a small chip area overhead, it is still limited by its capacity and the requirement of special hardware support such as Ternary Content-Addressable Memory (TCAM).

PCQ [45] and SP-PIFO [2] are two approximate programmable packet schedulers based on strict-priority queues. While they greatly reduce implementation overhead and boost the capacity of the scheduler, the impact of the packet inversions they introduce is not negligible. Approximate Fair Queuing (AFQ) [44] and Gearbox [24] are two other approximate packet schedulers focused on fair queuing and weighted fair queuing, but are not generalized for programmable packet scheduling.

AIFO [54] is a programmable packet scheduler design, which only requires a single FIFO queue and filters the incoming packets using admission control. However, AIFO can cause packet inversions similar to the above-mentioned approximate packet schedulers, and scaling it to support a large buffer capacity would be challenging.

7 Conclusion

In this paper, we present Sifter, an accurate, programmable packet scheduler that operates free of packet inversions and supports a large buffer size with low implementation overhead. By taking advantage of the "Speed-up Factor", Sifter sorts packets by "Sift Sorting" and combines the advantages of accurate scheduling of a PIFO with the large capacity provided by FIFO-based approximate schedulers. Our simulations in NS3 show that Sifter achieves better fairness, lower FCT, and delay by eliminating packet inversions. We implemented Sifter in VHDL, targeting an AMD/Xilinx Alveo U250 FPGA card [4] with a mid-speed grade XCVU13P FPGA. Our FPGA-based hardware prototype operates at 322 MHz and reaches a line rate of 100 Gb/s for packets larger than 370 bytes. Sifter uses less than 1.25% of the FPGA resources.

References

[1] Addanki, V., Apostolaki, M., Ghobadi, M., Schmid, S., and Vanbever, L. Abm: active buffer management in datacenters. In *Proceedings of the ACM SIGCOMM 2022 Conference* (2022), pp. 36–52.

[2] Alcoz, A. G., Dietmüller, A., and Vanbever, L. Sp-pifo: Approximating push-in first-out behaviors using strict-priority queues. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)* (2020), pp. 59–76.

[3] Alizadeh, M., Yang, S., Sharif, M., Katti, S., McKeown, N., Prabhakar, B., and Shenker, S. pfabric: Minimal near-optimal datacenter transport. In *ACM SIGCOMM Computer Communication Review* (2013), vol. 43, ACM, pp. 435–446.

[4] AMD/Xilinx. AMD/Xilinx Alveo U250 Data Center Accelerator Card. `https://www.xilinx.com/products/boards-and-kits/alveo/u250.html`, 2021.

[5] AMD/Xilinx. Vivado Design Suite, Integrated Design Environment. `https://www.xilinx.com/products/design-tools/vivado.html`, 2021.

[6] Ballani, H., Costa, P., Karagiannis, T., and Rowstron, A. Towards predictable datacenter networks. In *ACM SIGCOMM computer communication review* (2011), vol. 41, ACM, pp. 242–253.

[7] Bhagwan, R., and Lin, B. Fast and scalable priority queue architecture for high-speed network switches. In *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No. 00CH37064)* (2000), vol. 2, IEEE, pp. 538–547.

[8] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review 44*, 3 (2014), 87–95.

[9] Bosshart, P., Gibb, G., Kim, H.-S., Varghese, G., McKeown, N., Izzard, M., Mujica, F., and Horowitz, M. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. *ACM SIGCOMM Computer Communication Review 43*, 4 (2013), 99–110.

[10] Broadcom. Broadcom StrataDNX™ BCM88480 Traffic Management Architecture. `https://docs.broadcom.com/doc/88480-DG1-PUB`, 2021.

[11] Broadcom. High Capacity StrataXGS®Trident II Ethernet Switch Series. `http://www:broadcom:com/products/Switching/Data-Center/BCM56850-Series.`, 2021.

[12] Brown, R. Calendar queues: a fast 0 (1) priority queue implementation for the simulation event set problem. *Communications of the ACM 31*, 10 (1988), 1220–1227.

[13] Chao, H. J. Architecture emersesign for regulating and scheduling user's traffic in atm networks. In *ACM SIGCOMM Computer Communication Review* (1992), vol. 22, ACM, pp. 77–87.

[14] Chao, H. J., Cheng, H., Jenq, Y.-R., and Jeong, D. Design of a generalized priority queue manager for atm switches. *IEEE Journal on Selected Areas in Communications 15*, 5 (1997), 867–880.

[15] Chao, H. J., Jenq, Y.-R., Guo, X., and Lam, C.-H. Design of packet-fair queuing schedulers using a ram-based searching engine. *IEEE Journal on Selected Areas in Communications 17*, 6 (1999), 1105–1126.

[16] Choudhury, A. K., and Hahne, E. L. Buffer management in a hierarchical shared memory switch. In *Proceedings of INFOCOM'94 Conference on Computer Communications* (1994), IEEE, pp. 1410–1419.

[17] Cisco. Cisco Nexus 7700 F3-Series 12-Port 100 Gigabit Ethernet Module Data Sheet. `https://www.cisco.com/c/en/us/products/collateral/switches/nexus-7000-series-switches/data_sheet_c78-728423.html`, 2019.

[18] Cisco. Cisco Silicon One Product Family White Paper. `https://www.cisco.com/c/en/us/solutions/silicon-one.html`, 2021.

[19] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to algorithms*. MIT press, 2009.

[20] Demers, A., Keshav, S., and Shenker, S. Analysis and simulation of a fair queueing algorithm. In *ACM SIGCOMM Computer Communication Review* (1989), vol. 19, ACM, pp. 1–12.

[21] development group, C. Cocotb chip design testbench. In *https://www.cocotb.org/*. 2023, 2023, p. 1.

[22] Exegy. Exegy nxframework. In *https://www.enyx.com/nxframework/*. 2022, 2022, p. 1.

[23] Feng, Y., Chen, Z., Song, H., Xu, W., Li, J., Zhang, Z., Yun, T., Wan, Y., and Liu, B. Enabling in-situ programmability in network data plane: From architecture to language. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)* (2022), pp. 635–649.

[24] Gao, P., Dalleggio, A., Xu, Y., and Chao, H. J. Gearbox: A hierarchical packet scheduler for approximate weighted fair queuing. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)* (2022), pp. 551–565.

[25] Golestani, S. J. A self-clocked fair queueing scheme for broadband applications. In *Proceedings of INFOCOM'94 Conference on Computer Communications* (1994), IEEE, pp. 636–646.

[26] Goyal, P., Vin, H. M., and Chen, H. Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. In *ACM SIGCOMM Computer Communication Review* (1996), vol. 26, ACM, pp. 157–168.

[27] Hogan, M., Landau-Feibish, S., Arashloo, M. T., Rexford, J., and Walker, D. Modular switch programming under resource constraints. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)* (2022), pp. 193–207.

[28] Hong, C.-Y., Caesar, M., and Godfrey, P. Finishing flows quickly with preemptive scheduling. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication* (2012), ACM, pp. 127–138.

[29] Ioannou, A., and Katevenis, M. G. Pipelined heap (priority queue) management for advanced scheduling in high-speed networks. *IEEE/ACM Transactions on Networking (ToN) 15*, 2 (2007), 450–461.

[30] Jang, K., Sherry, J., Ballani, H., and Moncaster, T. Silo: Predictable message latency in the cloud. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (2015), pp. 435–448.

[31] Jeyakumar, V., Alizadeh, M., Mazières, D., Prabhakar, B., Greenberg, A., and Kim, C. {EyeQ}: Practical network performance isolation at the edge. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (2013), pp. 297–311.

[32] Jeyakumar, V., Alizadeh, M., Mazieres, D., Prabhakar, B., and Kim, C. Eyeq: Practical network performance isolation for the multi-tenant cloud. In *Presented as part of the* (2012).

[33] Klassen, F., and AppNeta. Tcpreplay. In *https://tcpreplay.appneta.com/wiki/captures.html*. 2023, 2023, p. 1.

[34] Leung, J. Y.-T. A new algorithm for scheduling periodic, real-time tasks. *Algorithmica 4*, 1-4 (1989), 209.

[35] MITTAL, R., AGARWAL, R., RATNASAMY, S., AND SHENKER, S. Universal packet scheduling. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (2016), pp. 501–521.

[36] NAGARAJ, K., BHARADIA, D., MAO, H., CHINCHALI, S., ALIZADEH, M., AND KATTI, S. Numfabric: Fast and flexible bandwidth allocation in datacenters. In *Proceedings of the 2016 ACM SIGCOMM Conference* (2016), ACM, pp. 188–201.

[37] NETWORK SIMULATOR DEVELOPMENT GROUP, T. The network simulator 3. In *https://www.nsnam.org/*. 2022, 2022, p. 1.

[38] OPEN, H.-L. L. F. D. F.-R. S. F. P. N. P. Broadcom. np. In *https://nplang.org/*. 2019, 2019, p. 1.

[39] PAREKH, A. K., AND GALLAGER, R. G. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM transactions on networking*, 3 (1993), 344–357.

[40] PAREKH, A. K., AND GALLAGER, R. G. A generalized processor sharing approach to flow control in integrated services networks: the multiple node case. *IEEE/ACM transactions on networking 2*, 2 (1994), 137–150.

[41] POPA, L., KUMAR, G., CHOWDHURY, M., KRISHNAMURTHY, A., RATNASAMY, S., AND STOICA, I. Faircloud: sharing the network in cloud computing. *ACM SIGCOMM Computer Communication Review 42*, 4 (2012), 187–198.

[42] RUFFY, F., WANG, T., AND SIVARAMAN, A. Gauntlet: Finding bugs in compilers for programmable packet processing. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (2020), pp. 683–699.

[43] SCHRAGE, L. A proof of the optimality of the shortest remaining processing time discipline. *Operations Research 16*, 3 (1968), 687–690.

[44] SHARMA, N. K., LIU, M., ATREYA, K., AND KRISHNAMURTHY, A. Approximating fair queueing on reconfigurable switches. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)* (2018), pp. 1–16.

[45] SHARMA, N. K., ZHAO, C., LIU, M., KANNAN, P. G., KIM, C., KRISHNAMURTHY, A., AND SIVARAMAN, A. Programmable calendar queues for high-speed packet scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)* (2020), pp. 685–699.

[46] SHIEH, A., KANDULA, S., GREENBERG, A. G., KIM, C., AND SAHA, B. Sharing the data center network. In *NSDI* (2011), vol. 11, pp. 23–23.

[47] SHREEDHAR, M., AND VARGHESE, G. Efficient fair queuing using deficit round-robin. *IEEE/ACM Transactions on networking*, 3 (1996), 375–385.

[48] SIVARAMAN, A., SUBRAMANIAN, S., AGRAWAL, A., CHOLE, S., CHUANG, S.-T., EDSALL, T., ALIZADEH, M., KATTI, S., MCKEOWN, N., AND BALAKRISHNAN, H. Towards programmable packet scheduling. In *Proceedings of the 14th ACM workshop on hot topics in networks* (2015), ACM, p. 23.

[49] SIVARAMAN, A., SUBRAMANIAN, S., ALIZADEH, M., CHOLE, S., CHUANG, S.-T., AGRAWAL, A., BALAKRISHNAN, H., EDSALL, T., KATTI, S., AND MCKEOWN, N. Programmable packet scheduling at line rate. In *Proceedings of the 2016 ACM SIGCOMM Conference* (2016), ACM, pp. 44–57.

[50] STEPHENS, B., SINGHVI, A., AKELLA, A., AND SWIFT, M. Titan: Fair packet scheduling for commodity multiqueue nics. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (2017), pp. 431–444.

[51] VARGHESE, G., AND LAUCK, A. Hashed and hierarchical timing wheels: efficient data structures for implementing a timer facility. *IEEE/ACM transactions on networking 5*, 6 (1997), 824–834.

[52] WILSON, C., BALLANI, H., KARAGIANNIS, T., AND ROWTRON, A. Better never than late: Meeting deadlines in datacenter networks. In *ACM SIGCOMM Computer Communication Review* (2011), vol. 41, ACM, pp. 50–61.

[53] XING, J., HSU, K.-F., KADOSH, M., LO, A., PIASETZKY, Y., KRISHNAMURTHY, A., AND CHEN, A. Runtime programmable switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)* (2022), pp. 651–665.

[54] YU, Z., HU, C., WU, J., SUN, X., BRAVERMAN, V., CHOWDHURY, M., LIU, Z., AND JIN, X. Programmable packet scheduling with a single queue. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (2021), pp. 179–193.

[55] YUAN, Y., ALAMA, O., FEI, J., NELSON, J., PORTS, D. R., SAPIO, A., CANINI, M., AND KIM, N. S. Unlocking the power of inline {Floating-Point} operations on programmable switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)* (2022), pp. 683–700.

[56] ZENO, L., PORTS, D. R., NELSON, J., KIM, D., LANDAU-FEIBISH, S., KEIDAR, I., RINBERG, A., RASHELBACH, A., DE-PAULA, I., AND SILBERSTEIN, M. {SwiSh}: Distributed shared state abstractions for programmable switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)* (2022), pp. 171–191.

[57] ZHANG, H., AND BENNETT, J. C. Wf2q: worst-case fair weighted fair queueing. In *IEEE INFOCOM* (1996), vol. 96, pp. 120–128.

Appendix

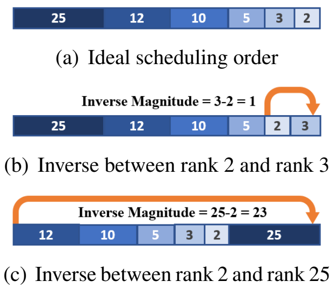
A Inversion Magnitude

In *Section* 2.2 we defined *Packet Inversions*. However, the number of packet inversions alone cannot fully reflect the severity of errors that an approximating packet scheduler may introduce. To quantify the severity of each packet inversion, we extend the concept of packet inversion with *Inversion Magnitude*.

Inversion Magnitude: When a packet with rank r departs from the scheduler and a packet with smaller rank r'[26] exists in the scheduler, the inversion magnitude is $r - r'$.

Take the example in Figure 14: There are packet inversions in both scenarios shown in Figure 14 (b) and (c), but the severities are different. In the case of Figure 14 (c), a large packet with rank = 25 is scheduled prior to a packet with a smaller rank = 2. When compared with the case of Figure 14 (b), packets with smaller ranks experience longer delays in Figure 14 (c).

As mentioned in *Section* 2.2.1, packet inversions with larger magnitudes lead to more severe consequences. For scheduling algorithms such as SCFQ [25] and SFQ [26], which update a virtual clock using packet departure times, a larger inversion magnitude leads to a worse skew of the virtual clock and results in worse throughput loss or additional delay on newly arrived flows. For the scheduling disciplines that aim to minimize tail packet delay, a larger inversion magnitude can cause packets with the least slack time to experience longer delays.



(a) Ideal scheduling order

(b) Inverse between rank 2 and rank 3

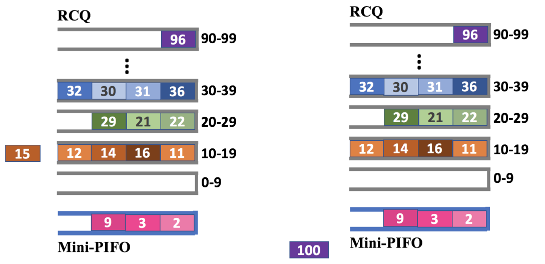
(c) Inverse between rank 2 and rank 25

Figure 14: Packet Inversion and Inversion Magnitude

B Overflows in Sifter

There are two types of overflows in Sifter: (1) *FIFO Overflow* and (2) *Calendar Queue Overflow*.

FIFO Overflow *FIFO overflows* are the cases where a packet is dropped when the associated FIFO is full. As shown in Figure 15 (a), a packet with rank 15 is dropped



(a) FIFO Overflow (b) Calendar Queue Overflow

Figure 15: Overflows in Sifter

since the FIFO that covers the rank range from 10 to 19 is already full. Such overflows lead to a different outcome when compared with an ideal PIFO. PIFO always drops the packet with the largest rank when the capacity is full, while Sifter may drop packets with relatively smaller ranks in the cases of *FIFO overflow*.

The solutions to mitigate *FIFO overflow* is straightforward: (1) extend the size of each FIFO in the RCQ or (2) reduce the FIFO granularity g so that each FIFO covers a narrower rank range, which reduces the number of packets in each FIFO. In *Appendix* D, we show that the above adjustments in the design parameters reduce the FIFO overflows.

Calendar Queue Overflow A *Calendar Queue Overflow*, occurs when an incoming packet has a rank value larger than the maximum rank that the scheduler covers. Figure 15 (b) presents an example of *Calendar Queue Overflow*, where the rank value of an incoming packet is 100 and the maximum rank value that the scheduler supports is 99. Such overflows have a similar effect as overflows in PIFOs, where packets with the largest ranks are dropped. *Calendar Queue Overflow* can be mitigated by increasing the rank range that the scheduler covers, however, if overflows are unavoidable, packets with the largest ranks are the ones to be dropped in most cases.

We further evaluate the impact of overflows and how the above solutions mitigate them in *Appendix* D.

C Average Number of Extra Memory Accesses

According to *Section* 3, the key idea for Sifter to eliminate packet inversions is to apply "Sift Sorting", which moves packet descriptors between the RCQ and the Mini-PIFO. While Sifter ensures accurate packet scheduling, the tradeoff is the extra memory accesses introduced by the sifting process. In our evaluation, we quantify the number of extra memory accesses that Sifter introduces to guarantee inversion-free packet scheduling.

As defined in *Section* 3.4, the *average extra number of*

[26]The smaller rank has higher priority

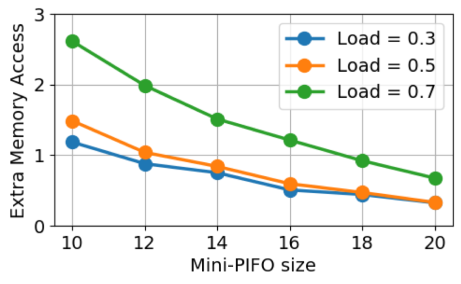


Figure 16: Average number of extra memory accesses per-packet.

memory accesses refers to the additional number of accesses to a packet descriptor between the times it is enqueued and dequeued from the packet scheduler when compared with a simple FIFO queue. For all FIFO-based packet schedulers, the number of memory accesses for each packet descriptor is equivalent to that of a simple FIFO queue, which is one access when writing the packet descriptor into the FIFO and another access when reading it out.

We conducted multiple tests with our single-node micro-benchmark in *Section* 5.1 to evaluate the *average number of extra memory accesses* with different Mini-PIFO sizes S_P. We used a configuration with 25 FIFOs with a depth of 40 and a Mini-PIFO size S_P between 10 and 20 in the evaluation.

Figure 16 shows the *average number of extra memory accesses* per packet scheduled by Sifter with different Mini-PIFO sizes S_P under different traffic loads[27]. From the evaluation results in Figure 16, Sifter has an *average number of extra memory accesses* of less than 2 in most cases, which, in total, is less than twice the baseline (the number of memory accesses for FIFO-based schedulers). This indicates under most circumstances that the packet descriptors in Sifter would at most experience one Sifting Process which introduces one extra read and one extra write operation before the dequeue operation. As the Mini-PIFO size S_P increases, Sifter can achieve nearly zero extra memory accesses since more packets would enqueue and dequeue directly from the Mini-PIFO, bypassing the RCQ FIFOs and the Sifting process.

D Evaluation of Design Parameters

As described in *Section* 3 and *Section* 4, there are multiple design parameters in Sifter including the number of FIFOs F, the FIFO size S_F, the PIFO size S_P, as well as the granularity g. In this section, we explore the performance and FPGA resource consumption as a function of the design parameters.

[27]We applied empirical traffic based on web-search flow distributions [3]. The configured traffic loads control the arrival rate of flows.

D.1 FCT Performance

We set up a series of evaluations on the single-node micro-benchmark[28] to test the performance with different design parameters.

FIFO size As stated in *Appendix* B, the *FIFO overflow* occurrence largely depends on the FIFO size S_F. Here we evaluate the impact of the FIFO size S_F on FCT. We set up four Sifter schedulers with 32 FIFO queues with different FIFO sizes S_F ranging from 32 to 256 locations. The Mini-PIFO size S_P of all four Sifter schedulers is set to 32. In order to meet the inversion-free condition (3) and (6), we set the speed-up factor K of the four Sifter schedulers to range between 2 and 16.

Figure 17(a) shows that when the FIFO size S_F is large enough to hold the incoming packets associated with each rank range, *FIFO overflows* are essentially eliminated and Sifter delivers an identical performance of an ideal PIFO.

PIFO size To investigate how the Mini-PIFO size S_P impacts performance, we start with a design parameter configuration to induce overflows with a specific traffic profile. We set up four Sifter schedulers with 32 FIFO queues, each with a FIFO size $S_F = 64$. We configured different Mini-PIFO sizes S_P as 32, 64, 128, and 256 to investigate whether a larger PIFO would reduce overflow events.

Figure 17(b) shows the FCT performance of Sifter schedulers with increasing Mini-PIFO sizes. From the results, a larger Mini-PIFO provides better FCT performance for the smaller flows (whose packets tend to have smaller ranks). In addition, according to *section* 3.4 and *Appendix* C, a larger Mini-PIFO decreases the frequency of Sifting operation, thereby reducing the number of extra memory accesses.

Number of FIFOs According to *Appendix* B, *FIFO overflow* occurs when the packets associated with a FIFO exceed the FIFO capacity. In addition to increasing the FIFO size S_F, another solution to reduce *FIFO overflows* is to decrease the rank range associated with each FIFO. To cover the same overall rank range when reducing the rank range associated with each FIFO, we increase the total number F of the FIFO queues.

We set the FIFO size $S_F = 64$ to observe the performance improvements due only to increasing the total number of FIFOs. All the Sifter schedulers cover the same total rank range of 512. When increasing the number F of FIFOs from 8 to 64, the granularity g (the rank range that each FIFO covers) of each FIFO is decreased from 64 to 8.

Figure 17(c) shows the FCT performance of Sifter schedulers with increasing the number of FIFO. Similarly to increasing the FIFO size S_F in the previous evaluations, in-

[28]Details of the micro-benchmark topology are given in *Section* 5.1.

creasing the number of FIFOs F also decreases the occurrence of *FIFO overflows*. When the number of FIFOs F is large enough to eliminate FIFO overflows, Sifter provides performance that matches that of an ideal PIFO.

D.2 Resource Overhead

In this section, we assess the logic resource utilization of the Sifter scheduler with different design parameters.

We synthesized the Sifter VHDL implementation with different parameter setups using AMD/Xilinx's Vivado software [5] targeting an AMD/Xilinx Alveo U250 FPGA board [4], which we used to implement the Sifter hardware prototype described in *Section* 5.3. Figure 18 shows the increase in resource consumption as we increase the FIFO size S_F, Mini-PIFO size S_P as well as the total number of FIFOs F. The evaluated resources on the FPGA device are *Look Up Tables* (LUT), *Flip Flops* (FF), and *Look Up Table RAM* (LUTRAM).

FIFO size To observe how the resource utilization grows as a function of the FIFO size S_F, we configured Sifter with 32 FIFOs and a Mini-PIFO of size 32 and increased the FIFO size S_F from 32 to 256.

Figure 18(a) shows the growth of the resource utilization as we increased the FIFO size S_F. The results indicate that the extra resources due to increasing the FIFO size S_F are not significant. From Figure 18(a), the resource consumption is around 1.5% when we apply a fairly large FIFO size $S_F = 256$. Combined with the performance evaluations in *Appendix* D.1, increasing the FIFO size S_F of Sifter is the most efficient way to reduce the occurrence of *FIFO overflows* and to match the performance of an ideal PIFO.

PIFO size According to *Section* 3.5, Sifter needs to fulfill condition (3) and (6) to guarantee inversion-free operation. To meet the above conditions, Sifter needs a larger Mini-PIFO to support larger FIFO sizes.

We increased the Mini-PIFO size S_P from 32 to 256 using the same parameter setup as in the previous evaluation, where we fixed the total number of FIFOs F to 32 and set the size of FIFOs S_F as 256.

Figure 18(b) shows how the resource utilization grows as the Mini-PIFO size S_P increases. From the results, we find that increasing the Mini-PIFO size S_P consumes more resources on the FPGA when compared with increasing the FIFO size S_F.

Number of FIFOs As stated in *Appendix* B and D, another way to reduce *FIFO overflows* is to increase the number of FIFOs F.

To evaluate the effect of increasing the number of FIFOs on resource utilization, we fixed the size of FIFOs S_F and the size of the Mini-PIFO S_P to 32 locations, while increasing the number of FIFOs F from 8 to 128. Figure 18(c) shows the growth of the resource utilization as we increase the number of FIFOs F. FIFOs consume fewer resources when compared with the Mini-PIFO. According to the results in Figure 18(c), a configuration with 64 FIFOs only consumes 2% of the resources on the FPGA while one with 128 FIFOs costs around 3.3% of the available resources.

Combining the results of resource consumption in Figure 18(a) and 18(c), both solutions of increasing FIFO size S_F and increasing the number of FIFOs F are efficient approaches to reduce *FIFO overflows*.

E Sifter Hardware Prototype Details

Packet Descriptor We used a 72-bit packet descriptor for the Sifter hardware prototype as shown in Figure 19. There are 5 fields in the descriptor: (1) a 15-bit *Packet Pointer* for the address of the packet in packet buffer, (2) an 11-bit *Packet Length* for the packet length in bytes, (3) a 20-bit *Packet Rank* for the assigned scheduling order of each packet, (4) a 10-bit *Flow ID* to identify the flow and (5) a 16-bit *Packet ID* for packet identification[29].

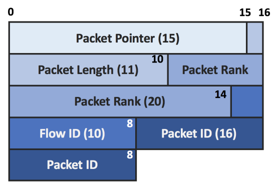
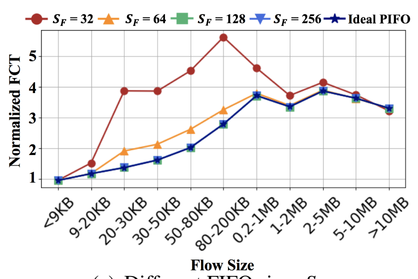


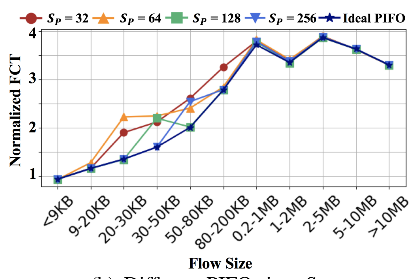
Figure 19: Packet Descriptor Structure

Packet Rank and FIFO indexing According to *Section* 4, we designed a fast RCQ FIFO indexing operation to find the index of the FIFO associated with the packet rank by bit slice selection and shifting. By configuring the number of FIFOs F and the granularity g as powers of two, the associated FIFO index is always equivalent to a grouping of bits in the packet rank.

In the configuration shown in Figure 20, the scheduler has 32 FIFOs in the RCQ, and the granularity g of each FIFO is 32 (each FIFO covers a rank range of 32). We subdivide the 20-bit packet rank into three sections: 10 *Rotation bits*, 5 *Index bits* and 5 *Lower bits*. Since the granularity $g = 32$, the RCQ does not discriminate between descriptors that have equal (15) upper bits but different (5) *Lower bits*. Since the RCQ has 32 FIFOs in total, the FIFO index contains 5 bits. Sifter uses the *Index bits* (9^{th} down to 5^{th} bits, inclusive) in the packet rank to find the FIFO associated with the rank value. The *Rotation bits* in the packet rank only indicates

[29]The Packed ID may not be necessary in many applications but we added it to detect and identify missing packets


(a) Different FIFO sizes S_F


(b) Different PIFO sizes S_P

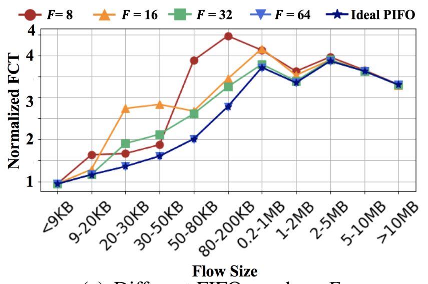
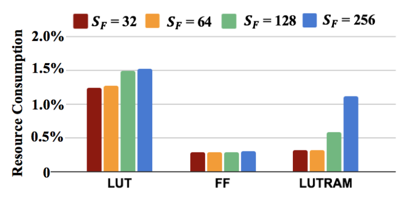
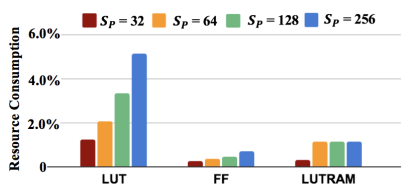

(c) Different FIFO numbers F

Figure 17: Normalized FCT with Different Design Parameters


(a) Different FIFO sizes S_F


(b) Different PIFO sizes S_P

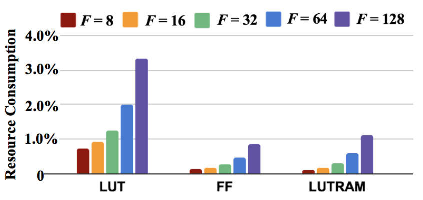

(c) Different FIFO numbers F

Figure 18: Resource Overhead with Different Design Parameters

the number of times the RCQ has rotated and does not affect FIFO indexing. In the following example in Figure 20, Sifter needs to find the FIFO index associated with packet rank 996. After we convert 996 into binary and select the *Index bits*, the value is 11111 and matches index 31 of the associated FIFO. This approach was used to implement fast RCQ FIFO indexing to avoid using time-consuming math operations.

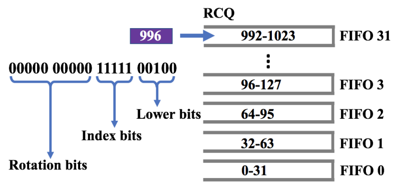


Figure 20: Fast RCQ FIFO Indexing

Speed-up Factor in Sifter Prototype In *Section* 3.5, we provided the conditions that Sifter must meet to guarantee inversion-free packet scheduling. Specifically, for a given FIFO size S_F and Sifting threshold Th_R, Sifter must provide a speed-up factor K that satisfies condition (3). In our Sifter prototype implementation, this speed-up factor K needs to be greater than 2.

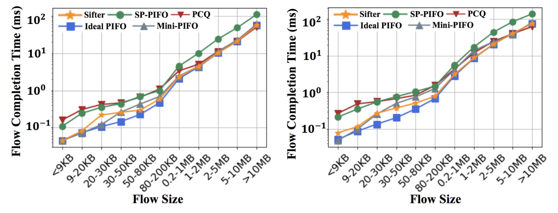
Our Sifter FPGA prototype implementation requires a minimum of 5 clock cycles for each Enqueue and Dequeue operation, as well as the operation to simultaneously "sift" a packet descriptor between the RCQ and the Mini-PIFO. However, to ensure inversion-free packet scheduling, we need to provide a speed-up factor $K = 2$, which means that the operation to sift a packet descriptor needs to be twice as fast as the dequeue operation. Therefore, our hardware prototype can sustain a dequeue interval of 10 clock cycles to achieve the necessary speed-up factor.

F Micro-benchmark Evaluation: FCT and end-to-end delay

We analyzed multiple flows using the micro-benchmark to evaluate the FCT and end-to-end delay performance of different packet schedulers. We extended the single-node star topology with 129 end-hosts to create a 128-to-1 in-cast pattern. We generated empirical traffic according to web-search flow size distributions in data centers [3]. The arrival pattern of the TCP flows follows a Poisson distribution, where the average arrival rate is determined by the configured traffic load. Upon the arrival of each TCP flow, the simulator randomly selects a sender host and starts a TCP connection[30] to the receiver host.

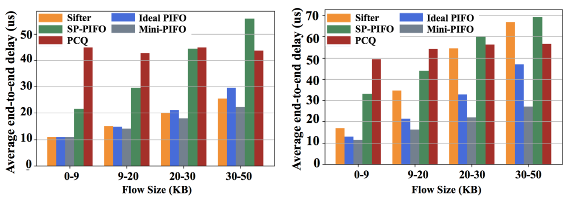
Sifter has the closest FCT to the ideal PIFO Figure 21 shows the FCT of Sifter, PCQ, SP-PIFO, the ideal PIFO, and the stand-alone Mini-PIFO in this single-node star topology. Overall, Sifter has the FCT performance that is closest to the ideal PIFO. For the small-sized flows, Sifter outperforms PCQ and SP-PIFO by eliminating packet inversions. When compared with the stand-alone Mini-PIFO, Sifter has a much

[30]For simplicity, all packets in our packet-based simulation have the same size of 1500 bytes and each TCP flow shares the same weight in SFQ.



(a) Single-node FCT, 70% load (b) Single-node FCT, 90% load

Figure 21: FCT with single-node topology



(a) Average delay, 70% load (b) Average delay, 90% load

Figure 22: Average end-to-end delay with the single-node topology

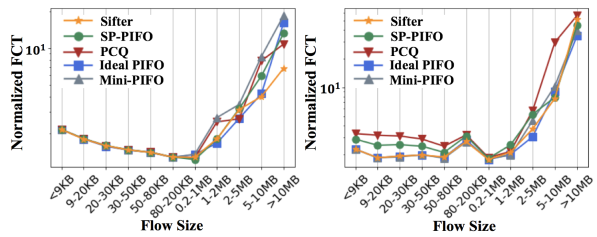
better capacity to reduce packet losses, which leads to fewer re-transmissions and lower FCT for the mid-sized flows.

Sifter provides lower delay for short flows We examine the effect of scheduling inversions on short-flow delays because they are more susceptible to those inversions.

Figure 22 shows the end-to-end delay for packets of short flows. When compared with PCQ and SP-PIFO, Sifter has a lower end-to-end average delay in most cases. As discussed in *Section* 2.2.1, packet scheduling inversions have a larger impact on the delays of short-flow packets. When packets with larger timestamps depart before packets with smaller timestamps from the short flows, packets from the short flows experience longer delays.

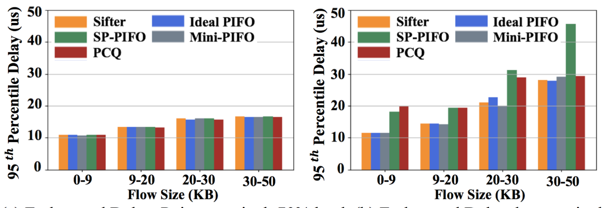
G Use Case: Modified Shortest Remaining Processing Time to minimize FCT

Shortest Remaining Processing Time (SRPT) [43] is an effective scheduling algorithm for minimizing average FCT. However, SRPT may lead to starvation and packet misordering within each flow. There are two mainstream solutions for these issues: per-flow queues and pFabric's starvation prevention [3]. However, these solutions have costly implementations. The per-flow queue solution requires the scheduler to support a number of queues equal to the number of active flows ($> 10K$ flows [49]), which is impractical for current switches. The second solution, starvation prevention by bit-wise comparison among packets from the same flow, requires complex hardware implementation and is power-hungry.



(a) Normalized FCT, Poisson arrival, 70% load (b) Normalized FCT, brust arrival

Figure 23: Normalized FCT with Modified SRPT



(a) End-to-end Delay, Poisson arrival, 70% load (b) End-to-end Delay, brust arrival

Figure 24: End-to-End Delay with Modified SRPT

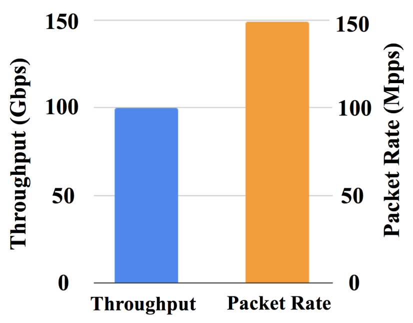
pFabric also proposed a modification to the SRPT scheduling algorithm to make it more practical with negligible impact on FCT performance. Instead of setting the rank of each packet using the remaining flow size, this modified SRPT algorithm simply sets the rank of all the packets from the same flow as the total flow size. Although the rank does not reflect the remaining size of each flow, the packet out-of-order and starvation issues are eliminated.

In this part of the simulation, we applied the modified SRPT algorithm for Sifter, PCQ, and SP-PIFO. Each of the schedulers has 16 FIFOs with a depth of 64 and Sifter has a Mini-PIFO with a capacity of 32. We also set up an ideal PIFO with a capacity of 1024 and a standalone Mini-PIFO with a capacity of 32, the same as Sifter.

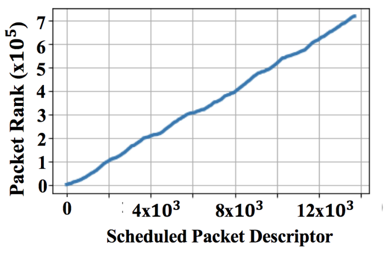
We applied two types of traffic in the evaluation: Poisson arrival traffic and bursty arrival traffic. We generated the Poisson arrival traffic following the procedure in *Appendix* F. The flow size follows empirical distributions of Web-search traffic [3] and the arrival pattern of the TCP flows follows a Poisson distribution. In addition, we added short bursts of TCP flows with Poisson arrivals to form bursty arrival traffic, aiming to create congestion on the bottleneck link.

Sifter achieves FCT and delay performance matching an ideal PIFO In the use case of the modified SRPT algorithm, each packet's rank is equivalent to the flow size it belongs to. As a result, the range of packet ranks is not larger than that of active flow sizes, which makes packet scheduling much easier. Therefore, all packet schedulers can achieve near-optimal FCT and delay performance, as shown in Figure 23 (a) and Figure 24 (a).

To increase the range of packet ranks, we introduced short bursts in flow arrivals. Figure 23 (b) and Figure 24 (b) show that with a wider range of packet ranks and heavier packet



(a) Throughput and Packet Rate



(b) Scheduling Order

Figure 25: ASIC Simulation

congestion, the performance of PCQ and SP-PIFO degrades while Sifter still closely approximates the benchmark of an ideal PIFO.

H Simulated ASIC Performance

As discussed in *Section* 5.3, the FPGA prototype sustains a 100 Gbps line rate for packets larger than 370 bytes, due to the limitation of the clock frequency of 322 MHz achievable in the target FPGA. The 370-byte value is derived as follows: The 322 MHz clock has a period of 3.1 ns. As explained in *Appendix* E it takes 10 clocks to achieve a K factor of 2. Sifter's FPGA prototype can dequeue a packet every 10 x 3.1 = 31 ns, which is equivalent to 3100 bits (387 bytes) at 100 Gbps. The 387-byte length includes 20 bytes of Ethernet overhead (Preamble, SFD, Interframe Gap), which makes the actual packet 367 bytes. We rounded up this value to 370 bytes so that we can easily control the dequeue rate in hardware by counting down 37 bytes per clock (10 x 37 = 370). To demonstrate that Sifter can achieve 100 Gbps with a faster clock rate (e.g., in an ASIC), we ran a simulation using our Python-based Cocotb test environment [21], where we set the clock frequency to 1.7 GHz. As shown in Figure 25 (a), Sifter can sustain a 100 Gbps line rate with the minimum size packets of 64 bytes. Figure 25 (b) shows that Sifter's VHDL implementation performs inversion-free scheduling for 64-byte packets when running at 1.7 GHz.

I Pseudo-code of Sifter

We summarize the *Enqueue*, *Dequeue*, and *Sifting* processes with pseudo-code in **Algorithm** 1.

Algorithm 1 Enqueue, Dequeue and Sifting process of Sifter

```
1: function INITIALIZATION
2:     Initialize Sentinel $s = \infty$
1: function ENQUEUE PACKET($P_{(i,k)}$)
2:     if $R_{(i,k)} \leq s$ then
3:         Mini-PIFO enqueue $P_{(i,k)}$
4:         if $P_{(i',k')}$ evicted from Mini-PIFO then
5:             RCQ enqueue $P_{(i',k')}$
6:     else
7:         RCQ enqueue $P_{(i,k)}$
1: function DEQUEUE PACKET( )
2:     Dequeue packet $P_{(i,k)}$ from Mini-PIFO
3:     if $O_P \leq Th_S$ then
4:         SIFTING( )
1: function SIFTING( )
2:     Find earliest non-empty FIFO $f$ in RCQ
3:     Update $s$ as the largest rank covered by FIFO $f$
4:     for Packet $P_{(i,k)}$ in $f$ do
5:         if $R_{(i,k)} \leq s$ then
6:             Mini-PIFO enqueue $P_{(i,k)}$
7:             if $P_{(i',k')}$ evicted from Mini-PIFO then
8:                 RCQ enqueue $P_{(i',k')}$
9:                 $s = MAX\{R_{(i',k')}, s\}$
10:        else
11:            RCQ enqueue $P_{(i,k)}$
12:    if RCQ is empty then
13:        Reset Sentinel $s = \infty$
14:    else if $O_P \leq Th_S$ then
15:        SIFTING( )
```

For variable names and definitions, please refer to Table 1.

Acknowledgements

We acknowledge AMD/Xilinx for providing the Alveo U250 board and Exegy for providing the nxFramework, both of which enabled the FPGA prototyping of Sifter. We extend our sincere gratitude to our shepherd, Brent Stephens, and the anonymous NSDI reviewers for their valuable feedback.