Quiver: Dynamic Quantile-Assisted Scheduling to Reduce Rank Inversion

Kunvar Kanhaiya†, Sourabh Singh†, Pralhad Magadum, Rituraj Patel and Aniruddha Singh Kushwaha
Department of Computer Science & Engineering Indian Institute of Technology Indore, India
(phd2201101015, phd2201201007, msrphd2304101012, phd2401101006, aniruddha)@iiti.ac.in

***Abstract*—Programmable packet scheduling is essential for supporting diverse Quality-of-Service demands in modern networks. The Push-In First-Out (PIFO) queue offers an abstraction for programmable packet scheduling. However, achieving a line-rate and scalable implementation remains an open challenge. Consequently, prior research has explored approximating PIFO semantics using commodity Strict Priority (SP) queues. Existing approaches predominantly employ static or heuristic-based rank-to-queue mappings, which often lead to scheduling inaccuracies manifested as rank inversions.**

In this paper, we present Quiver, a dynamic quantile-assisted scheduler that continuously adapts to traffic by sampling packet ranks and computing statistical bounds for strict priority queues, minimising rank inversions at line rate.

We evaluate Quiver using the NetBench simulator across diverse traffic distributions. Results show that its quantile-based design closely matches the performance of the optimal static Gradient-based algorithm while significantly reducing rank inversions. Furthermore, Quiver drops fewer packets of high-priority flows compared to existing designs and achieves Flow Completion Times (FCTs) comparable to an ideal PIFO.

***Index Terms*—Quantile scheduling, PIFO approximation, programmable packet scheduling, Flow completion time (FCT), Dynamic queue adaptation, Rank inversion.**

I. Introduction

The rapid growth of diverse network services—from massive-scale data centre transfers to ultra-low-latency real-time applications—places stringent Quality of Service (QoS) demands on modern network infrastructures [5], [12]. Network programmability is a key enabler for addressing application-specific requirements. Therefore, to introduce programmability, modern networks have increasingly adopted Software-Defined Networking (SDN) [14], which decouples the control plane from the data forwarding plane, enabling centralised control and fine-grained programmability. This architectural evolution is driving a shift away from expensive, proprietary, fixed-function hardware toward open, commodity programmable switches, typically programmed using high-level languages such as P4 [6], [8]. While early deployments of programmable hardware focused primarily on data centre environments, recent research and deployment trends highlight its growing importance in broader contexts, including 5G wireless-wireline convergence [8]. By enabling flexible traffic management and scheduling policies, this shift toward programmable networks provides the agility needed to meet diverse QoS objectives efficiently.

Packet scheduling is a crucial network function that ensures QoS requirements are met by controlling the exact transmission order of packets within the network data plane. The choice of scheduling algorithm directly dictates key performance metrics such as Flow Completion Time (FCT), tail latency, fairness, and overall throughput [5], [7]. However, traditional hardware switches support only a limited set of fixed scheduling algorithms, such as Strict Priority (SP) [2], [5], Deficit Round Robin (DRR) [17] etc. This hardware rigidity severely restricts the network's ability to adapt to dynamic traffic distributions and evolving application demands [19].

Over the recent years, in highly programmable data planes, enabling the definition and deployment of custom, expressive packet scheduling algorithms has gained interest due to its critical significance in meeting QoS requirements. The Push-In First-Out (PIFO) queue [19] introduced the key abstraction model that enables the scheduler's programmability. PIFO abstracts scheduling into a two-step process: assigning a scalar rank to a packet upon arrival, and inserting the packet into a sorted priority queue based on that rank. While PIFO enables the deployment of a broad spectrum of scheduling strategies [2], [10], [13], [15], [16], [18], [20], realizing a perfect PIFO queue at line rates and large flow scales imposes formidable hardware constraints. Maintaining a perfectly sorted data structure at line rate demands complex hardware logic, fundamentally limiting the capacity of exact-PIFO implementations to merely a few thousand flows on modern switch ASICs [9].

To overcome these hardware implementation challenges, recent research has focused on approximating PIFO behaviour using the limited hardware primitives natively available on commodity switches. This is typically achieved using either multiple Strict Priority (SP) queues [2], [3] or a single FIFO queue [20]. A critical challenge in SP-based approximation is accurately mapping a large, continuous space of packet ranks onto a small number of discrete hardware strict priority queues. Inaccurate or static mapping strategies inevitably cause scheduling errors known as *rank inversions* – lower-priority (higher-rank) packets are assigned to a higher-priority queue and transmitted before higher-priority (lower-rank) packets [2], [3]. Such inversions severely violate the

†Both authors contributed equally to this work.

ISBN 978-3-903176-82-9

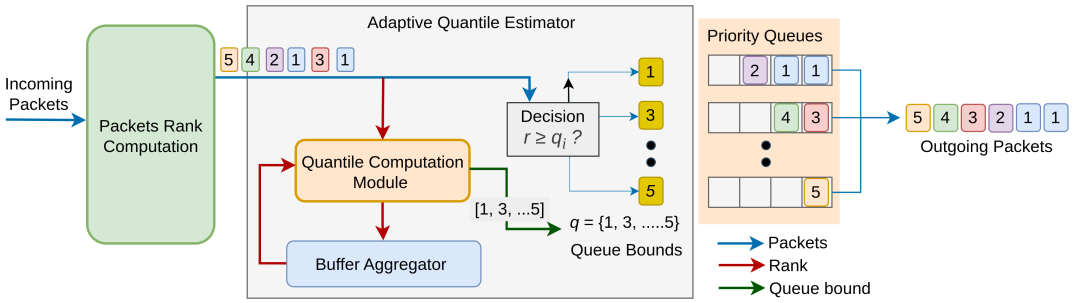


Fig. 1: Proposed Quiver architecture.

intended scheduling policy and degrade the performance of the application.

The SP-PIFO framework [2] identified the challenge of mapping PIFO behaviour to limited hardware and proposed a gradient-based "Greedy" algorithm to compute optimal static queue boundaries. While effective offline, it cannot operate in real time, and its hardware heuristic struggles to maintain optimal performance under bursty, dynamic traffic. Consequently, a critical research question remains:

How can we dynamically and deterministically map arbitrary packet-rank distributions to SP queues at runtime to minimise rank inversions and achieve near-optimal performance?

In this paper, we introduce Quiver, a programmable packet scheduling architecture that accurately approximates PIFO behaviour using standard SP queues without relying on reactive heuristics or offline computations. Quiver achieves this through an Adaptive Quantile Estimator (AQE) that continuously samples incoming packet ranks into an internal buffer. This buffer forms an empirical representation of the short-term network traffic. By dividing this sampled distribution into N segments, Quiver periodically computes deterministic, quantile-based queue boundaries that evenly partition the observed rank space across the N available SP queues. By dynamically and proactively updating these bounds, Quiver effectively lowers the rank inversions and closely matches the theoretical upper-limit performance of optimal static partitioning. The contributions of this paper are as follows:

- The Quiver scheduler design, detailing its rank sampling, quantile-based queue bounds computation, and dynamic update mechanisms.
- A comprehensive implementation and evaluation of Quiver using the Netbench simulator, demonstrating its effectiveness in approximating ideal PIFO performance through an extensive simulation study.

The remainder of the paper is organized as follows. Section II details the Quiver architecture, and in Sections III and VI, we present the micro-architectural analysis and macro-level evaluation. Lastly, Section VIII concludes the work.

II. System Overview

In this section, we describe the design and operational mechanism of our proposed Quiver packet scheduler.

A. System Design

The Quiver design consists of: 1) a programmable *Rank Computation Module*, 2) a fixed logic *Adaptive Quantile Estimator*, and 3) a bank of *Priority Queues*, as shown in Fig. 1.

1) Rank Computation Module: The Rank Computation Module operates on a per-packet basis, invoking a programmable scheduling function, similar to the PIFO's transaction mechanism [19], to ascertain a packet's rank. This rank is calculated using predefined scheduling algorithms and packet ranking strategies such as Start-Time Fair Queuing (STFQ) [11], Shortest Remaining Processing Time (SRPT) [5], Deficit Round Robin (DRR) [17]. The computed rank fundamentally determines the packet's scheduling priority and, consequently, its placement within the queuing architecture.

2) Adaptive Quantile Estimator (AQE): The AQE comprises two integrated components. First, the Quantile Computation and Bound Update (QCBU) module utilises a sampling buffer of capacity k. This buffer temporarily stores incoming packet ranks in non-decreasing order to construct a sample population, which estimates the statistical distribution of the traffic. Although this buffer stores ranks in sorted order, it does not pose significant implementation challenges because the value of k is fixed and relatively small. The module leverages this estimation to dynamically calculate queue boundaries. These boundaries subsequently serve as strict rank thresholds (bounds) that govern packet admission into the respective priority queues. Second, a Buffer Aggregator synthesises the most recent rank distribution data, updating the primary sampling buffer with the newly aggregated values. The operational details of the AQE are elaborated in Section II-B.

3) Priority Queues: The Quiver system employs a set of parallel priority queues, each corresponding to a specific rank interval defined by the bounds produced by the Quantile Computation Module. Upon arrival, a packet's computed rank is compared against these bounds to determine the appropriate queue. Packets are then enqueued accordingly

and later dequeued for transmission under a strict-priority scheduling discipline, which services higher-priority queues first.

B. System Working

This section details the operational mechanics of the Quiver scheduler. The workflow of Quiver is partitioned into three primary phases: (1) packet enqueue, (2) queue bound computation, and (3) buffer aggregation. The enqueue process maps incoming packets to the appropriate priority queues using the current rank boundaries. Concurrently, the bound computation mechanism periodically refines these boundaries based on observed traffic patterns, enabling dynamic adaptation of the rank-to-queue mapping. Finally, buffer aggregation compresses the historical rank data, updating the buffer with a statistical summary of the recent traffic distribution.

1) Packet Enqueue: Upon arrival, the Rank Computation Module assigns a rank to each packet based on the configured scheduling algorithm. This rank is simultaneously forwarded to the AQE module for queue selection and stored in its internal sampling buffer for future distribution estimation. It is crucial to emphasise that the sampling buffer stores only the rank values of the packets.

Within the AQE module, a packet's target queue is determined by comparing its computed rank r against the active set of queue bounds $q = \{\lfloor q_0 \rfloor, \lfloor q_1 \rfloor, \ldots, \lfloor q_{N-1} \rfloor\}$, where $\lfloor q_i \rfloor$ denotes the rank threshold for queue i. The comparison proceeds sequentially from the lowest-priority queue (associated with q_{N-1}) toward the highest-priority queue (q_0). The packet is assigned to the highest-index queue i for which the condition $r \geq q_i$ holds. If the rank r is strictly lower than all current queue bounds, the packet is automatically assigned to queue 0, representing the highest priority.

For example, consider the illustration in Fig. 1 with queue bounds $q = \{\lfloor q_0 \rfloor = 1, \lfloor q_1 \rfloor = 3, \lfloor q_2 \rfloor = 5\}$ for $N = 3$ queues, and an arrival sequence of 5 4 2 1 3 1, processed from right to left. A packet with rank $r = 1$ first evaluates queue 2; since $r \geq \lfloor q_2 \rfloor$ is false, it evaluates queue 1, which also fails. The condition finally holds for queue 0, where the packet is successfully enqueued. Subsequently, a packet with rank $r = 3$ fails the condition for queue 2 but satisfies $r \geq \lfloor q_1 \rfloor$, resulting in its placement in queue 1. This deterministic comparison logic applies to all incoming packets.

2) Quantile computation and Bound update: The AQE module forms the adaptive core of the Quiver scheduler, continuously aligning the queue thresholds with the empirical rank distribution of the incoming traffic. This adaptation operates in two distinct stages: (*i*) quantile computation, where new statistical rank thresholds are derived from the sampled population, and (*ii*) bound update, where these thresholds are applied to redefine the operational queue bounds for the subsequent scheduling interval.

Quantile Computation. This process is triggered when the sampling buffer B reaches its capacity k. The AQE maintains this buffer in a strictly sorted order, ensuring that $B[0]$ corresponds to the smallest rank (highest priority) and $B[k-1]$ to the largest rank (lowest priority). Upon reaching capacity, the buffer represents a short-term empirical distribution of the network traffic.

From this ordered array, the module isolates N quantiles corresponding to the N priority queues. For a given queue index $i \in \{0, 1, \ldots, N-1\}$, the exact index position within the buffer is calculated as:

$$j_i = \left\lfloor \frac{k}{N} \cdot i \right\rfloor \tag{1}$$

The rank value located at position j_i in the sorted buffer defines the empirical quantile value:

$$\lfloor q_i \rfloor = B[j_i] \tag{2}$$

The resulting set $\{\lfloor q_0 \rfloor, \lfloor q_1 \rfloor, \ldots, \lfloor q_{N-1} \rfloor\}$ forms a non-decreasing sequence of rank thresholds that accurately partition the observed rank space into N distinct intervals.

Bound Update. Once the new quantiles $\lfloor q_i \rfloor$ are computed, they are immediately adopted as the active queue boundaries. Consequently, each queue i is bound by the rank interval $[\lfloor q_i \rfloor, \lfloor q_{i+1} \rfloor)$, where lower indices represent higher transmission priorities. This update mechanism atomically replaces the previous bounds to ensure consistent, uninterrupted packet classification during the transition.

3) Buffer Aggregation: Simply flushing the sampling buffer after a bound computation cycle would erase all historical traffic context, potentially causing severe bound oscillation under highly bursty traffic conditions. To preserve this context, Quiver compresses the k sorted samples into a concise historical summary.

The sampling buffer B is logically partitioned into N segments corresponding to the rank intervals of the priority queues. For each queue index $i \in \{0, 1, \ldots, N-1\}$, the corresponding segment spans the buffer indices from j_i to $j_{i+1} - 1$ (where $j_N = k$). The AQE computes the arithmetic mean of the rank values within each segment to generate a representative summary state:

$$B[i] = \left\lfloor \frac{1}{j_{i+1} - j_i} \sum_{m=j_i}^{j_{i+1}-1} B[m] \right\rfloor \tag{3}$$

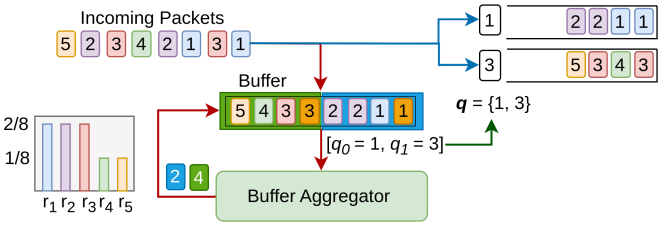
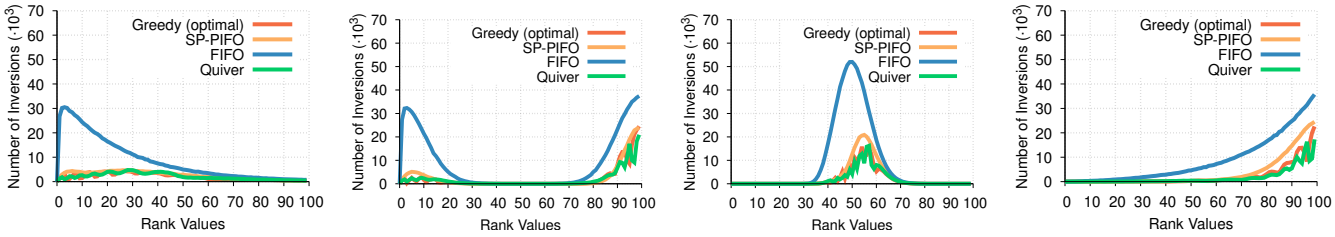


Fig. 2: Bound update mechanism ($N = 2, k = 8$)



(a) Exponential (b) Convex (c) Poisson (d) Inv. exponent

Fig. 3: Alternative distributions: Inversion

These N aggregated values are retained at the front of the buffer ($B[0 \dots N-1]$), while the remaining $k-N$ slots are cleared to accept the next cycle of incoming packet ranks. This aggregation guarantees that each new estimation cycle is initialized with the statistical weight of the previous interval, effectively dampening volatile bound fluctuations.

Fig. 2 illustrates this complete update cycle for an architecture with $N = 2$ priority queues and a sampling buffer capacity of $k = 8$. Assuming the buffer receives the rank sequence $\{5, 2, 3, 4, 2, 1, 3, 1\}$, the sorted state of the sampling buffer becomes:

$$B = \{1, 1, 2, 2, 3, 3, 4, 5\} \tag{4}$$

The quantile positions are calculated as $j_0 = \lfloor \frac{8}{2} \cdot 0 \rfloor = 0$ and $j_1 = \lfloor \frac{8}{2} \cdot 1 \rfloor = 4$, yielding the new boundaries $\lfloor q_0 \rfloor = B[0] = 1$ and $\lfloor q_1 \rfloor = B[4] = 3$. These updated bounds dynamically bisect the rank space.

During the subsequent buffer aggregation phase, the buffer is divided into two segments: $B[0 \dots 3]$ and $B[4 \dots 7]$. The arithmetic mean of the first segment $\{1, 1, 2, 2\}$ evaluates to $\lfloor 6/4 \rfloor = 2$. The second segment $\{3, 3, 4, 5\}$ averages to $\lfloor 15/4 \rfloor = 4$. The buffer is then compacted to contain only the historical summary 4 2, leaving 6 empty slots for the next sampling cycle.

III. System Analysis

This section presents a switch-level simulation study of Quiver, comparing its performance against the ideal PIFO, FIFO, SP-PIFO, and the optimal Greedy algorithm to characterize its operational effectiveness.

A. Experimental Setup

The analysis employs *NetBench* [1], a packet-level simulator. The setup consists of a single switch with 1500 flows, each transmitting 1 MB of data over one second. The data is sent through a 10 Gbps link operating at 75% utilization. The systems with strict-priority schedulers (Quiver, SPPIFO) use 8 queues with 10-packet capacity each, while the FIFO baseline is allocated an 80-packet buffer for fairness.

To determine the optimal architectural configuration, we first evaluate Quiver's sampling buffer capacity under a uniform rank distribution across varying sizes ($k \in \{16, 32, 64, 128, 256\}$). Following this, the system's performance is evaluated across Poisson, Convex, Exponential, and Inverse Exponential packet rank distributions. The primary metric is the number of *rank inversions*, recorded whenever a dequeued packet has a higher rank than any packet remaining in the queues.

B. Analysis of rank inversions

Fig. 3 presents the number of rank inversions observed under each rank distribution. In all cases, Quiver closely follows the Greedy algorithm, showing its ability to approximate ideal rank-to-queue mapping without iterative optimization. SP-PIFO performs well but shows slightly higher inversions, especially under non-uniform traffic. FIFO, lacking rank awareness, produces the most inversions and serves as a lower performance bound.

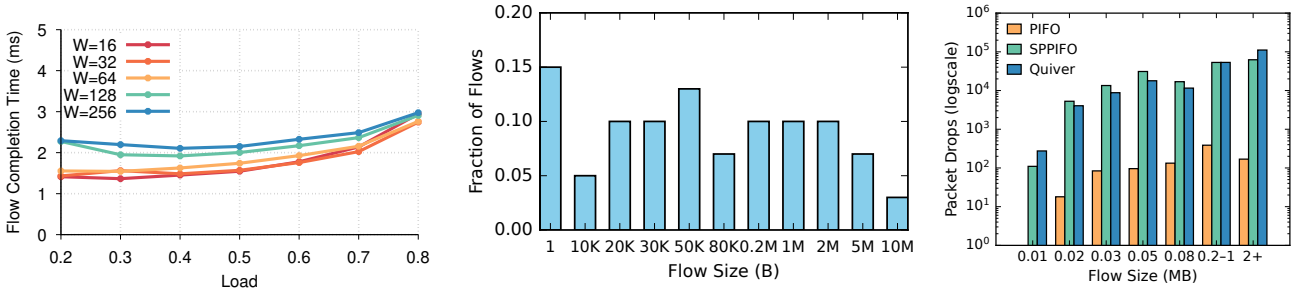
Quiver maintains a consistently low inversion rate across all traffic patterns. Its quantile-driven adaptation accurately captures rank variations and adjusts queue bounds in real time. By using statistical sampling instead of heuristic tuning, Quiver preserves stable queue partitioning even under dynamic traffic conditions. These results confirm that Quiver effectively infers the underlying rank distribution and updates queue thresholds dynamically, achieving near-optimal scheduling performance comparable to the Greedy algorithm (the theoretical upper limit).

IV. Evaluation

This section presents the packet-level simulation conducted to assess the performance of Quiver. The obtained results are compared against existing state-of-the-art approaches to highlight its end-to-end effectiveness.

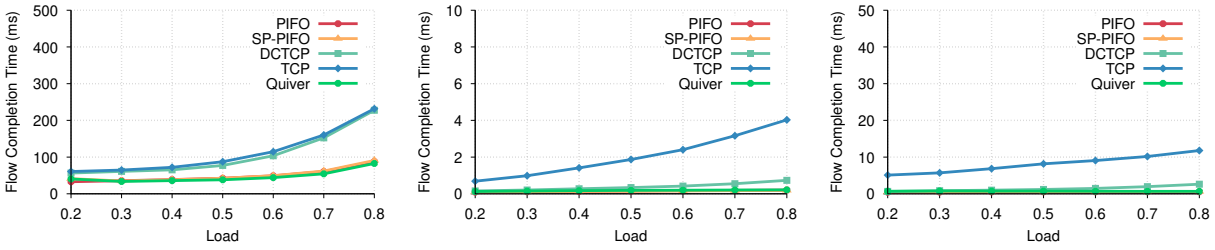
A. Methodology

We simulated a large-scale data center network adopting a leaf–spine topology using NetBench, a packet-level simulator [1]. The topology comprises 4 spine switches, 9 leaf switches, and 144 servers, with each leaf switch connected to 16 servers. The interconnections between leaf and spine switches operate at 40 Gbps, while the server access links are configured at 10 Gbps. The simulation uses the pFabric web application and data mining workload [5] with Poisson-distributed flow arrivals to measure Flow Completion Time (FCT). For SP-PIFO and Quiver, we configured 8 priority



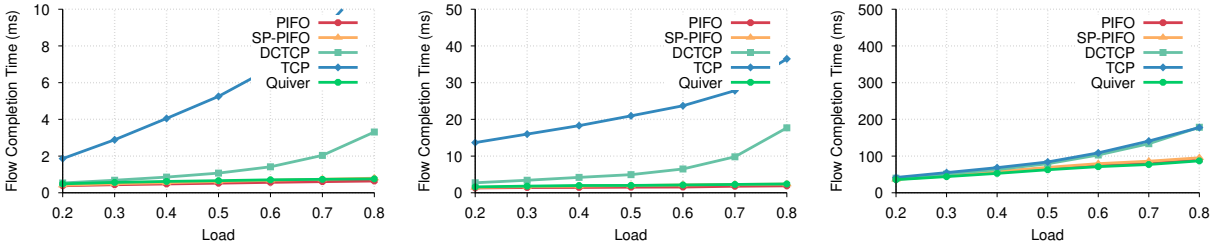
(a) (b) (c)

Fig. 4: (a) Impact of buffer size under varying network loads, (b) Flow size distribution, and (c) Packet drop.



(a) [1MB, ∞)]: Avg (b) [0, 100KB]: Avg (c) Less than 100KB 99th FCT

Fig. 5: pFabric: Data Mining



(a) [0, 100KB]: Avg (b) Less than 100KB 99th FCT (c) [1MB, ∞)]: Avg

Fig. 6: pFabric: Web Search Workload.

queues, each with a 10-packet capacity, whereas for PIFO, TCP, and DCTCP, a single FIFO drop-tail queue of 80-packet capacity was used.

B. Impact of Buffer Size on FCT

We evaluate the influence of Quiver's sampling buffer capacity ($k \in \{16, 32, 64, 128, 256\}$) on the average normalized FCT using the data-mining workload. As shown in Fig. 4a, average FCT predictably scales with network load due to base congestion. However, the optimal buffer size depends heavily on this load. Under lighter traffic, smaller capacities yield a slight performance advantage; their rapid adaptation to transient traffic shifts outweighs the statistical maturity of larger buffers, which update too slowly when packet arrivals are infrequent.

As network load increases, this performance gap closes. Under sustained congestion, the high density of arriving packets allows larger buffers to form highly stable rank estimates, achieving comparable FCTs. This reveals a fundamental architectural trade-off: constrained buffers optimize for temporal responsiveness, whereas larger buffers prioritize statistical precision.

C. Packet drop

We analyze packet drop behavior under the web search workload with 15,000 flows using pFabric's SRPT scheduling, as shown in Fig. 4c. Both Quiver and SP-PIFO successfully prioritize small flows, protecting them from tail drops. However, Quiver demonstrates a clear structural advantage, incurring significantly fewer drops than SP-PIFO across the 20 KB to 1 MB flow sizes. As illustrated in Fig. 4b, this range constitutes a massive fraction of the total workload. This superior drop protection at the application level correlates directly with Quiver's ability to minimize hardware-level rank inversions, ensuring high-priority packets are serviced promptly before their designated queues overflow.

D. Flow completion time

We used pFabric, which assigns packet priorities according to the Shortest Remaining Processing Time (SRPT) rule, to measure flow completion times. For comparison, we included SP-PIFO, the ideal PIFO, and transport protocols such as TCP and DCTCP [4]. Figures 5 and 6 present the mean and 99^{th} percentile flow completion times (FCTs) for short ($<$100 KB) and large ($\geq$1 MB) flows under the data mining and web search workloads, respectively.

1) Data Mining Workload: Under the data mining workload, Quiver shows near-PIFO performance for both short and large flows. For short flows ($<$100 KB), it closely follows SP-PIFO while completing flows about $11\times$ faster than TCP and $2\times$ faster than DCTCP. Mean and 99^{th} percentile FCTs remain low across all loads, demonstrating effective adaptation to varying traffic. For large flows ($\geq$1 MB), Quiver performs slightly better than SP-PIFO, achieving up to 6% lower mean FCT and staying within 4% of PIFO. Overall, Quiver provides a closer approximation of PIFO.

2) Web Search Workload: Quiver maintains performance close to PIFO across all load levels. For short flows ($<$100 KB), it achieves FCT comparable to SP-PIFO while outperforming TCP and DCTCP with $16\times$ and $4.5\times$ respectively. The mean FCT remains within $15-25\%$ of PIFO and narrows further at higher loads, with Quiver occasionally surpassing SP-PIFO by about 7%. For large flows ($\geq$1 MB), Quiver sustains throughput comparable to PIFO and SP-PIFO, achieving up to $8-10\%$ lower mean FCT and roughly $2\times$ faster completion than TCP and DCTCP. These results demonstrate that Quiver provides near-PIFO flow completion performance.

Conclusion

This paper presented Quiver, a quantile-based programmable packet scheduler that approximates Push-In First-Out (PIFO) behavior using standard Strict-Priority (SP) queues. Quiver dynamically samples incoming packet ranks and employs statistical quantile estimation to compute adaptive queue bounds, enabling near-optimal rank-to-queue mapping. Extensive evaluations demonstrated that Quiver closely approximates the performance of the optimal Greedy algorithm across diverse traffic distributions, significantly reducing rank inversions compared to SP-PIFO. In end-to-end network simulations, Quiver achieved near-PIFO Flow Completion Time (FCT) performance while providing lower packet drops, particularly for latency-sensitive short flows.

References

[1] Netbench, 2026. Last accessed 01 Mar 2026. URL:https://github.com/ndal-eth/netbench.

[2] Albert Gran Alcoz, Alexander Dietmüller, and Laurent Vanbever. SP-PIFO: Approximating Push-In First-Out behaviors using Strict-Priority queues. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 59–76, 2020.

[3] Albert Gran Alcoz, Balázs Vass, Gábor Rétvári, and Laurent Vanbever. Everything matters in programmable packet scheduling. *arXiv preprint arXiv:2308.00797*, 2023.

[4] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 63–74, 2010.

[5] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pFabric: Minimal near-optimal datacenter transport. *ACM SIGCOMM Computer Communication Review*, 43(4):435–446, 2013.

[6] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[7] Fahad R Dogar, Thomas Karagiannis, Hitesh Ballani, and Antony Rowstron. Decentralized task-aware scheduling for data center networks. *ACM SIGCOMM Computer Communication Review*, 44(4):431–442, 2014.

[8] Mostafa Elbediwy, Bill Pontikakis, Alireza Ghaffari, Jean-Pierre David, and Yvon Savaria. DR-PIFO: A dynamic ranking packet scheduler using a push-in-first-out queue. *IEEE Transactions on Network and Service Management*, 21:355–371, 2024.

[9] Peixuan Gao, Anthony Dalleggio, Jiajin Liu, Chen Peng, Yang Xu, and H Jonathan Chao. Sifter: An Inversion-Free and Large-Capacity programmable packet scheduler. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 75–95, 2024.

[10] Peixuan Gao, Anthony Dalleggio, Yang Xu, and H Jonathan Chao. Gearbox: A hierarchical packet scheduler for approximate weighted fair queuing. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 551–565, 2022.

[11] Pawan Goyal, Harrick M Vin, and Haichen Chen. Start-time fair queueing: A scheduling algorithm for integrated services packet switching networks. In *Conference proceedings on Applications, technologies, architectures, and protocols for computer communications*, pages 157–168, 1996.

[12] Aniruddha Kushwaha, Sidharth Sharma, Naveen Bazard, and Ashwin Gumaste. Bitstream: A flexible sdn protocol for service provider networks. In *2018 IEEE International Conference on Communications (ICC)*, pages 1–7. IEEE, 2018.

[13] Pralhad Magadum, Sourabh Singh, and Aniruddha Singh Kushwaha. Ct-pifo: A congruity tree based packet scheduler to improve flow completion time. In *2025 34th International Conference on Computer Communications and Networks (ICCCN)*, pages 1–9. IEEE, 2025.

[14] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM computer communication review*, 38(2):69–74, 2008.

[15] Habib Mostafaei, Maciej Pacut, and Stefan Schmid. RIFO: Pushing the efficiency of programmable packet schedulers. *IEEE Transactions on Networking*, 2025.

[16] Naveen Kr Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. Approximating fair queueing on reconfigurable switches. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 1–16, 2018.

[17] Madhavapeddi Shreedhar and George Varghese. Efficient fair queuing using deficit round-robin. *IEEE/ACM Transactions on networking*, 4(3):375–385, 1996.

[18] Vishal Shrivastav. Fast, scalable, and programmable packet scheduler in hardware. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 367–379. 2019.

[19] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. Programmable packet scheduling at line rate. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 44–57, 2016.

[20] Zhuolong Yu, Chuheng Hu, Jingfeng Wu, Xiao Sun, Vladimir Braverman, Mosharaf Chowdhury, Zhenhua Liu, and Xin Jin. Programmable packet scheduling with a single queue. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 179–193, 2021.