Enabling Virtual Priority in Data Center Congestion Control

Zhaochen Zhang[★], Feiyang Xue[★], Keqiang He[△], Zhimeng Yin[◇], Gianni Antichi[□], Jiaqi Gao[+], Yizhi Wang[★], Rui Ning[★], Haixin Nan[★], Xu Zhang[★], Peirui Cao[★], Xiaoliang Wang[★], Wanchun Dou[★], Guihai Chen[★], Chen Tian[★]
[★]Nanjing University [△]Shanghai Jiao Tong University [◇]City University of Hong Kong
[□]Politecnico Milano & Queen Mary University of London [+]Unaffiliated

Abstract

In data center networks, various types of traffic with strict performance requirements operate simultaneously, necessitating effective isolation and scheduling through priority queues. However, most switches support only around ten priority queues. Virtual priority can address this limitation by emulating multi-priority queues on a single physical queue, but existing solutions often require complex switch-level scheduling and hardware changes. Our key insight is that virtual priority can be achieved by carefully managing bandwidth contention in a physical queue, which is traditionally handled by congestion control (CC) algorithms. Hence, the virtual priority mechanism needs to be tightly coupled with CC. In this paper, we propose PrioPlus, a CC enhancement algorithm that can be integrated with existing congestion control schemes to enable virtual priority transmission. PrioPlus assigns specific delay ranges to different priority levels, ensuring that flows transmit only when the delay is within the assigned range, effectively meeting virtual priority requirements. Compared to Swift CC with physical priority queues, PrioPlus provides strict priority for high-priority flows without impacting performance sensibly. Meanwhile, it benefits low-priority flows from 25% to 41% as its priority-aware design enhances CC's ability to fully utilize available bandwidth once higher-priority traffic completes. As a result, in coflow and model training scenarios, PrioPlus improves job completion times by 21% and 33%, respectively, compared to Swift with physical priority queues.

CCS Concepts: • **Networks** → **Transport protocols**; **Network protocol design**; *Data center networks.*

Keywords: Congestion control algorithm, In-network priority, Data center network

EuroSys '25, March 30–April 3, 2025, Rotterdam, Netherlands

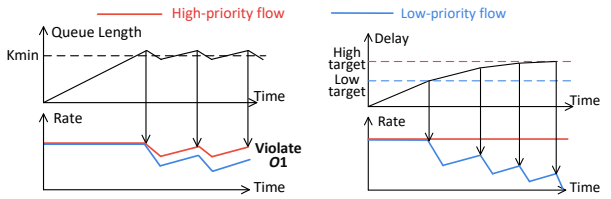
ACM ISBN 979-8-4007-1196-1/25/03
https://doi.org/10.1145/3689031.3717463

ACM Reference Format:
Zhaochen Zhang, Feiyang Xue, Keqiang He, Zhimeng Yin, Gianni Antichi, Jiaqi Gao, Yizhi Wang, Rui Ning, Haixin Nan, Xu Zhang, Peirui Cao, Xiaoliang Wang, Wanchun Dou, Guihai Chen, Chen Tian. 2025. Enabling Virtual Priority in Data Center Congestion Control . In *Twentieth European Conference on Computer Systems (EuroSys '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3689031.3717463

1 Introduction

As cloud computing advances, data centers have been hosting more and more services [35, 36, 81, 99]. These services include latency-sensitive real-time services [22, 36, 40], coflow-dependent computing tasks [11, 30, 57], and throughput-sensitive backend tasks [56, 63, 93], each with its own quality of service (QoS) requirements (e.g., delay and bandwidth). To address the diverse QoS demands of different traffic types, priority queues have become a standard solution for providing weighted bandwidth sharing, effectively isolating traffic from interfering with one another [34, 45, 55, 99]. Moreover, priority queues facilitate priority-based scheduling to enhance application performance by providing strict prioritization within *the same type of traffic* including independent flows [18, 58, 97], coflows [11], RPCs [70], and distributed ML model training [25]. The effectiveness of these priority-based algorithms has been shown to improve with the number of available priority queues [11, 18, 25, 58, 70].

Despite the significance of priority-based isolation and scheduling, they are fundamentally constrained by the limited number of physical priority queues (typically 8 [51] or 12 [24]) due to the following two restrictions. (i) Protocols used in data centers lack support for more priorities. For example, the DSCP field in the IP header, which is used to designate priority, supports only 12 priorities as per the standard [19]. The PFC protocol [49], used for providing lossless priorities for RDMA, supports just 8 priorities. (ii) The consistent trend of bandwidth growth outpacing the increase in switch buffer sizes [44, 71] makes supporting a large number of priority levels increasingly challenging. Microsoft [45] reported that it used only two lossless priorities to provide coarse-grained isolation for real-time and bulk-transfer traffic class on Trident2 switch [1] to conserve buffer exclusively allocated to the PFC headroom. However, the buffer-to-bandwidth ratio in the latest switch [4] is now two times smaller than Trident2, making it challenging to support



(a) D2TCP. **(b)** Expected behavior.

Figure 1. Comparison of D2TCP with expected behavior.

more lossless priorities. Due to the limited number of priority queues, Meta reports having to place traffic from different congestion control (CC) algorithms into the same queue, complicating the deployment of new CCs [34]. This scarcity also hinders the large-scale implementation of priority-based traffic scheduling algorithms [16].

Targeting this fundamental issue, we focus on supporting *multiple* priority levels within *a single* physical queue, a concept we refer to as *virtual priority*. We argue that the limited physical priority queues in the network should be reserved for performance isolation across different traffic classes, while virtual priority can provide strict priorities within each traffic class for scheduling, thereby enhancing overall performance. The virtual priority mechanism aims to emulate the functionality of strict priority queues and shall ideally meet the following three objectives. ***O*1**: Multi-priority assurance – supporting the transmission of flows with varying priorities within a single physical queue, enabling strict and granular prioritization for higher-priority traffic. ***O*2**: Work conservation – fully utilizing available bandwidth without any waste. ***O*3**: Readily deployable – easily implemented in data centers without requiring hardware replacements or upgrades.

Current methods that offer functionality similar to virtual priority rely on packet scheduling algorithms implemented on switches [15, 16, 63, 68, 85, 95, 96]. As switch bandwidth increases, these algorithms must be integrated into the switch's ASIC to achieve line-rate virtual priority [16, 95]. This necessitates switch replacement for deployment. Given the scarcity of priority queues and the limitations of deploying existing switch-based virtual priority mechanisms, we ask this question in this paper: *Can we achieve virtual priority without switch replacements or upgrades?*

Observations: Our observation is that virtual priority can be achieved by carefully managing bandwidth contention in a physical priority queue, which is traditionally managed by the Congestion Control (CC) algorithm. Hence, the virtual priority mechanism needs to be tightly coupled with CC. However, none of the existing CCs can achieve virtual priority. D2TCP [90] is a representative ECN-based CC that prioritizes urgent traffic as illustrated in Figure 1a. When the switch queue length exceeds ECN threshold, both high and low-priority flows receive ECNs and decelerate, thereby violating ***O*1** (§ 3.1). Ideally, low-priority flows should detect congestion and decelerate earlier, leaving high-priority flows unaffected. As demonstrated in Figure 1b, achieving this requires multi-bit congestion signals (*e.g.*, delay) and distinct thresholds for each priority. Our analysis of Swift [52], the state-of-the-art delay-based data center CC, also reveals its inability to support virtual priority directly (§ 3.2 and § 3.3).

PrioPlus: In this paper, we propose PrioPlus, a congestion control enhancement algorithm that can be integrated with existing delay-based CCs to enable virtual priority transmission. The key idea of PrioPlus is assigning specific delay ranges, referred to as channels, to each priority. Channels with larger delay thresholds are allocated to higher priorities. By restricting flows to transmit only when their delay is below or within the assigned channel and guiding the delay to converge to that channel, PrioPlus effectively meets the performance objectives of ***O*1** and ***O*2**. Implementing PrioPlus requires only CC modifications at the end host, ensuring easy deployment and fulfilling ***O*3**. PrioPlus addressed two challenges.

- **Find the sweet point between *O*1 and *O*2.** For a PrioPlus flow, there is a tension between avoiding interference with higher-priority flows (***O*1**) and maximizing bandwidth utilization (***O*2**), as the former requires the flow to be conservative while the latter requires aggression. PrioPlus carefully navigates the two objectives from various factors across a flow's lifecycle. (i) After yielding bandwidth to higher priority traffic, a PrioPlus flow employs *probing with collision avoidance* to gather timely congestion signals with minimal bandwidth usage. (ii) When starting in an ambiguous network environment, a PrioPlus flow adopts *linear start* strategy to start up quickly while minimizing potential buffer occupancy. (iii) When delay falls below the assigned range, a PrioPlus flow precisely raises the delay into the target range without overreaction through *dual-RTT multiplicative increase*.
- **Set channel width correctly and tightly.** A higher priority will be assigned larger delay thresholds, exceeding the summed channel widths of lower priorities, which may increase latency for high-priority flows if the channel widths are large. The channel width for a priority must accommodate the CC's normal delay fluctuations and delay measurement noises to avoid misreaction. PrioPlus employs *delay-based flow cardinality estimation* that regulate the overall aggressiveness of flows according to the observed delay to stabilize the CC's delay fluctuations. For noises caused by protocol offloading (*e.g.*, TSO [6]) and hardware and software jitters and so on, we establish a relationship between channel width and the magnitude of these noises to optimize performance (§ 4.3).

We implemented PrioPlus on top of Swift, adding only 79 lines of code within DPDK, demonstrating PrioPlus's compatibility and ease of deployment (***O*3**). Testbed evaluations confirm its ability to achieve ***O*1** and ***O*2** while effectively managing queue lengths in real-world conditions. Extensive simulations confirmed that, compared to Swift CC

with physical priority, PrioPlus provides strict priority for high-priority flows with no more than 9% degradation. It benefits low-priority flows from 25% to 41% as its priority-aware design enhances CC's ability to fully utilize available network bandwidth once higher-priority traffic completes. Consequently, PrioPlus outperforms Swift with physical priority queues by 21% and 33% in coflow scheduling and model training scenarios, respectively.

Our artifacts are publicly available online [8] as a contribution to the research community and to ensure reproducibility.

This work does not raise any ethical issues.

2 Motivation and Background

This section highlights the importance of priority queues in data center networks (§ 2.1) and emphasizes the need for virtual priority due to the limited availability of physical priority queues (§ 2.2). We then introduce the objectives of virtual priority and discuss existing mechanisms that, while promising, present challenges in deployment (§ 2.3).

2.1 The Need for a Large Number of Priority Queues

Priority queues are used to provide traffic *isolation* for different traffic classes. Data centers run multiple applications and services with varying Quality of Service (QoS) requirements within the network [81, 99], including latency-sensitive real-time services, coflow-dependent computing tasks, and throughput-sensitive backend operations [56, 93]. In addition, various transport protocols such as RDMA, DCTCP, and traditional TCP operate concurrently in data centers, often leading to compatibility issues when competing for network resources [9, 34, 47]. To accommodate these requirements, priority queues are employed to provide weighted bandwidth sharing for different QoS classes [45, 63, 99] and traffic using different protocols [9, 34, 55, 60, 72]. Priority queues are also employed in multi-tenant data centers for tenants [54, 61, 86] with differing priorities and QoS requirements [20, 84, 89].

Priority queues are used to perform traffic *scheduling* to achieve better performance in the same traffic class. Although optimal traffic scheduling in general networks is known as NP-hard [15], priority queue-based approaches can be utilized to achieve a 2-approximation to the optimum [21] by assigning flows to different strict priority queues based on their sizes [11, 18, 58, 64, 75, 97]. The effectiveness of these priority-based algorithms has been shown to improve with the number of utilized priority queues across various types of traffic, including independent flows [18, 58], coflows [11], RPCs [70], and distributed ML model training [25]. Furthermore, recent research [77, 78] finds that simply assigning different priorities to models' traffic in the machine learning cluster can improve the training speed of all models.

2.2 Only a Few Priority Queues Are Offered

Current commercial switches in data centers typically support only 8 [51] or 12 [24] priority queues, which cannot be increased in the short term due to limitations in protocols and hardware resources.

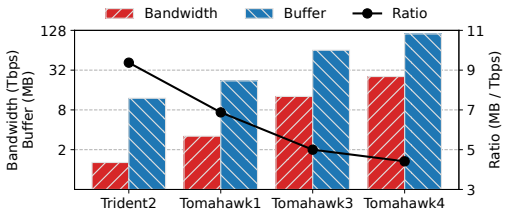


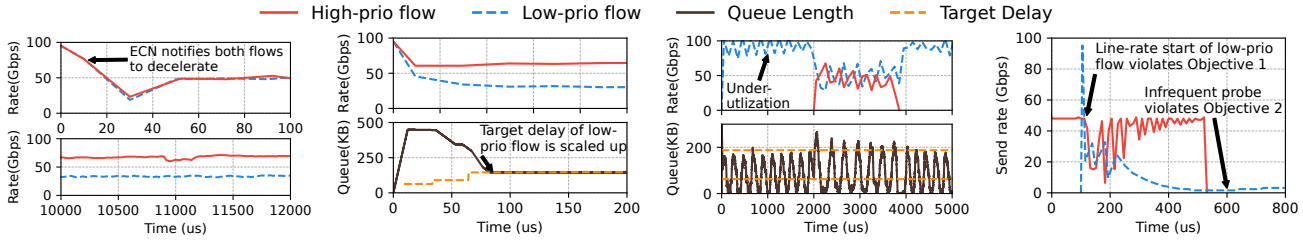
Figure 2. Buffer-to-bandwidth ratios of representative switch chips across different generations.

Priority-related protocols limit the number of priority queues. The DSCP (Differentiated Service Code Points) field in the IP header is used to designate priority [45], with standards defining only 12 mappings of DSCP code points to service classes [19]. Switch vendors may employ custom DSCP code points to support additional priorities. However, the inconsistency in DSCP mappings across different vendors can complicate its deployment[1]. Another protocol limitation is imposed by PFC [49], which is widely used in modern data centers [17, 42, 45] to provide lossless priorities for RDMA. The format of PFC frames [49] limits the number of lossless priorities to a maximum of 8. Since PFC is hardware-implemented [76], updating the PFC frame format requires data center operators to upgrade their switches.

The trend of decreasing buffer-to-bandwidth ratio in switches limits the number of priority queues. The number of priority queues in the switches is limited by the scarce on-chip SRAM resource, which contributes significantly to switch chip cost and power dissipation. Furthermore, PFC requires dedicated buffer headroom for each lossless priority, with the required size proportional to the bandwidth. As bandwidth increases consistently outpace buffer size growth [44, 71], supporting a large number of lossless priorities is increasingly difficult. Figure 2 shows the declining buffer/bandwidth ratios across different generations of switch chips. A study from Microsoft [45] reports that they accommodate merely two lossless priorities on Trident2, which has a buffer/bandwidth ratio of 9.4. In contrast, the newer Tomahawk4 chip only has a ratio of 4.4, hindering the support for additional priority queues.

Current priority queue usage. Due to the scarcity of priority queues, data center operators strive to allocate a single priority queue for each traffic type with different QoS requirements [45, 63, 99]. A report from Meta [34] indicates that the lack of priority queues forces them to place traffic from different CCs into the same queue, complicating the deployment of new CC. Although experiments and analysis show that priority-based traffic scheduling algorithms significantly enhance performance [11, 25, 70] within a traffic

[1] As an example, in HP switches [2], DSCP values 001010 and 001100 are mapped to the same priority, whereas in Cisco switches [3], they are mapped to different priorities.



(a) D2TCP flows with different DDLs converge to weighted allocation. **(b)** Swift flows with different targets converge to weighted allocation. **(c)** Swift w/o scaling fails for virtual priority under numerous flows. **(d)** Meticulous tradeoff is needed for achieving virtual priority.

Figure 3. Micro-benchmark of using existing CCs for virtual priority.

type, their large-scale deployment is hindered by the lack of priority queues [16].

2.3 Calling for Virtual Priority Capability

The growing demand for priority queues, along with the current shortage of *physical priority* queues, has made *virtual priority* queues an attractive solution. Virtual priority refers to the capability of supporting multiple *strict priorities* within a single physical priority queue. We argue that the limited physical priority queues in the network should be reserved for bandwidth sharing and performance isolation across different traffic classes (e.g., traffic from distinct product areas) , while enabling virtual priority at the transport layer on the end host to offer strict priorities within each traffic class for traffic scheduling, thereby enhancing performance. To achieve this vision, virtual priority should achieve the following objectives.

- ***O*1: Multi-priority assurance**. The virtual priority mechanism must support the transmission of multi-priority traffic through a single physical queue, ensuring strict prioritization of higher-priority traffic.
- ***O*2: Work conservation**. The virtual priority mechanism should aim to fully utilize bandwidth without any waste.
- ***O*3: Readily deployable**. The virtual priority mechanism should be readily deployable in existing data centers without hardware upgrades or overuse of scarce resources.

Existing virtual priority technologies primarily involve packet scheduling algorithms that operate on switches. However, they necessitate programmable switches [96] or additional specialized hardware [15, 16, 63, 68, 85, 95, 100]. Given the scarcity of priority queues and the deployment challenges of existing switch-based virtual priority mechanisms, we raise the question: *Can we achieve virtual priority without switch replacement/upgrades?*

3 Design Rationale

Our insight is that virtual priority can be achieved by carefully managing bandwidth contention among flows within a physical priority queue, a task traditionally handled by the Congestion Control (CC) at the host. Therefore, the virtual priority mechanism must be tightly integrated with CC.

Through micro-benchmarks on existing CCs, we derive three observations for integrating virtual priorities into CC, guiding the design of PrioPlus. (i) A CC supporting virtual priority must utilize a multi-bit congestion signal and set distinct congestion thresholds for different priorities (§ 3.1). (ii) It should effectively control queue fluctuation (§ 3.2). (iii) It should cautiously balance multi-priority assurance (***O*1**) with work conservation (***O*2**) (§ 3.3).

We utilize the ns-3 simulator [5] to analyze bandwidth contention in a physical priority at a bottleneck port. The topology comprises a switch connected to multiple hosts, with one host acting as the receiver and the others as senders. The switch's port to the receiver is the bottleneck. The link bandwidth is set at 100 Gbps, and the round-trip time (RTT) is 12 μs to mimic a typical data center network environment.

3.1 The Need for the Multi-bit Congestion Signal

The state-of-the-art data center CC that can prioritize a proportion of traffic is D2TCP [90], an extension of DCTCP [14][2]. It prioritizes flows with imminent deadlines (DDL) by reducing their rate less significantly. We run two D2TCP flows with deadlines set at one and two times the ideal flow completion time (ideal FCT, calculated as flow size divided by bandwidth), respectively. To meet the established DDLs, D2TCP should first allocate all bandwidth to the high-priority flow (the flow with DDL as the ideal FCT), ensuring it completes within one ideal FCT. Once it completes, D2TCP should then allocate all bandwidth to the low-priority flow, *i.e.*, strictly prioritizing the high-priority flow without any bandwidth waste.

The results are shown in Figure 3a. As shown in the upper subplot of Figure 3a, when the network notifies ECN to both senders, they reduce their sending rates simultaneously, thereby compromising the rate of the high-priority flow and undermining ***O*1**. Additionally, as shown in the lower subplot, when two flows converge, the high-priority flow is not strictly prioritized. This is because when an ECN is received, high-priority flow will reduce its rate. Although this reduction is moderate, it prevents high-priority traffic from monopolizing the bandwidth (*i.e.*, violate ***O*1**).

Ideally, the network should converge to a state where high-priority flows do not perceive congestion and maintain their rate, while low-priority flows continuously detect congestion

[2]Other CCs that can prioritize traffic, such as LEDBAT [80] and PCC Proteus [66], are internet CCs and only provide an additional background priority. They are discussed in detail in § 7

and decrease their rate until they relinquish all bandwidth. However, this is unattainable by single-bit congestion signals like ECN that notifies all flows of congestion once the queue length surpasses the ECN threshold. Therefore, we have:

Observation 1 *: CC that supports virtual priority requires a multi-bit congestion signal and setting distinct congestion thresholds for flows with different priorities.*

3.2 The Need for the Queue Fluctuation Management

As demonstrated in § 3.1, implementing virtual priority necessitates that higher-priority flows do not decelerate during bandwidth competition, while lower-priority flows promptly decelerate. This can be achieved by utilizing multi-bit congestion signal, such as delay [52] and delay gradient [69], and assigning higher congestion thresholds to higher-priority flows and lower ones to lower-priority flows. When the congestion signal falls between the thresholds of different priorities, higher-priority flows can tolerate the congestion and maintain their speed, whereas low-priority flows progressively slow down.

Therefore, we next investigate whether existing multi-bit congestion signal-based CC, *i.e.* Swift [52], can achieve virtual priority by setting different thresholds. In our experiments, we set a delay threshold (target delay in Swift) of base RTT plus 15 μs for high-priority flows and base RTT plus 5 μs for low-priority flows. Note that the higher target delay in Swift indicates higher-priority flows decelerate more conservatively than lower-priority flows during bandwidth contention, rather than necessarily experiencing high delay as the actual delay depends on queue length.

Figure 3b illustrates the interaction between two high-priority and two low-priority flows, with the upper subplot displaying the total send rate of both types of flows. It is observed that Swift does not strictly prioritize high-priority traffic. This is due to Swift's target scaling mechanism. When the flow rate decreases, Swift tends to assume an increase in the number of flows within the network, consequently raising the target delay to accommodate the queue fluctuations caused by numerous flows [52]. As illustrated in the upper subplot, after rate reductions, low-priority flows increase their target delay, allowing for a weighted sharing of bandwidth with high-priority flows (*i.e.*, violate ***O*1**).

Figure 3c illustrates an experiment in which 300 low-priority flows, starting at 0 ms, compete with a single high-priority flow that begins at 2 ms, using the Swift congestion control algorithm without target scaling. Two horizontal lines in the lower subplot represent the target delays of high and low priorities, respectively. Without target scaling, Swift fails to maintain the queue length near the target delay of low priority, resulting in under-utilization of bandwidth (*i.e.*, violate ***O*2**). Once the high-priority traffic appears, the delay fluctuations exceed the high-priority flow's target delay, making it decelerate (*i.e.*, violate ***O*1**)

As we can see, Swift cannot achieve virtual priority regardless of whether target scaling is enabled. This limitation stems from its ineffective management of queue fluctuations caused by numerous flows. In summary, we have

Observation 2 *: Queue fluctuation management is essential for virtual priority CC to avoid sending erroneous congestion signals to flows of varying priorities.*

3.3 Virtual Priority Necessitates Meticulous Tradeoff

We then investigate whether implementing virtual priorities requires additional meticulous design. To avoid the issues discussed in § 3.2, we examine Swift without target scaling in a scenario with two high-priority flows and two low-priority flows. The two high-priority flows converge initially, followed by the commencement of the low-priority flows at 100 μs. Figure 3d shows the rates of one high-priority flow and one low-priority flow. Two major tradeoffs between aggressive and conservative are observed.

(i) Trade-off between flow start rate and buffer occupation. RDMA CCs commonly start flows at the line rate. However, the blind line rate start of low-priority flows leads to substantial buffer occupation, resulting in the deceleration of high-priority flows (*i.e.*, violate ***O*1**). Conversely, starting below the line rate may lead to bandwidth underutilization, prolonging the completion time of flows (*i.e.*, violate ***O*2**).

(ii) Trade-off between signal frequency and bandwidth occupation. For most CCs, congestion signals are piggybacked by the ACK. Thus, CCs must keep a minimum send rate to ensure the periodic reception of congestion signals. In the experiment, after around 400 μs, the rate of the low-priority flow decelerates to the minimum rate limit, which is set to 100 Mbps in the experiment. This is corresponding to sending one packet every 84 μs. When the high-priority flows cease, the low-priority flow experiences an 80μs idle before detecting the end of contention by the next packet's ACK. This results in a slow rate increase, causing bandwidth under-utilization, thereby violating ***O*2**. Conversely, increasing the frequency of congestion signals means raising the minimum rate limit, which would incur greater bandwidth occupation, especially when there are many low-priority flows (*i.e.*, violate ***O*1**). In summary, we have

Observation 3 *: Many existing CC designs are not suitable for virtual priority CC, which fail to balance multi-priority assurance (**O*1*) with work conservation (**O*2*).*

4 Design

In this section, we introduce PrioPlus, a congestion control enhancement algorithm that can be integrated into existing delay-based CCs to enable virtual priority.

4.1 Overview

Key idea. PrioPlus assigns specific delay ranges, termed as *channels*, to each priority. Channels with larger thresholds are assigned to higher priorities. A channel i has two delay thresholds, D^i_{target} and D^i_{limit}, satisfying $D^{i-1}_{limit} < D^i_{target} <$

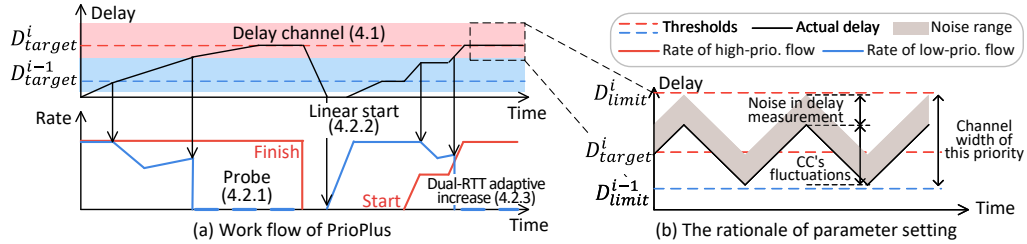


Figure 4. Overview of workflow and parameter setting for adjacent priorities i (higher) and $i-1$ (lower).

Priority	Priority Number	Delay Threshold
Higher	Larger	Larger
Lower	Smaller	Smaller

Table 1. Key trems used in the paper.

D^i_{limit} By restricting flows of priority i to control delay to its D^i_{target} and to suspend transmission when delay larger than its D^i_{limit}, PrioPlus meets the performance requirements of ***O*1** and ***O*2**, thus achieving virtual priority.

Following the convention in scheduling research [25, 70], we use larger numbers to denote higher priorities. Higher priorities are assigned channels with larger delay thresholds. To ensure clarity of our terminology, we list key terms in Table 1. Note that higher delay thresholds indicate that higher priorities decelerate more conservatively in bandwidth contention, rather than necessarily experiencing higher delays, as elaborated in § 3.1 and will be demonstrated in § 6.3.

Integrate to existing delay-based CCs. PrioPlus can integrate with most delay-based CCs (referred to as original CC) that set a target delay for flows and adjust their windows or rates to maintain the delay close to this target. PrioPlus integrates with original CC by assigning its target delay to D_{target} (and disabling any target scaling mechanisms), modifying CC's window or rate, and fine-tuning its AI (Additive Increase) step. In this section, we present the design of PrioPlus with Swift, the state-of-the-art delay-based data center CC. We also integrate PrioPlus with LEDBAT [80], with methods and effects discussed in § 4.4 and § 6.2. PrioPlus can leverage either RTT (Round-Trip Time) or OWD (One-Way Delay) depending on the delay metric used by the original CC algorithm. For simplicity, we will focus on RTT in the following sections, as it is used by Swift.

Challenges. Implementing virtual priority is not merely about setting different delay ranges for each priority. There are two primary challenges.

- *Find the sweet point between **O*1 *and **O*2 *(§ 4.2).* Current CCs lack dedicated design considerations for virtual priority, encompassing various aspects of a flow's lifecycle, such as maintaining congestion signal frequency after yielding bandwidth, initiating low-priority flow without interfering with higher-priority flows, and quickly adjusting delay to the designated channel. As highlighted in Observation 3, achieving this requires a careful balance between aggressive and conservative strategies.
- *Set channel width correctly and tightly (§ 4.3).* A higher priority will be assigned larger delay thresholds, exceeding the summed channel widths of lower priorities. This allocation may increase latency for high-priority flows, thereby impacting end-to-end performance. As shown in Figure 4b, the channel width must accommodate both CC's normal fluctuation and any noise in delay measurement to prevent misreactions. To minimize channel width, it is essential to regulate the amplitude of CC's fluctuations and to figure out the relationship between the noise and the channel width.

Workflow. We introduce PrioPlus's workflow through the example shown in Figure 4a.

(i) Delay above D_{limit}. When the delay exceeds D_{limit}, the low-priority flow infers that higher-priority flows are transmitting and, therefore, halts its transmission, relinquishing bandwidth to the higher-priority flow. Then it employs a *probe with collision avoidance* scheme to obtain high-frequency congestion signals with minimal bandwidth usage (§ 4.2.1).

(ii) Delay equal to base RTT[3]. After the high-priority flow finishes, the low-priority flow observes a delay equal to base RTT, which implies that the path's bandwidth utilization may vary from being completely idle to just adequately utilized but no queues are built up. In such circumstances, PrioPlus employs a *linear start* strategy to balance switch buffer occupancy and flow startup speed (§ 4.2.2).

(iii) Delay between base RTT and D_{target}. When the second high-priority flow starts, it observes a delay between base RTT and D_{target}. The flow deduces that the network traffic consists of lower-priority flows. Consequently, the flow employs a *dual-RTT multiplicative increase* strategy to promptly elevate the delay to D_{target} to prompt lower-priority flows relinquish the bandwidth while avoiding overreaction (§ 4.2.3).

(iv) Delay around D_{target} and below D_{limit}. In this circumstance, the flow enters its designated channel and employs the original CC to regulate the flow rate.

Compress and configure the channel width. As shown in Figure 4b, the channel width must accommodate (i) CC's normal fluctuations and (ii) the noise in delay measurement.

To compress the channel width, (i) we adopt a *delay-based flow cardinality estimation* to reduce delay fluctuations. Once the delay exceeds D_{limit}, PrioPlus estimates the cardinality of flows with the same priority based on the current delay and its own window size, and accordingly adjusts the overall aggressiveness of the flows in the network to reduce delay fluctuations. (ii) For noise caused by protocol offloading (*e.g.*, TSO) and hardware and software jitters, we propose a *filter mechanism* to filter out infrequent large delay noises (§ 4.3.1).

[3]The base RTT represents the RTT when a packet does not experience queuing delays, which can be directly calculated in data centers [52].

Subsequently, we present the methods for configuring channel width in different use cases and detail the applicability of PrioPlus (§ 4.3.2).

The overall algorithm is specified in Algorithm 1.

4.2 Balancing Aggressive and Conservative Tactics

In this section, we describe PrioPlus's mechanisms for balancing aggressive and conservative tactics across three scenarios. (i) obtain congestion signals with minimal bandwidth occupation, (ii) startup in ambiguous network conditions, and (iii) increasing delay to D_{target} while avoiding overreaction.

4.2.1 Probe with Collision Avoidance

As shown in Observation 3, the low-priority flows face a tradeoff in choosing the minimum rate. PrioPlus addresses this tradeoff through *probe with collision avoidance*. Probe is used for a lower-priority flow when it starts or when it relinquishes all bandwidth due to higher-priority flows presenting. High-priority flows should start transmission without a probe. We will discuss this design choice in § 4.4.

When a probe is needed, the sender sends a minimal-sized probe packet, which will be echoed back once received by the receiver, allowing the sender to determine the RTT. Considering sending a 64-byte probe packet per base RTT (*e.g.*, 12us), the bandwidth occupation is only 42 Mbps, allowing for the checking of the delay with each base RTT.

However, the straightforward probe mechanism presents two issues. (i) Collisions. In the presence of high-priority flows, all low-priority flows simultaneously start to probe and then restart transmission if the probed delay is low, potentially leading to transient congestion. (ii) Bandwidth occupancy. Although a single flow requires only 42 Mbps for one probe per base RTT, probe packets can impose a heavy load on the bandwidth when there are numerous flows.

PrioPlus employs *probing with collision avoidance* which is inspired by *priority assisted CSMA/CA*, an advanced technique used in WLANs [82, 88]. When $delay$ exceeds D_{limit}, PrioPlus flow sends a probe after $(delay - D_{target}) + random(BaseRtt)$ where $random(BaseRtt)$ is a random value between 0 and base RTT (line 22). The former item leverages the temporal locality of traffic, predicting that the queue length (*i.e.*, delay) will not reduce to D_{target} (the target delay of this priority) within the time frame of $delay - D_{target}$. Additionally, in networks with flows at multiple priorities probing, it keeps the probing frequency of higher-priority flows while decreasing the bandwidth usage of lower-priority ones. The randomization in the latter item can reduce collision probabilities.

If the probe ACK indicates that the delay still exceeds D_{limit}, the next probe will be sent following the method described above (line 27). Probe losses are recovered through the original CC's RTO (Retransmission Timeout). When a probe ACK is received, PrioPlus resets the original CC's RTO.

4.2.2 Linear Start

An important trade-off emerges in starting a flow when facing an uncertain network environment. This situation occurs when observed delays are equal to the base RTT or when a new flow starts without a probe. A delay equal to the base RTT implies that the path utilization could range from being completely unused to just adequately utilized.

RDMA CCs typically *start data transmission at line rate* to fully utilize the bandwidth [102]. However, when transmitting at line rate, a single lower-priority flow could cause one BDP of buffer backlog in the worst-case scenario (*i.e.*, the path was fully utilized), impairing the higher-priority flows.

From another perspective, TCP CCs employ an *exponential start strategy* [37], which starts at a low *cwnd* (congestion window) and doubles the *cwnd* every RTT if no congestion is detected[4]. However, the low starting rate may result in

Algorithm 1: PrioPlus's Main Algorithm

```
1  Procedure NewAck(delay)
2      if pkt.seq > rttEndSeq then
3          rttPass ← true, rttEndSeq ← sndNxt
4          dualRttPass ← ¬dualRttPass
5          if dualRttPass == false then
6              W_AI ← W_AIorigin / #flow   // End of adaptive increase (4.2.3)
7      if delay >= D_limit && consec++ ≥ 2 then
           /* Flow cardinality estimation (4.3.1)                    */
8          #flow ← max(#flow, delay·LineRate / cwnd)
9          W_AI ← W_AIorigin / #flow, countDown ← BaseBdp / W_LS
10         StopSending(), ScheduleProbe(delay)   // Probe (4.2.1)
11     else
12         if delay <= D_target && rttPass then
13             if delay == BaseRtt then
14                 cwnd ← cwnd + W_LS / #flow   // Linear start (4.2.2)
15                 if countDown == 0 then #flow ← #flow · 2
16                 else countDown ← countDown − 1
17             else if dualRttPass then
18                 W_AI   // Start of Dual-RTT adaptive increase(4.2.3)
19                     ← W_AI + min(cwnd/2, (D_target − delay)/delay · cwnd)
20             consec ← 0                                   // (4.3.2)
21         OriginalCC(delay)
22 Function ScheduleProbe(delay)
23     probeDelay ← delay − D_target + random(BaseRtt)
24     SendProbeAfter(probeDelay)
25 Function NewProbeAck(delay)
26     if delay >= D_limit then
27         ScheduleProbe(delay) , return
28     if delay == BaseRtt then
29         cwnd ← W_LS / #flow                          // Linear start (4.2.2)
30         if countDown == 0 then #flow ← max(1, #flow/2)
31         else countDown ← countDown − 1
32     else
33         cwnd ← PktSize   // Set the window conservatively (4.4)
34     ResumeSending()
35     rttEndSeq ← sndNxt, dualRttPass ← false
```

[4]To provide a clearer explanation, the following analysis will use rate to describe CC's behavior.

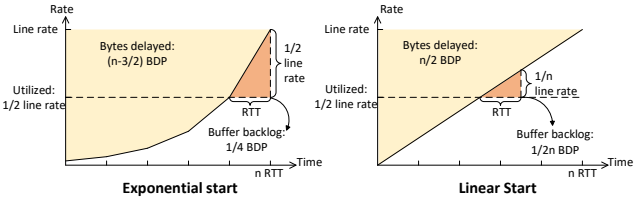


Figure 5. Illustration of exponential start and linear start.

Flow Start Strategy	Bytes delayed	Maximum extra buffer
Line-rate start	0 ✓	1 BDP ✗
Exponential start	$n - 3/2$ BDP ✗	0.5 BDP ✗
Linear start	$n/2$ BDP ✗	$1/n$ BDP ✓

Table 2. Comparison of start strategies.

significant delays in data transmission. Assume it takes n RTT to reach line rate, the left diagram of Figure 5 shows the amount of delayed bytes (compared to line-rate start), which is represented by the area above the rate curve as $n - \frac{3}{2}$ BDP. Moreover, it does not effectively reduce the additional buffer backlog. In a scenario where link utilization is 50%, the link becomes fully utilized exactly at $n - 1$ RTTs. The sender can only observe the queue buildup and stop increasing the rate at n RTT. During the last RTT, the sender increases the rate by $\frac{1}{2}$ line rate, resulting in a buffer backlog of $\frac{1}{4}$ of the BDP.

We notice that a flow's maximum potential buffer occupancy is related to its rate increase per RTT. To achieve a line rate within n RTTs, the strategy with the *least* potential buffer occupancy is to *linearly* accelerate by $\frac{1}{n}$ of the line rate each RTT. As shown in the right diagram of Figure 5, a linear startup over n RTTs only incurs $\frac{n}{2}$ BDP delay in data transmission and additional $\frac{1}{2n}$ of BDP buffer backlog.

Based on this, we propose the *linear start* strategy, whereby a flow starts transmitting with *cwnd* set to W_{LS} (window step for linear start) (line 29), and if no queue buildup is observed, its rate increases by W_{LS} per RTT (line 14). As illustrated in Table 2, compared to the line-rate start and exponential start, the linear start only delays the transmission by $\frac{n}{2}$ BDP, while at most occupying $\frac{1}{2n}$ of the base BDP, efficiently balancing the bandwidth utilization and the potential extra buffer occupancy. Linear start is also adopted when an empty queue is observed during data transmission (line 14).

We prove the following theorem through *Variational method*, with the detailed proof placed in Appendix C.

Theorem 4.1. *Linear start is the start strategy that incurs the least potential buffer backlog when increasing the send rate from 0 to the line rate in a given period.*

4.2.3 Dual-RTT Adaptive Increase

Delay between base RTT and D_{target} indicates that only lower-priority flows are transmitting. In such cases, a flow should raise the delay to D_{target} (higher than D_{limit} of lower priorities) to prompt lower-priority flows to relinquish bandwidth. Using a predefined increase step size fails in this scenario as, for instance, a fixed increase step size may be effective with ten flows but is inadequate for a single flow and excessive for a hundred flows.

PrioPlus employs an *adaptive increase* approach to rapidly and accurately elevate the delay to D_{target}, regardless of the number of flows. Consider a scenario where n flows are at the same priority, and the current delay is $delay$. The amount of inflight data is given by the formula $delay \cdot LineRate$, which equals the total $cwnd$ of n flows, *i.e.*, $\sum_{i=1}^{n} cwnd_i$. To elevate the delay to D_{target}, each flow's window size needs to be increased by the ratio $\frac{D_{target} \cdot LineRate}{delay \cdot LineRate}$, which means increasing the $cwnd$ by the step $\frac{D_{target} - delay}{delay} \cdot cwnd$.

Adaptive increase could significantly elevate the delay. To prevent excessive fluctuations in delay, PrioPlus constrains that the step of each adaptive increase does not exceed $\frac{cwnd}{2}$. Rather than directly enlarging the window size after determining the adaptive increase step, PrioPlus incorporates this step within the original CC's additive increase (AI) step (line 19). Then the original CC's gradual window expansion mechanism (*e.g.*, increase the window $\frac{W_{AI}}{cwnd}$ per ACK) will complete the adaptive increase within one RTT.

To avoid overreactions, PrioPlus initiates an adaptive increase every two RTTs. We use the scenario depicted in Figure 6 to illustrate the rationale. Assume that the per-hop delay is 1 second and the bottleneck has an output rate of 0.5 packets per second. Initially, packet $p0$ is queued at the bottleneck, and the $cwnd$ is 3 packets. After $p1$ is sent, it takes 1 second to arrive at the bottleneck, where $p0$ is just being dispatched. Consequently, $p1$ waits 2 seconds at the bottleneck, and it takes another 3 seconds for the acknowledgment of $p1$ to return to the sender, marking the end of $RTT1$. After observing a 4-second delay at the end of $RTT1$, the sender initiates an adaptive increase, expanding the window to 4 packets and sending packets $p4$ to $p7$ during $RTT2$. However, by the end of $RTT2$ (marked by the ACK of $p4$), the expanded window has not yet affected the delay, which remains at 4 seconds. By the end of $RTT3$, when $p8$ is acknowledged, the delay increases to 6 seconds, corresponding to the 1 packet of $cwnd$ increase. In conclusion, the delay elevation only becomes observable precisely two RTTs after initiating the adaptive increase. If we perform another adaptive increase at one RTT after the initial one, it amounts to applying two consecutive adaptive increases to the same network conditions, leading to overreaction.

Therefore, PrioPlus utilizes a dual RTT design for adaptive increase. The *dualRttPass* is a boolean variable that toggles at the end of each RTT (line 4). An adaptive increase is executed when *dualRttPass* is true and the delay is below D_{target} (line 17). When *dualRttPass* turns false, PrioPlus resets the W_{AI} to $W_{AIorigin}$ (line 6).

4.3 Compress and Configure the Channel Width

As shown in Figure 4b, the channel width must accommodate both CC's normal fluctuations and the noise in the delay measurement. To prevent channel width from being excessively large, we propose two methods to mitigate CC's fluctuations and manage delay noise (§ 4.3.1). Then we detail

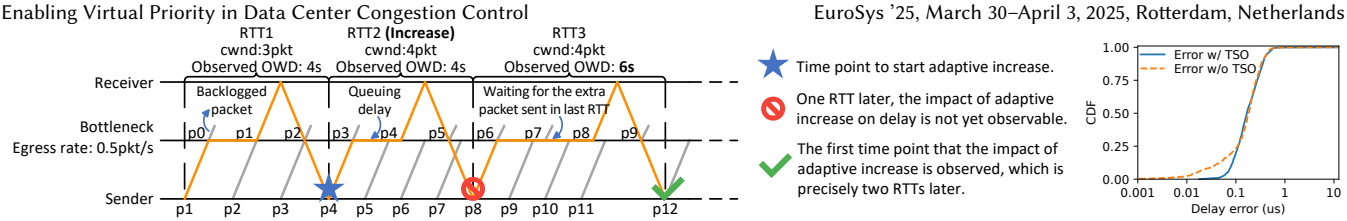


Figure 6. After conducting an adaptive increase, it takes two RTTs to observe the effect in delay.

Figure 7. Delay noise.

the configuration of the channel width in various use cases (§ 4.3.2).

4.3.1 Compress the Channel Width

Alleviate CC's fluctuations. CCs typically show increasing fluctuations with the number of flows, as each flow independently adjusts its rate using a fixed AI step [14, 43, 52, 102]. Additionally, PrioPlus adopts a linear startup mechanism with a step size of $\frac{1}{n}BaseBdp$ upon detecting an empty queue. When the number of flows exceeds n, a problematic cycle may occur: an empty queue triggers a linear start for each flow, which causes the delay exceeding D_{limit}, halting transmission and resulting in an empty queue again.

PrioPlus address this issue by *delay-based flow cardinality estimation*. When the delay exceeds D_{limit}, the flow estimates the flow cardinality $\#flow$ (*i.e.*, the number of active flows at the same priority level) as the ratio of inflight size $delay \cdot LineRate$ to its own $cwnd$ (line 8). Based on the estimation, the flow proportionally adjusts the W_{AI} (lines 6 and 9) and W_{LS} (lines 10 and 27). In this way, when the delay fluctuates exceeding D_{limit} due to numerous flows, the aggressiveness of PrioPlus flows can be controlled based on the estimated flow cardinality. The cardinality-based adjustment does not compromise bandwidth utilization, as the total increase step of all adjusted flows is similar to that of a single flow.

PrioPlus flow employs a countdown mechanism to reset the estimated flow cardinality. Each time the flow cardinality is estimated, the *countdown* is set to $\frac{BaseBdp}{W_{LS}}$ (line 9). For each RTT that observes a delay equal to the base RTT, the *countdown* is decremented by one if it is larger than zero; otherwise, the estimated flow cardinality $\#flow$ is halved. The rationale is that when the RTT equals the base RTT, all flows will adopt the linear start strategy (§ 4.2.2). If the number of flows matches the estimated cardinality $\#flow$, the total $cwnd$ of concurrent flows will exceed the $BaseBdp$ after $\frac{BaseBdp}{W_{LS}}$ RTTs. If the delay remains equal to the base RTT after this, it indicates that some flows have finished, leading to an overestimation of the flow cardinality.

Accommodate the delay noise. As shown in previous works [53, 69] and our latter measurement (§ 4.3.2), delay noise in modern data centers exhibits a long-tail distribution, meaning that large noises are infrequent. PrioPlus employs a simple and effective *filter mechanism* to filter out infrequent large delay noises: a flow relinquishes bandwidth only when the delay exceeds the D_{limit} in two consecutive measurements (lines 7 and 20). This design allows data center operators to easily adjust the channel width to accommodate delay noise. Operators could first measure noise in their data centers and then expand the channel width according to a high percentile of the delay noise.

4.3.2 Configure the Channel Width

The configuration of the channel width is dependent on the use cases. We next introduce how to determine CC fluctuations and delay noise across various use cases and how to set the channel width accordingly.

Determine CC's fluctuations. CC fluctuations can be estimated via theoretical tools, such as fluid model [83] and discrete cycle analysis [74, 91]. As an example, Swift exhibits a fluctuation of $\frac{n \cdot W_{AI}}{LineRate} + \max\left(\frac{n \cdot \beta \cdot W_{AI}}{LineRate \cdot Target}, max\_mdf\right) \cdot Target$, where W_{AI} is the AI step, n is the number of competing flows, $Target$ is the target delay, β and max_mdf are the parameters used in multiplicative decrease (Appendix D). Operators can estimate fluctuations based on the typical flow count in the scenario. If the actual flow count is lower, no negative effects occur. Otherwise, the *flow cardinality estimation mechanism* (§ 4.3.1) can effectively mitigate them.

Characterize delay noise. Noises in delay measurement may stem from jitter in software and hardware, protocol offloading techniques (*e.g.*, TSO [6]), and so on. In data centers, noise can be measured through ping-pong packets between hosts under the same top-of-rack (ToR) switch when the network is idle. Since delay noise is additive noise [53] (*i.e.*, the measured delay will always be larger than the actual network delay), the minimum value from a series of delay measurements can serve as the baseline. The distribution of delay noise is then determined by subtracting this baseline from the remaining measurements. A single host can also complete the aforementioned process by sending packets to its own IP address if the protocol stack allows this operation.

Next, we present the typical delay noise of modern data centers observed in our laboratory environment. Currently, data centers commonly utilize hardware timestamps on network interface cards (NICs) to precisely record packet send and receive times, with errors usually within 1 μs [52, 69]. Figure 7 shows the Cumulative Distribution Function (CDF) of noise in delay measured through NIC hardware timestamping in our testbed (§ 5), with both TSO enabled and disabled. The average delay noise is approximately 0.3 μs, with merely less than 0.1% probability of exceeding 1 μs.

Set channel thresholds. As shown in Figure 4b, the channel width must accommodate CC's normal fluctuations and delay noises. For priority i, the distance between its target D^i_{target} and the previous priority's maximum delay D^{i-1}_{limit}

should accommodate half of the CC fluctuations, while the distance between its target D^i_{target} and its limit D^i_{limit} should accommodate both half of the CC fluctuations and the delay noise. Once a flow's priority i is determined, the host can configure the channel parameters of this flow as $D^i_{target} = BaseRtt + i \cdot (A + B)$ and $D^i_{limit} = D^i_{target} + \frac{A}{2} + B$ with A denotes CC fluctuations and B denotes tolerable delay noise.

In this paper, we use 3.2 μs to accommodate CC fluctuation, corresponding to fluctuations of 150 swift flows. According to the measured delay noise (Figure 7), we select the 99.85^{th} percentile of delay noise (0.8 μs) as the tolerable delay noise. The *filter mechanism* (§ 4.3.1) ensures that the misreaction only occurs every 400 MB data transfer on average[5]. Therefore, in the following evaluations, the channel width is set to 4 μs with $D^i_{target} = 4 \cdot i$ μs and $D^i_{limit} = 4 \cdot i + 2.4$ μs.

Use cases. PrioPlus can be applied to a variety of use cases. The primary use cases of PrioPlus are high-performance networks, such as parameter plane networks for model training [39, 50] and modern storage clusters [42]. In such use cases, CCs are implemented in kernel-bypass software [65, 101] or NICs [67], where the noise in delay measurement at the μs level. PrioPlus can also work in use cases where non-congestive delay exists. Such scenarios include data center networks with middleware [98] and regional inter-DC data transmissions [17]. In these scenarios, operators can first measure the non-congestive delay using well-established methods [98], then incorporate the fixed part into the base RTT and the variable part into delay noise. Experiments demonstrate that PrioPlus can operate effectively with non-congestive delay as long as non-congestive delay variations are limited in a range (e.g., $< 30\mu s$) (§ 6.3).

4.4 Miscellaneous Design Discussion

Whether to probe before data transmission. For high-priority or latency-sensitive PrioPlus flows, data transmission can be initiated with a linear start without probing. This helps to avoid increased latency, while the linear start's conservative nature ensures that it does not significantly impact higher-priority traffic.

Probed delay between base RTT and D_{target}. When the probed delay is between the base RTT and D_{target}, the PrioPlus flow adopts a conservative strategy as setting the $cwnd$ to one packet. This is because only one delay measurement is available, necessitating caution to prevent potential delay fluctuations. If subsequent delay measurements are between base RTT and D_{target} , the PrioPlus flow will rapidly raise the delay towards D_{target} through adaptive increase (§ 4.2.3).

Setting of W_{LS}. The linear start step size W_{LS} determines the time it takes for a flow to accelerate from zero to full speed (§ 4.2.2). If we desire this acceleration to occur over n RTTs, the step size should be set to $\frac{BaseBdp}{n}$. For high-priority, delay-sensitive traffic, we recommend a step size equal to $BaseBdp$. For medium-priority, non-delay-sensitive traffic, a step size of $0.25 \cdot BaseBdp$ is advised. For low-priority traffic, we recommend a step size of $0.125 \cdot BaseBdp$.

Reverse congestion. PrioPlus assumes that ACKs are transmitted at the highest physical priority to avoid the impact of reverse congestion on RTT, which is a common practice in modern data centers [23, 39, 46]. We demonstrate through experiments in the Appendix A.3 that PrioPlus is only slightly affected when ACKs cannot be prioritized.

[5]Consider the MTU is 1KB and per-packet ACK is adopted.

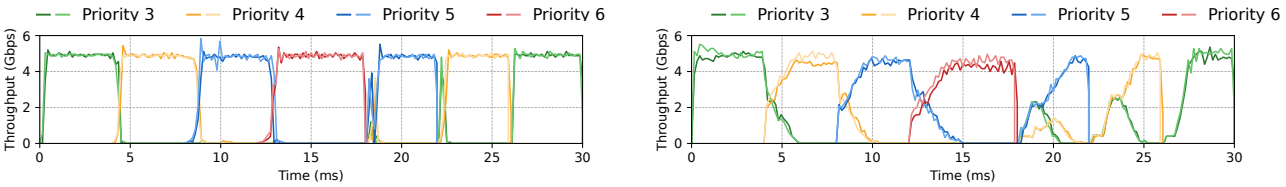
5 Implementation and Testbed Results

In this section, we use two testbed experiments to demonstrate the software-implemented PrioPlus's ability to achieve ***O*1** and ***O*2** while effectively managing delay fluctuations. Additionally, we discuss the feasibility of implementing PrioPlus in RDMA NIC (RNIC) hardware.

DPDK implementation and testbed setup. We implement PrioPlus in Linux using DPDK [38]. Based on a Swift implementation, we add 79 lines of code to incorporate PrioPlus. Our testbed uses a tree topology with 10 Gbps links, where four leaf nodes are senders, and the root is the receiver. The RTT is around 13 μs. Each node is equipped with an E5-2650 2.9 GHz CPU and Mellanox ConnectX-5 NIC.

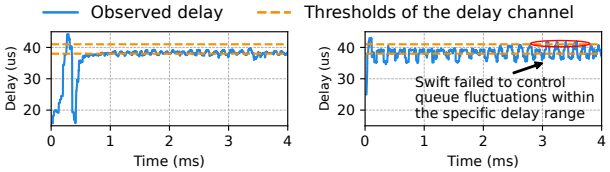
PrioPlus can balance *O*1 and *O*2. The closer the priorities, the smaller the difference in their delay ranges, which demands higher precision for the algorithm's ability to distinguish between priorities. Therefore, we select four adjacent priorities (3, 4, 5, 6), each with two flows, to evaluate PrioPlus in real implementations. The delay channel setting corresponds to § 4.3.2. As shown in Figure 8, flows of the four priorities start from the lower priority to the higher priority at 4 ms intervals and end at 4 ms intervals. Figure 8a shows the throughput of PrioPlus with Swift. Upon starting of higher-priority flows, lower-priority flows quickly yield the bandwidth (***O*1**). Once the higher-priority flows cease, multiple lower-priority flows concurrently accelerate through Linear Start. Following a brief contention, remaining higher-priority flows quickly consume the available bandwidth again (***O*2**). As a comparison, we test Swift with target delays aligned with D_{target} of PrioPlus. The target scaling is disabled. For Swift, lower-priority flows yield bandwidth entirely within about 2 ms after the higher-priority traffic emerges. After the higher-priority flows cease, it takes about 3 ms for the lower-priority traffic to fully utilize the bandwidth again.

PrioPlus can effectively manage delay fluctuations. In our second experiment, we demonstrate that PrioPlus can effectively manage queue fluctuations under numerous flows. We use four flows and increase the step size for each flow to simulate the fluctuations of numerous flows. We set W_{AI} for Swift to 0.75 KB, approximately five times the recommended value, and the W_{LS} for PrioPlusto 75 KB, which is half of the base BDP. The D_{target} and D_{limit} for PrioPlus flows are set to 37 μs and 39.4 μs (priority 6), respectively, while Swift's target delay is set to 37 μs. Figure 9 shows the observed delays for one of the flows. After the PrioPlus flows start,



(a) PrioPlus with Swift **(b)** Swift.

Figure 8. Throughput of flows with four virtual priorities.



(a) PrioPlus with Swift **(b)** Swift.

Figure 9. Observed delay by one of the flows.

the delay exceeds D_{limit}. Subsequently, PrioPlus estimates the flow cardinality, thus controlling overall aggressiveness and keeping the delay near the target. In contrast, Swift struggles to control delay fluctuations, with delays frequently exceeding the threshold. This validates the effectiveness of the delay-based flow cardinality estimation (§ 4.3.1).

Implementation on RNIC. The primary challenges in implementing CCs on RNICs are the limited memory size and timer constraints [59]. PrioPlus, as an enhancement mechanism for CC, introduces only nine variables, requiring a total of 13 bytes – just one-fifth of the approximately 60 bytes used by typical CC implementations [62]. Additionally, a PrioPlus flow requires only one extra timer for its probe with collision avoidance mechanism (§ 4.2.1).

6 Evaluations

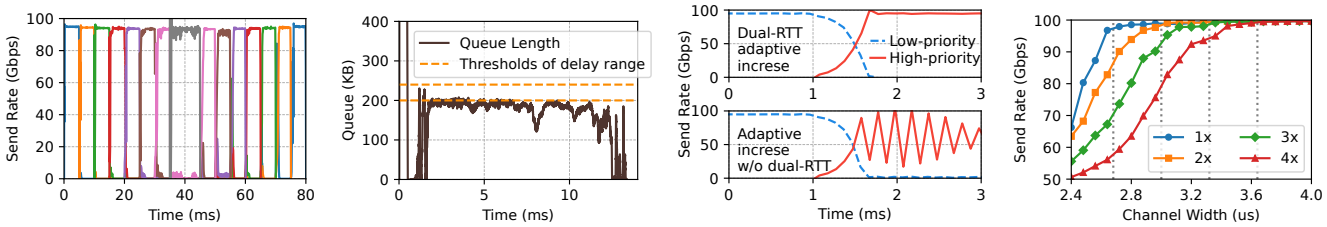
We evaluate the performance of PrioPlus using the NS3 simulator [5]. We conduct fine-grained micro-benchmarks (§ 6.1) to demonstrate that PrioPlus can support multiple priorities under heavy traffic loads and maintain delay fluctuations within the specified range during severe incast. Then, we validate the effectiveness of various designs of PrioPlus through ablation experiments and quantitative analysis in micro-benchmarks. In large-scale simulations (§ 6.2), we conduct a detailed breakdown analysis of the flow scheduling scenario to show that PrioPlus can provide prioritization comparable to physical priority. Compared with physical priority, PrioPlus provides strict priority for high-priority flows with degradation of no more than 9%. It benefits low-priority flows from 25% to 41% as it enhances CCs to boost once higher-priority traffic completes. In the coflow scheduling scenario, PrioPlus' average CCT (Coflow Completion Time) and tail CCT are improved by up to 21% and 2% compared to physical priority. In the machine learning cluster scenario, by simply assigning priorities to each model's train traffic, PrioPlus speeds up the training speed by up to 33% compared to physical priority without creating unfairness among models. *In all simulation evaluations of PrioPlus, the delay noise collected from the testbed (§ 5) is incorporated to enhance the simulation's fidelity.*

Scenarios and workloads. Our large-scale simulation experiments include three scenarios. (i) In the generic flow scheduling scenario, we generate traffic using the representative WebSearch workload [81], with traffic load at 70%. (ii) For coflow scheduling scenarios, we simulate the traffic of cluster computing (*e.g.*, Hadoop) jobs with two traffic patterns. The file request traffic, where multiple nodes send a part of a file to a single node[6], represents the data inputting process from distributed storage systems. And the coflow traffic is generated from Facebook's Hadoop trace [29, 31]. (iii) For the model training scenario, we generate training traffic of ResNet[48] and VGG [7] based on Astra-sim [79]. The detailed introduction of each scenario is deferred to § 6.2.

Topology. In the micro-benchmarks, a tree topology with multiple senders, a receiver and a single bottleneck port is used to analyze PrioPlus's behaviour on a bottleneck. Each link has a bandwidth of 100 Gbps and a latency of 3 μs to align the RTT with the typical RTT value in the data center, *i.e.*, 12 μs. In the flow scheduling scenario, we use a standard fat-tree [12] with $k = 6$ and 100 Gbps links. In this scenario, the switches' buffer/bandwidth ratio is set to 4.4 MB/Tbps, aligned with the latest switch [4]. In coflow scheduling scenario, a non-blocking fat-tree topology with 5 pods and 320 hosts is adopted. The links from hosts to edge switches are 100 Gbps, while the inter-switch links are 400 Gbps. In the machine learning cluster scenario, we adopt a spine-leaf topology similar to that used in CASSINI [77], featuring a 2:1 subscription ratio and 24 servers connected by 100Gbps links. In the last two scenarios, the switches' buffer is directly set to 32 MB to avoid insufficient buffer impacting the performance of physical priorities. The buffer of switches is shared, with PFC and dynamic threshold algorithms [28] enabled. Unless specified, all links' latency is 1us.

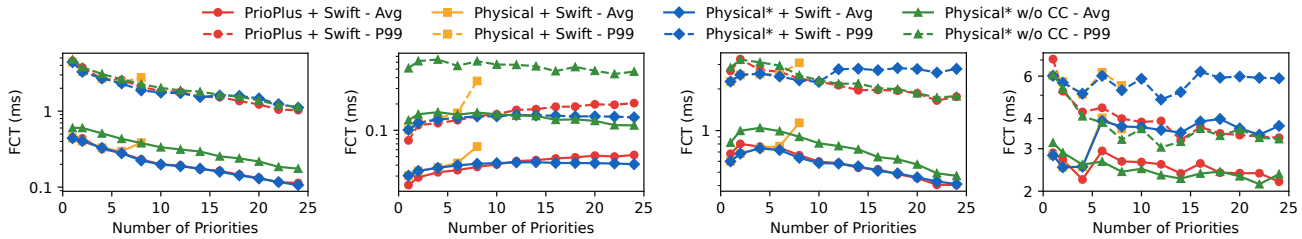
Comparisons and parameters. The primary comparison involves physical priority, D2TCP, Swift and LEDBAT. We integrate PrioPlus with Swift and LEDBAT, respectively. For D2TCP, the DDLs for traffic are set from 1.5 × to 12 × the optimal FCT, assigned in descending order of priority. For Swift and LEDBAT, the target delays are set from 32 μs to 4 μs plus base RTT, assigned in descending order of priority. The ECN threshold is consistent with the parameters used

[6]Essentially, it represents incast traffic widely seen in data center [13, 26, 102].



(a) Sending rate of 8 priorities each with 30 flows. **(b)** Queue length during an incast with 300 flows. **(c)** Ablation experiment of dual-RTT adaptive increase. **(d)** Relationship between channel width and delay noise.

Figure 10. Micro-benchmarks of PrioPlus with Swift.



(a) Total. **(b)** Small flows - $[0, 300KB)$. **(c)** Middle flows - $[300KB, 6MB)$. **(d)** Large flows - $[6MB, 30MB]$.

Figure 11. The breakdown result of the flow scheduling scenario, where *Physical** represents ideal physical priorities queues.

in the current data center [59]. All other parameters are set according to the recommended values in the original literature [52, 80, 90].

6.1 Micro-benchmarks

PrioPlus can support multiple virtual priorities with *O*1 and *O*2 both meeting. In Figure 10a, flows of eight different priorities start from low to high priority at 5 ms intervals and terminate at the same intervals. Each priority consists of 30 individual flows. It is observed that when higher-priority traffic emerges, the lower-priority traffic immediately relinquishes all bandwidth (***O*1**). Once the high-priority traffic ceases, the next-priority traffic instantly utilizes the bandwidth fully (***O*2**).

PrioPlus can manage queue fluctuation even in a heavy incast. In Figure 10b, 300 flows, each with a D_{target} of 32 μs (20 μs plus a base RTT of 12 μs) and a D_{limit} of 34.4 μs, start simultaneously, resulting an incast. After experiencing a delay exceeding D_{limit}, PrioPlus effectively moderates the overall aggressiveness by utilizing the estimated flow cardinality, maintaining the delay close to D_{target}.

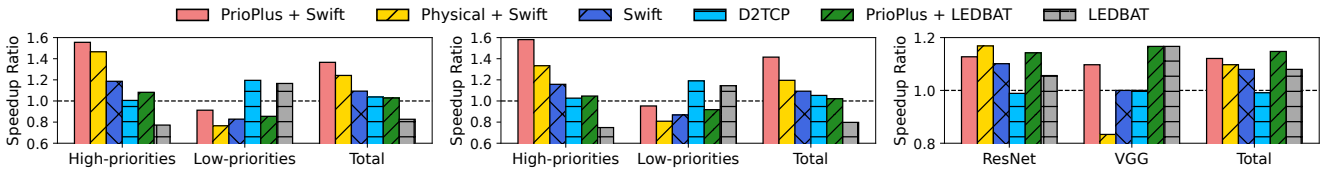
Dual-RTT adaptive increase (§ 4.2.3) can elevate delay precisely without overraction. Figure 10c depicts the process of ten high-priority flows preempting bandwidth from ten low-priority flows. In the upper subfigure, high-priority flows initiate at 1 ms and rapidly occupy all bandwidth within 1 ms through dual-RTT adaptive increase. In the lower subfigure, high-priority flows employ adaptive increase every RTT, resulting in severe overreaction.

PrioPlus robustly tolerates delay noise. Figure 10d shows the total throughput of five PrioPlus flows with the same priority passing through the same port under different scale ratios of delay noise shown in Figure 7. We use dashed lines to indicate the channel widths required to achieve more than 98% bandwidth utilization. It is observed that the required channel width increases linearly with the magnitude of the noise, demonstrating PrioPlus's robustness to noise.

6.2 Large-Scale Simulations

PrioPlus can support numerous virtual priorities to deliver performance improvement consistent with ideal physical priorities. Implementing and comparing priority-based flow scheduling algorithms is beyond the scope of this paper. Instead, in general flow scheduling scenario, we categorize all flows into groups by size, assigning higher priority to the smaller-sized flow group to approximate general flow scheduling algorithms [15, 18, 58, 70]. The metric of interest is flow completion time (FCT). Figure 11a shows the overall FCT. Physical queues can only support eight priorities due to buffer constraint (§ 2.2). When the number of physical priorities exceeds 6, the excessive buffer occupation of headroom incurs a significant increase in FCT due to frequent PFC. We implement an ideal physical priority (*Physical**), which is free from protocol constraints and headroom occupies no switch buffers. The performance of PrioPlus with Swift closely approaches ideal physical priority with Swift, with average FCT being at most 8% worse.

PrioPlus ensures the performance of high-priority (small and middle) flows (*O*1). Figure 11b and 11c show the FCT for small and medium flows. When the number of priorities is low, PrioPlus effectively reduces the switch buffer usage by suppressing low-priority flow injections, thus reducing the triggering of PFC and achieving lower FCT for small flows. As the number of priorities increases, the FCT for PrioPlus's small flows is slightly impaired due to the increased CC's target caused by accumulated channel width. For medium



(a) Coflow scenario at 40% load. **(b)** Coflow scenario at 70% load. **(c)** Model training.

Figure 12. Speedup ratio in coflow scheduling and model training scenarios.

flows, PrioPlus consistently performs close to ideal physical priorities. Overall, for small and medium flows, the average FCT of PrioPlus is at most 9% worse than ideal physical priorities, and the p99 FCT is at most 19% worse.

PrioPlus effectively utilizes bandwidth, thus benefiting low-priority (large) flows (*O*2). Figure 11d shows the FCT for large flows. When the number of priorities is fewer than six, most large flows are placed together with medium flows in the same priority, achieving a lower FCT at the expense of increased FCT for medium flows. When the number of priorities exceeds six, all large flows are assigned to lower priorities. At this point, the FCT of the physical priority with Swift significantly downgrades. This is because flows at lower physical priorities experience substantial delays when higher-priority flows are present, leading to the deceleration of Swift and suffering from slow acceleration after the higher-priority flows have ceased. PrioPlus addresses this issue through linear start (§ 4.2.2). We also compare with ideal physical priority without CC, which results in better FCT for large flows but significantly worse FCT for small and medium flows. For large flows in low priorities, PrioPlus with Swift outperforms ideal physical priority with Swift from 25% to 41% on average and from 24% to 43% on the tail.

PrioPlus can provide virtual priority for coflow scheduling and outperforms physical priority. In coflow scheduling scenario, the load ratio of coflow traffic to file request traffic is 1:1. For each file request, 20 random nodes send a piece of data to a randomly selected node. Similar to previous experiments, we categorize coflows into eight groups based on size and assign higher priorities to the smaller groups. The metric of interest is the speedup ratio of coflow completion time (CCT) compared to the baseline, which uses Swift with default parameters as the CC without priority scheduling.

Figure 12a and 12b present the average CCT speedup ratios for the high four and low four priorities, as well as the overall CCT speedup ratio. When the load is at 40% (Figure 12a), the average speedup ratio of PrioPlus with Swift is 12% higher than that of physical priority with Swift. In the higher priorities, the speedup ratio of PrioPlus is 9% higher than that of physical priority, and is 15% higher for the lower priorities. Consistent with previous findings, this is because flows in coflow scenario are almost middle and large flows. Thus, the bandwidth underutilization of physical priority with Swift impairs the performance. When the load is at 70% (Figure 12b), the trend is similar. However, the issue of underutilization is more severe for physical priority. The overall speedup of PrioPlus with Swift is 21% higher than that of physical priority. The analysis of tail speedup, performance of physical priority without CC and HPCC in coflow scenario is placed in Appendix A.2 and A.4 due to space limitations.

PrioPlus can improve the speed of model training. In the model training scenario, we simulate a medium-sized machine learning cluster following that used in CASSINI [77]. The models we employed are ResNet[48] and VGG [7], utilizing data parallelism and implementing the all-reduce operation via the ring algorithm. By assigning different priorities to the traffic of various models, the training traffic can be interleaved to accelerate the training of all models [78]. We assigned four higher priorities to four ResNet models and the other four to four VGG models. The focus metric is the speedup ratio[7] compared to the baseline, which uses Swift with default parameters as the CC without using any priority scheduling. As shown in Figure 12c, we categorize models by their type, showing their training speedups and the overall speedups. PrioPlus with Swift, compared to the baseline, accelerates the training of ResNet and VGG by 12% and 15%, respectively, with a total acceleration ratio of 13%. However, while the physical priority with Swift increases ResNet's training speed by 16%, it reduces VGG's training speed by 18%, which is 33% lower than PrioPlus, resulting in an overall acceleration ratio of just 9%.

6.3 Design Choice Verification

PrioPlus can operate under non-congestive delay. We replicate the testbed experiment (Figure 8a) in the simulation with non-congestive delay introduced at the bottleneck. The generation of non-congestive delay followed a uniform distribution within its range. The performance of PrioPlus is characterized by the Normalized FCT Gap, which is defined as the sum of $\frac{|FCT_{PrioPlus} - FCT_{Physical}|}{FCT_{Physical}}$ for all flows, where $FCT_{PrioPlus}$ is FCT of PrioPlus+Swift and $FCT_{Physical}$ is the FCT of Physical+Swift. As shown in Figure 13, with tolerable delay noise settings of 10 μs, 20 μs and 30 μs, the smallest range of non-congestive delay variations that impact performance are 14 μs, 24 μs and 32μs, respectively. This demonstrates that PrioPlus can effectively operate under non-congestive delay variations by incorporating the range of non-congestive delay variations into the tolerable delay noise setting.

[7]The training speed refers to the number of iterations in a fixed period.

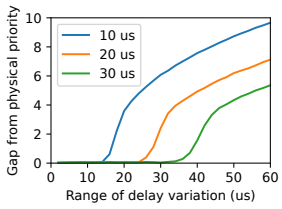
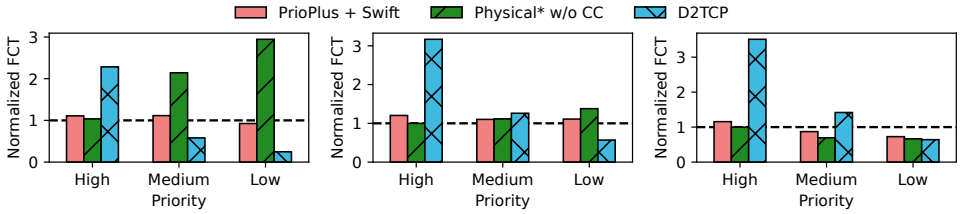


Figure 13. Performance of PrioPlus with non-congestive delay.



(a) Sub-RTT flows - $[0, 150KB)$. **(b)** Middle flows - $[150KB, 6MB)$. **(c)** Large flows - $[6MB, 30MB]$.

Figure 14. FCT breakdown for different priorities and flow sizes, normalized by FCT of Physical*+Swift.

Setting higher delay thresholds for high-priority flows does not necessarily increase their experienced delay. We conduct evaluations in the same environment as the flow scheduling scenario. Instead of dividing the WebSearch workload by flow size and assigning higher priority to smaller flows to mimic scheduling algorithms, we assign each priority a complete WebSearch workload to assess the performance of various flow sizes in different priorities. Each priority has the same load, with the total load at 50%. The result is shown in Figure 14, the performance is characterized by FCT normalized by the FCT of Physical*+Swift[8].

We classify priorities into three levels: 11 (high), 6-10 (middle), and 0-5 (low). The D_{target} for the highest priority is 60 μs, yet the average FCT for sub-RTT high-priority flows is only 20.9 μs, indicating that higher delay thresholds do not necessarily increase experienced delays.

Probe before start mechanism does not significantly raise sub-RTT flows' delay. The "probe before start" mechanism (4.2.1), which is enabled for middle and low priorities may raise concerns about introducing an additional RTT delay. However, as shown in Figure 13a, PrioPlus just raises the delay of medium-priority sun-RTT flows by 12% while reducing it by 8% for low-priority ones compared with Physical*+Swift. This is because the probe before start mechanism significantly reduces the PFC pausing in the network. In contrast, Physical*+Swift blindly injects middle and low-priority packets into the network, which may accumulate in the switch for a long time given their lower priority, making PFC trigger more easily.

The performance of Physical* w/o CC and D2TCP is consistent with the findings of previous experiments and is detailed in Appendix A.1. Among evaluated algorithms, PrioPlus+Swift consistently maintains virtual priority (slowdown less than 21% compared with Physical*+Swift) across all flow sizes and priorities.

7 Discussion

Related works. To the best of our knowledge, PrioPlus is the first algorithm to achieve strict virtual priority without switch support. There are many existing flow differentiation techniques [32, 90], achieving weighted priority by adjusting AIMD step sizes. As demonstrated in § 3.1, prior techniques that adjust AIMD step sizes can not meet objectives of strict priority. Besides, while some CCs [66, 80] effectively manage low-priority background traffic, they lack support for multiple priorities (***O1***). Most data center CCs, including sender-driven ones [10, 14, 52, 59, 69, 92, 102] and receiver-driven ones [27, 46, 60] aim for fair convergence across all flows, which has been shown to be detrimental to application performance when prioritization is required [90, 94]. Homa [70] and pHost [41] are receiver-driven CCs that incorporate strict priorities. However, the number of priorities they support depends on the number of physical priorities, which PrioPlus aims to address. Our future work will focus on integrating PrioPlus with these CCs.

Weighted virtual priority. This paper focuses on strict virtual priority, where high priority preempts all bandwidth, which is required by many performance-enhancing scheduling algorithms [11, 18, 25]. Although many CCs aim for weighted sharing of bandwidth at the flow level [33, 73], excessive low-priority flows can lead to priority inversion, breaking weighted sharing among priorities. This will be the direction of our future research.

Deployment requirements and feasibility of supporting ECN-based CCs are discussed in Appendix B.

8 Conclusion

PrioPlus is a congestion control enhancement algorithm designed to integrate with existing delay-based CC mechanisms, enabling multiple virtual priorities within each physical queue in data centers. Testbed evaluations and large-scale simulations demonstrate that PrioPlus meets the performance objectives for virtual priority. It is lightweight, requiring fewer than 100 lines of code modifications to existing CC implementations.

Acknowledgments

We would like to thank our shepherd, Andreas Schmidt, and the anonymous EuroSys '25 reviewers for all their constructive feedback and comments. This research is supported by the National Natural Science Foundation of China under Grant Numbers 62325205 and 62172204, the Key Program of Natural Science Foundation of Jiangsu under grant No. BK20243053, the Nanjing University-China Mobile Communications Group Co., Ltd. Joint Institute. Chen Tian is the corresponding author.

[8]Recall that Physical* denotes ideal physical priority which supports more than 8 lossless priorities and PFC's headroom does not occupy switch buffer (§ 6.2).

References

[1] 2013. Trident2 / BCM56850 Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56850-series

[2] 2015. HP: Quality of Service (QoS): Managing bandwidth effectively. https://support.hpe.com/techhub/eginfolib/networking/docs/switches/RA/15-18/5998-8155_ra-2620_atmg/content/ch04.html.

[3] 2016. Cisco: DSCP and Precedence Values. https://www.cisco.com/c/en/us/td/docs/switches/datacenter/nexus1000/sw/4_0/qos/configuration/guide/nexus1000v_qos/qos_6dscp_val.pdf.

[4] 2022. Tomahawk4 | BCM56990 Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56990-series

[5] 2024. Ns3 network simulator. https://www.nsnam.org/.

[6] 2024. Segmentation Offloads in the Linux Networking Stack. https://www.kernel.org/doc/Documentation/networking/segmentation-offloads.txt.

[7] 2024. VGG16 architecture. http://https://iq.opengenus.org/vgg16.

[8] 2025. Online resource of PrioPlus. https://github.com/NASA-NJU/PrioPlus.

[9] Vamsi Addanki, Wei Bai, Stefan Schmid, and Maria Apostolaki. 2024. Reverie: Low Pass Filter-Based Switch Buffer Sharing for Datacenters with RDMA and TCP Traffic. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[10] Vamsi Addanki, Oliver Michel, and Stefan Schmid. 2022. {PowerTCP}: Pushing the performance limits of datacenter networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[11] Saksham Agarwal, Shijin Rajakrishnan, Akshay Narayan, Rachit Agarwal, David Shmoys, and Amin Vahdat. 2018. Sincronia: Near-optimal network design for coflows. In *ACM Special Interest Group on Data Communication (SIGCOMM)*.

[12] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A scalable, commodity data center network architecture. In *ACM Special Interest Group on Data Communication (SIGCOMM)*.

[13] Mohammad Alizadeh and Tom Edsall. 2013. On the data path performance of leaf-spine datacenter fabrics. In *IEEE 21st Annual Symposium on High-performance Interconnects*.

[14] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data center tcp (dctcp). In *ACM Special Interest Group on Data Communication (SIGCOMM)*.

[15] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. pFabric: minimal near-optimal datacenter transport. In *ACM Special Interest Group on Data Communication (SIGCOMM)*.

[16] Nirav Atre, Hugo Sadok, and Justine Sherry. 2024. BBQ: A Fast and Scalable Integer Priority Queue for Hardware Packet Scheduling. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[17] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, et al. 2023. Empowering azure storage with {RDMA}. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[18] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. 2015. {Information-Agnostic} flow scheduling for commodity data centers. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[19] Fred Baker, Jozef Babiarz, and Kwok Ho Chan. 2006. Configuration Guidelines for DiffServ Service Classes. RFC 4594. https://doi.org/10.17487/RFC4594

[20] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. 2011. Towards predictable datacenter networks. In *ACM Special Interest Group on Data Communication (SIGCOMM)*.

[21] Amotz Bar-Noy, Magnús M Halldórsson, Guy Kortsarz, Ravit Salman, and Hadas Shachnai. 2000. Sum multicoloring of graphs. *Journal of Algorithms* 37, 2 (2000), 422–450.

[22] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. 2017. Attack of the killer microseconds. *Commun. ACM* 60, 4 (March 2017), 48–54. https://doi.org/10.1145/3015146

[23] Tommaso Bonato, Abdul Kabbani, Daniele De Sensi, Rong Pan, Yanfang Le, Costin Raiciu, Mark Handley, Timo Schneider, Nils Blach, Ahmad Ghalayini, et al. 2024. SMaRTT-REPS: Sender-based Marked Rapidly-adapting Trimmed & Timed Transport with Recycled Entropies. *arXiv preprint arXiv:2404.01630* (2024).

[24] Broadcom. 2019. BCM56980 12.8 Tb/s Multilayer Switch. https://docs.broadcom.com/doc/56980-DS.

[25] Jiamin Cao, Yu Guan, Kun Qian, Jiaqi Gao, Wencong Xiao, Jianbo Dong, Binzhang Fu, Dennis Cai, and Ennan Zhai. 2024. Crux: GPU-Efficient Communication Scheduling for Deep Learning Training. In *ACM Special Interest Group on Data Communication (SIGCOMM)*.

[26] Yanpei Chen, Rean Griffith, Junda Liu, Randy H Katz, and Anthony D Joseph. 2009. Understanding TCP incast throughput collapse in datacenter networks. In *ACM Workshop on Research on Enterprise Networking*.

[27] Inho Cho, Keon Jang, and Dongsu Han. 2017. Credit-scheduled delay-bounded congestion control for datacenters. In *ACM Special Interest Group on Data Communication (SIGCOMM)*.

[28] A.K. Choudhury and E.L. Hahne. 1998. Dynamic queue length thresholds for shared-memory packet switches. *IEEE/ACM Transactions on Networking* 6, 2 (April 1998), 130–140. https://doi.org/10.1109/90.664262 Conference Name: IEEE/ACM Transactions on Networking.

[29] Mosharaf Chowdhury and Ion Stoica. 2015. Efficient Coflow Scheduling Without Prior Knowledge. In *ACM Special Interest Group on Data Communication (SIGCOMM)*.

[30] Mosharaf Chowdhury, Ion Stoica, M Chowdhury, et al. 2012. Coflow: An application layer abstraction for cluster networking. In *ACM Hotnets*.

[31] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. 2014. Efficient coflow scheduling with Varys. In *ACM Special Interest Group on Data Communication (SIGCOMM)*.

[32] Jon Crowcroft and Philippe Oechslin. 1998. Differentiated end-to-end Internet services using a weighted proportional fair sharing TCP. *SIGCOMM Comput. Commun. Rev.* 28, 3 (July 1998), 53–69. https://doi.org/10.1145/293927.293930

[33] Jon Crowcroft and Philippe Oechslin. 1998. Differentiated end-to-end Internet services using a weighted proportional fair sharing TCP. *ACM SIGCOMM Computer Communication Review* 28, 3 (1998), 53–69.

[34] Abhishek Dhamija, Balasubramanian Madhavan, Hechao Li, Jie Meng, Shrikrishna Khare, Madhavi Rao, Lawrence Brakmo, Neil Spring, Prashanth Kannan, Srikanth Sundaresan, et al. 2024. A large-scale deployment of DCTCP. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[35] Sheng Di, Derrick Kondo, and Franck Cappello. 2013. Characterizing cloud applications on a Google data center. In *IEEE International Conference on Parallel Processing (ICPP)*.

[36] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast remote memory. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[37] Wesley Eddy. 2022. Transmission Control Protocol (TCP). RFC 9293. https://doi.org/10.17487/RFC9293

[38] Linux Foundation. 2015. Data Plane Development Kit (DPDK). http://www.dpdk.org

[39] Adithya Gangidi, Rui Miao, Shengbao Zheng, Sai Jayesh Bondu, Guilherme Goes, Hany Morsy, Rohit Puri, Mohammad Riftadi, Ashmitha Jeevaraj Shetty, Jingyi Yang, et al. 2024. RDMA over Ethernet for Distributed Training at Meta Scale. In *Proceedings of the*

ACM SIGCOMM 2024 Conference. 57–70.

[40] Peter X Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network requirements for resource disaggregation. In *USENIX symposium on operating systems design and implementation (OSDI)*.

[41] Peter X Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2015. phost: Distributed near-optimal datacenter transport over commodity network fabric. In *ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT)*.

[42] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, et al. 2021. When cloud storage meets RDMA. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[43] Yixiao Gao, Yuchen Yang, Tian Chen, Jiaqi Zheng, Bing Mao, and Guihai Chen. 2018. Dcqcn+: Taming large-scale incast congestion in rdma over ethernet networks. In *IEEE International Conference on Network Protocols (ICNP)*.

[44] Prateesh Goyal, Preey Shah, Kevin Zhao, Georgios Nikolaidis, Mohammad Alizadeh, and Thomas E Anderson. 2022. Backpressure Flow Control. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[45] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. 2016. RDMA over commodity ethernet at scale. In *ACM Special Interest Group on Data Communication (SIGCOMM)*.

[46] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. 2017. Re-architecting datacenter networks and stacks for low latency and high performance. In *ACM Special Interest Group on Data Communication (SIGCOMM)*.

[47] Keqiang He, Eric Rozner, Kanak Agarwal, Yu Gu, Wes Felter, John Carter, and Aditya Akella. 2016. AC/DC TCP: Virtual congestion control enforcement for datacenter networks. In *ACM Special Interest Group on Data Communication (SIGCOMM)*.

[48] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *IEEE conference on computer vision and pattern recognition (CVPR)*.

[49] IEEE DCB. 2011. 802.1Qbb – Priority-based Flow Control. http://www.ieee802.org/1/pages/802.1bb.html..

[50] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, et al. 2024. {MegaScale}: Scaling large language model training to more than 10,000 {GPUs}. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 745–760.

[51] Nemanja Kamenica. 2020. End-to-End QoS Implementation and Operation with Nexus. https://www.ciscolive.com/c/dam/r/ciscolive/emea/docs/2020/pdf/BRKDCN-3346.pdf.

[52] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan MG Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, et al. 2020. Swift: Delay is simple and effective for congestion control in the datacenter. In *ACM Special Interest Group on Data Communication (SIGCOMM)*.

[53] Changhyun Lee, Chunjong Park, Keon Jang, Sue Moon, and Dongsu Han. 2016. Dx: Latency-based congestion control for datacenters. *IEEE/ACM Transactions on Networking* 25, 1 (2016), 335–348.

[54] Long Li, Yaqian Zhao, and Rengang Li. 2018. Enabling Work-Conserving Bandwidth Guarantees for Multi-tenant Datacenters with Network Support. In *IEEE International Conference on Computer and Communications (ICCC)*.

[55] Qiang Li, Yixiao Gao, Xiaoliang Wang, Haonan Qiu, Yanfang Le, Derui Liu, Qiao Xiang, Fei Feng, Peng Zhang, Bo Li, et al. 2023. Flor: An Open High Performance RDMA Framework Over Heterogeneous RNICs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[56] Shijing Li, Tian Lan, Moo-Ryong Ra, and Rajesh Panta. 2018. Joint scheduling and source selection for background traffic in erasure-coded storage. *IEEE Transactions on Parallel and Distributed Systems* 29, 12 (2018), 2826–2837.

[57] Wenxin Li, Sheng Chen, Keqiu Li, Heng Qi, Renhai Xu, and Song Zhang. 2020. Efficient online scheduling for coflow-aware machine learning clusters. *IEEE Transactions on Cloud Computing* 10, 4 (2020), 2564–2579.

[58] Wenxin Li, Xin He, Yuan Liu, Keqiu Li, Kai Chen, Zhao Ge, Zewei Guan, Heng Qi, Song Zhang, and Guyue Liu. 2024. Flow scheduling with imprecise knowledge. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[59] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. 2019. HPCC: High precision congestion control. In *ACM Special Interest Group on Data Communication (SIGCOMM)*.

[60] Hwijoon Lim, Jaehong Kim, Inho Cho, Keon Jang, Wei Bai, and Dongsu Han. 2023. FlexPass: A Case for Flexible Credit-based Transport for Datacenter Networks. In *European Conference on Computer Systems (EuroSys)*.

[61] Zhuotao Liu, Kai Chen, Haitao Wu, Shuihai Hu, Yih-Chun Hut, Yi Wang, and Gong Zhang. 2018. Enabling work-conserving bandwidth guarantees for multi-tenant datacenters via dynamic tenant-queue binding. In *IEEE Conference on Computer Communications (INFOCOM)*.

[62] Yuanwei Lu, Guo Chen, Bojie Li, Kun Tan, Yongqiang Xiong, Peng Cheng, Jiansong Zhang, Enhong Chen, and Thomas Moscibroda. 2018. Multi-Path transport for RDMA in datacenters. In *USENIX symposium on networked systems design and implementation (NSDI)*.

[63] Yuanwei Lu, Guo Chen, Larry Luo, Kun Tan, Yongqiang Xiong, Xiaoliang Wang, and Enhong Chen. 2017. One more queue is enough: Minimizing flow completion time with explicit priority notification. In *IEEE Conference on Computer Communications (INFOCOM)*.

[64] Shouxi Luo, Hongfang Yu, Yangming Zhao, Sheng Wang, Shui Yu, and Lemin Li. 2016. Towards practical and near-optimal coflow scheduling for data center networks. *IEEE Transactions on Parallel and Distributed Systems* 27, 11 (2016), 3366–3380.

[65] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C Evans, Steve Gribble, et al. 2019. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 399–413.

[66] Tong Meng, Neta Rozen Schiff, P Brighten Godfrey, and Michael Schapira. 2020. PCC proteus: Scavenger transport and beyond. In *ACM Special Interest Group on Data Communication (SIGCOMM)*.

[67] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, et al. 2022. From luna to solar: the evolutions of the compute-to-storage networks in Alibaba cloud. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 753–766.

[68] Radhika Mittal, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2015. Universal packet scheduling. In *ACM Workshop on Hot Topics in Networks (HotNets)*.

[69] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. 2015. TIMELY: RTT-based congestion control for the datacenter. In *ACM Special Interest Group on Data Communication (SIGCOMM)*.

[70] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. 2018. Homa: A receiver-driven low-latency transport protocol using network priorities. In *ACM Special Interest Group on Data Communication (SIGCOMM)*.

[71] Andrea Monterubbiano, Jonatan Langlet, Stefan Walzer, Gianni Antichi, Pedro Reviriego, and Salvatore Pontarelli. 2023. Lightweight Acquisition and Ranging of Flows in the Data Plane. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 7, 3 (2023), 1–24.

[72] Ali Munir, Ghufran Baig, Syed M Irteza, Ihsan A Qazi, Alex X Liu, and Fahad R Dogar. 2014. Friends, not foes: synthesizing existing transport strategies for data center networks. In *ACM Special Interest Group on Data Communication (SIGCOMM)*.

[73] Vikram Nathan, Vibhaalakshmi Sivaraman, Ravichandra Addanki, Mehrdad Khani, Prateesh Goyal, and Mohammad Alizadeh. 2019. End-to-end transport for video QoE fairness. In *ACM Special Interest Group on Data Communication (SIGCOMM)*.

[74] Jitendra Padhye, Victor Firoiu, Don Towsley, and Jim Kurose. 1998. Modeling TCP throughput: A simple model and its empirical validation. In *Proceedings of the ACM SIGCOMM'98 conference on Applications, technologies, architectures, and protocols for computer communication*. 303–314.

[75] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. 2019. A generic communication scheduler for distributed DNN training acceleration. In *ACM Symposium on Operating Systems Principles (SOSP)*.

[76] Kun Qian, Wenxue Cheng, Tong Zhang, and Fengyuan Ren. 2019. Gentle flow control: Avoiding deadlock in lossless networks. In *ACM Special Interest Group on Data Communication (SIGCOMM)*.

[77] Sudarsanan Rajasekaran, Manya Ghobadi, and Aditya Akella. 2024. CASSINI: Network-Aware Job Scheduling in Machine Learning Clusters. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[78] Sudarsanan Rajasekaran, Manya Ghobadi, Gautam Kumar, and Aditya Akella. 2022. Congestion control in machine learning clusters. In *ACM Workshop on Hot Topics in Networks (HotNets)*.

[79] Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, and Tushar Krishna. 2020. Astra-sim: Enabling sw/hw co-design exploration for distributed dl training platforms. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*.

[80] Dario Rossi, Claudio Testa, Silvio Valenti, and Luca Muscariello. 2010. LEDBAT: the new BitTorrent congestion control protocol. In *International Conference on Computer Communications and Networks (ICCCN)*.

[81] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. 2015. Inside the social network's (datacenter) network. In *ACM Special Interest Group on Data Communication (SIGCOMM)*.

[82] Mustafa Shakir, Obaid Ur Rehman, Mudassir Rahim, Nabil Alrajeh, Zahoor Ali Khan, Mahmood Ashraf Khan, Iftikhar Azim Niaz, and Nadeem Javaid. 2016. Performance optimization of priority assisted csma/ca mechanism of 802.15. 6 under saturation regime. *Sensors* 16, 9 (2016), 1421.

[83] Sanjay Shakkottai and Rayadurgam Srikant. 2002. How good are deterministic fluid models of Internet congestion control?. In *Proceedings. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, Vol. 2. IEEE, 497–505.

[84] Alan Shieh, Srikanth Kandula, Albert Greenberg, Changhoon Kim, and Bikas Saha. 2011. Sharing the data center network. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[85] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. 2016. Programmable packet scheduling at line rate. In *ACM Special Interest Group on Data Communication (SIGCOMM)*.

[86] Jungmin Son and Rajkumar Buyya. 2018. Priority-aware VM allocation and network bandwidth provisioning in software-defined networking (SDN)-enabled clouds. *IEEE Transactions on Sustainable Computing* 4, 1 (2018), 17–28.

[87] Michael Struwe and M Struwe. 2000. *Variational methods*. Vol. 991. Springer.

[88] The Working Group for WLAN Standards. [n. d.]. IEEE 802.11 Wireless Local Area Networks. https://www.ieee802.org/11/.

[89] Chen Tian, Ali Munir, Alex X Liu, Yingtong Liu, Yanzhao Li, Jiajun Sun, Fan Zhang, and Gong Zhang. 2017. Multi-tenant multi-objective bandwidth allocation in datacenters using stacked congestion control. In *IEEE Conference on Computer Communications (INFOCOM)*.

[90] Balajee Vamanan, Jahangir Hasan, and TN Vijaykumar. 2012. Deadline-aware datacenter tcp (d2tcp). In *ACM Special Interest Group on Data Communication (SIGCOMM)*.

[91] Subir Varma. 2015. *Internet congestion control*. Morgan Kaufmann.

[92] Weitao Wang, Masoud Moshref, Yuliang Li, Gautam Kumar, TS Eugene Ng, Neal Cardwell, and Nandita Dukkipati. 2023. Poseidon: Efficient, Robust, and Practical Datacenter CC via Deployable INT. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[93] Zhufan Wang, Guangyan Zhang, Yang Wang, Qinglin Yang, and Jiaji Zhu. 2019. Dayu: Fast and low-interference data recovery in very-large storage systems. In *USENIX Annual Technical Conference (ATC)*.

[94] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. 2011. Better never than late: Meeting deadlines in datacenter networks. In *ACM Special Interest Group on Data Communication (SIGCOMM)*.

[95] Ruyi Yao, Zhiyu Zhang, Gaojian Fang, Peixuan Gao, Sen Liu, Yibo Fan, Yang Xu, and H Jonathan Chao. 2023. BMW Tree: Large-scale, High-throughput and Modular PIFO Implementation using Balanced Multi-Way Sorting Tree. In *ACM Special Interest Group on Data Communication (SIGCOMM)*.

[96] Zhuolong Yu, Chuheng Hu, Jingfeng Wu, Xiao Sun, Vladimir Braverman, Mosharaf Chowdhury, Zhenhua Liu, and Xin Jin. 2021. Programmable packet scheduling with a single queue. In *ACM Special Interest Group on Data Communication (SIGCOMM)*.

[97] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy Katz. 2012. DeTail: Reducing the flow completion time tail in datacenter networks. In *ACM Special Interest Group on Data Communication (SIGCOMM)*.

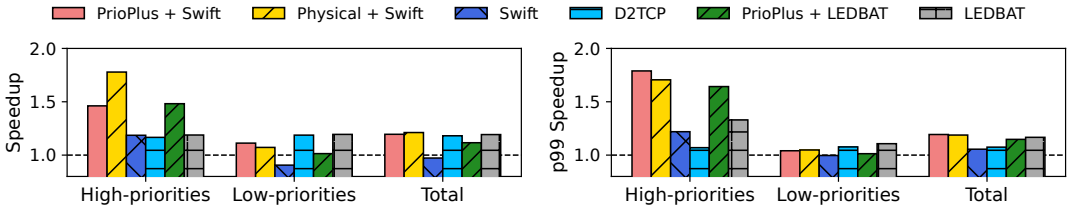
[98] Junxue Zhang, Wei Bai, and Kai Chen. 2019. Enabling ECN for datacenter networks with RTT variations. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*. 233–245.

[99] Yiwen Zhang, Gautam Kumar, Nandita Dukkipati, Xian Wu, Priyaranjan Jha, Mosharaf Chowdhury, and Amin Vahdat. 2022. Aequitas: Admission control for performance-critical rpcs in datacenters. In *ACM Special Interest Group on Data Communication (SIGCOMM)*.

[100] Zhiyu Zhang, Shili Chen, Ruyi Yao, Ruoshi Sun, Hao Mei, Hao Wang, Zixuan Chen, Gaojian Fang, Yibo Fan, Wanxin Shi, et al. 2024. vPIFO: Virtualized Packet Scheduler for Programmable Hierarchical Scheduling in High-Speed Networks. In *ACM Special Interest Group on Data Communication (SIGCOMM)*.

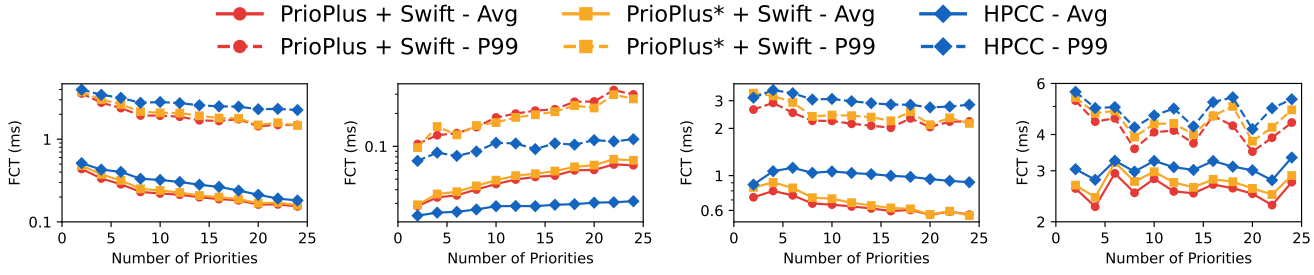
[101] Lingjun Zhu, Yifan Shen, Erci Xu, Bo Shi, Ting Fu, Shu Ma, Shuguang Chen, Zhongyu Wang, Haonan Wu, Xingyu Liao, et al. 2023. Deploying user-space {TCP} at cloud scale with {LUNA}. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 673–687.

[102] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion control for large-scale RDMA deployments. In *ACM Special Interest Group on Data Communication (SIGCOMM)*.



(a) 40% load. **(b)** 70% load.

Figure 15. Tail CCT speedup in coflow scheduling scenario.



(a) Total. **(b)** Small flows - $[0, 300KB)$. **(c)** Middle flows - $[300KB, 6MB)$. **(d)** Large flows - $[6MB, 30MB]$.

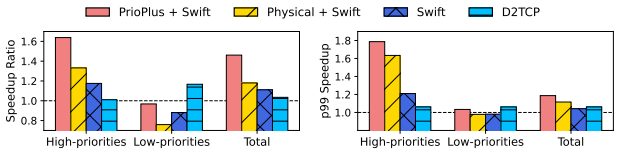
Figure 16. The breakdown result of the flow scheduling scenario, where *PrioPlus** represents ACK uses the same priority as data packets.

A Supplemental Results

A.1 Additional explanation of Figure 14

In the experiment depicted by Figure 14, Physical* w/o CC can provide ideal strict priority for high-priority traffic across all flow sizes. However, it causes severe PFC pauses at medium and low priorities as it injects medium-priority and low-priority packets without any control. The PFC pauses significantly impacting small flows at medium and low priorities. This phenomenon, where flows of around 100 bytes are most affected by PFC, aligns with the findings of flow control research [44]. Compared with Physical*+Swift, which leads to notable underutilization in larger flows in medium and low priorities (§ 6.2), flows in Physical* w/o CC are free of the underutilization caused by CC, resulting in lower FCT for larger flows in medium and low priorities.

As demonstrated in § 3.1, D2TCP fails to enforce strict virtual priorities for high priority. Thus, it incurs large slowdowns for high-priority flows compared to Physical*+Swift. Conversely, D2TCP offers lower FCT for low-priority traffic compared to PrioPlus+Swift and PrioPlus+Swift. For sub-RTT traffic, D2TCP starts transmissions without probing the network environment, achieving lower FCTs but detrimentally affecting higher-priority traffic (note that both PrioPlus+Swift and D2TCP use only one physical priority queue). For medium and large low-priority flows, D2TCP's FCT improvement stems from not ceding bandwidth to higher priorities. Considering the average FCT of all large flows, PrioPlus+Swift performs 4% better than D2TCP. D2TCP's inability to provide strict virtual priorities results in poor performance in scheduling scenarios that require strict priorities compared with PrioPlus+Swift, as shown in § 6.2.



(a) Average Speedup Ratio. **(b)** P99 Speedup Ratio.

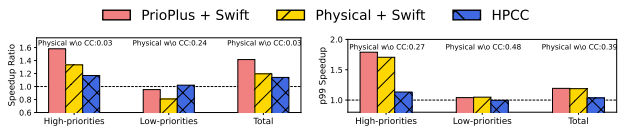
Figure 17. Speedup ratio in coflow scheduling under lossy environment at 70% load.

A.2 Tail speedup ratio in coflow scheduling scenario

Figure 15 shows the tail (99th percentile) speedup ratio in the evaluation mentioned in Section 6.2. We get the ratio by calculating the average value of the tail speedup ratio for each priority. Overall, the trend of the tail speedup ratio matches the average speedup ratio shown in Figure 12. As shown in 15a, under 40% load, the tail speedup ratio of PrioPlus with Swift has a 2% disadvantage over that of physical priority. In the higher priorities, the speedup ratio of PrioPlus with Swift is 17% less than the physical priority with Swift, but 6% higher for the lower priorities. The situation shown in Figure 15b is similar.

A.3 Supplemental Results in flow scheduling scenario

We use the same flow scheduling scenario as in Section 6.2, and compare PrioPlus + Swift with HPCC and PrioPlus* + Swift, which means in each priority, ACK uses the same priority as data packets. The result is shown in Figure 16. As shown in Figure 16a, the behavior of PrioPlus* + Swift is close to PrioPlus+ Swift with less than 10% worse performance. They both are better than HPCC. Overall, the average FCT of HPCC is at least 15% worse than PrioPlus + Swift and the p99 FCT is at least 11% worse. Figure 16b shows that HPCC tries



(a) Average Speedup Ratio. **(b)** P99 Speedup Ratio.

Figure 18. Speedup ratio in coflow scheduling at 70% load.

its best to maintain the performance of small flows. However, this causes the poor performance of medium and large flows.

A.4 Supplemental Results in coflow scheduling scenario

We use the same coflow scheduling scenario as in Section 6.2 to compare PrioPlus + Swift with HPCC and pure physical without any congestion control algorithm. Figure 18 shows that the average CCT of HPCC is 24% worse than PrioPlus + Swift, and the P99 CCT of HPCC is 15% worse. The behavior of physical without any congestion control algorithm is extremely poor, because of no control under the congested environment. The elaborate speedup ratio is marked in the figure.

A.5 Speedup ratio in coflow scheduling scenario under lossy environment

Figure 17 shows the speedup ratio under the lossy environment. The scenario is the same as the evaluation mentioned in Section 6.2, but we turn off the PFC and use IRN to deal with packet loss. The behavior of PrioPlus is nearly the same as in the lossless environment, thanks to good buffer management, which in turn reduces packet loss.

B Additional Discussion

Deployment requirement. PrioPlus requires all the traffic in a physical queue using PrioPlus for supporting virtual priority, while it does not presume the CCs employed in other priority queues.

Extend PrioPlus to support ECN-based CCs. Given that many CCs rely on ECN as their congestion signal, it would be beneficial to extend PrioPlus to support ECN-based CCs. One potential approach is to modify the ECN marking algorithm so that the marking probability varies based on flow priorities. This concept offers a promising direction while it changes switch. Hence it is not *readily deployable*. This could be a future research direction for the community.

C Analysis of Linear Start

We employ *Variational method* [87] to prove the following theorem.

Theorem C.1. *Linear start is the start strategy that incurs the least potential buffer backlog when increasing the send rate from 0 to the line rate in a given period.*

Proof. Denote the line rate as R and the total time as T. The start strategy can be described as a function from time t to rate r, *i.e.*, $r(t)$, which passes through $(0,0)$ and (T,R).

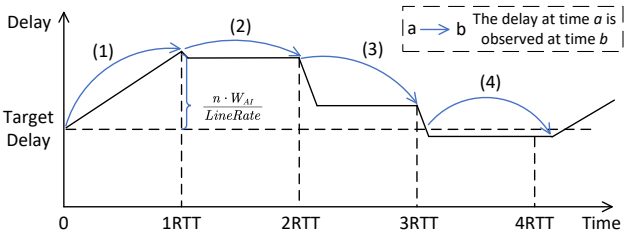


Figure 19. Delay fluctuation of synchronized Swift flows.

When the buffer begins to backlog at time a, the sender requires one RTT (denoted as τ) to detect the queueing. The amount of data excessively sent by the sender from a to $a+\tau$, *i.e.*, $\int_a^{a+\tau}[r(t)-r(a)]\mathrm{d}t$, represents the potential buffer accumulation, denoted as $b(a)$.

The start strategy $r(t)$ that minimizes potential buffer backlog is such that for all $a \in [0, T-\tau]$, the queue buildup $b(a)$ is minimized. The necessary and sufficient condition for this is that $r(t)$ minimizes the total accumulation of potential backlog B. Then we have:

$$\begin{aligned} B &= \int_0^{T-\tau} b(a) \\ &= \int_0^{T-\tau} \left\{ \int_a^{a+\tau} [r(t)-r(a)]\mathrm{d}t \right\} \mathrm{d}a \end{aligned}$$

Denote $f(r,r') = \int_a^{a+\tau}[r(t)-r(a)]\mathrm{d}t$. According to *Euler-Lagrange equation*, The r minimizes B is the r meets:

$$\frac{\partial f}{\partial r} - \frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial f}{\partial r'}\right) = 0$$

As $\frac{\partial f}{\partial r'} = 0$, we have $\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial f}{\partial r'}\right) = 0$. Therefore, we have:

$$\frac{\partial}{\partial r}\left(\int_a^{a+\tau}[r(t)-r(a)]\mathrm{d}t\right) = 0$$

This indicates that $\int_a^{a+\tau}[r(t)-r(a)]\mathrm{d}t$ is a constant for $a \in [0, T-\tau]$. Let δt represent an infinitesimally small amount of time. We have

$$\begin{aligned} &\int_a^{a+\tau}[r(t)-r(a)]\mathrm{d}t = \int_{a+\delta t}^{a+\tau+dt}[r(t)-r(a)]\mathrm{d}t \\ \Rightarrow &\int_a^{a+\tau}[r(t)-r(a)]\mathrm{d}t - \int_{a+\delta t}^{a+\tau+dt}[r(t)-r(a)]\mathrm{d}t = 0 \\ \Rightarrow &\int_a^{a+\delta t}[r(t)-r(a)]\mathrm{d}t - \int_{a+\tau}^{a+\tau+dt}[r(t)-r(a)]\mathrm{d}t = 0 \\ \Rightarrow &r(a+\delta t)-r(a) = r(a+\tau+\delta t)-r(a+\tau) \\ \Rightarrow &\frac{r(a+\delta t)-r(a)}{\delta t} = \frac{r(a+\tau+\delta t)-r(a+\tau)}{\delta t} \\ \Rightarrow &r'(a) = r'(a+\tau) \end{aligned}$$

This implies that the derivatives of $r(t)$ are equal for all $t \in [0,T]$ Since $r(t)$ passes through the points $(0,0)$ and (T,R), we arrive $r(t) = \frac{R}{T}t$. □

D Fluctuation analysis of Swift

In this section, we analyze the worst-case (*i.e.*, flows are synchronized [14]) delay fluctuations of Swift, which is related to the number of flows n, and the Swift parameters $Target, W_{AI}, \beta, max\_mdf$. Figure 19 shows the steady circle of synchronized Swift flows. Analyzing the steady cycle can begin from any point within the cycle. We choose to start at the point where the delay exactly equals the target.

From time 0 to time $1RTT$, n Swift flows increase their window size by $n \cdot W_{AI}$, resulting in an increase in delay of $\frac{n \cdot W_{AI}}{LineRate}$. At $1RTT$, as shown by arrow (1), Swift flows observe the delay at time 0, which is equal to the target delay, prompting Swift flows to stop accelerating. After time $1RTT$, Swift flows detect the delay slightly above the target, prompting the first multiplicative decrease in the cycle. Note that Swift flow decreases by $\max\left(\beta \cdot \frac{delay - Target}{Target}, max\_mdf\right)$ once per RTT if observed delay $delay$ is larger than target delay $Target$. Since the observed delay is only slightly higher than the target delay, the reduction in windows is also minimal. At time $2RTT$, one RTT after the first decrease, Swift flows observe the delay around time $1RTT$, which is around $Target + \frac{n \cdot W_{AI}}{LineRate}$, as shown by arrow (2). Therefore, Swift flow decrease by $\max\left(\frac{n \cdot \beta \cdot W_{AI}}{LineRate \cdot Target}, max\_mdf\right)$ At time $2RTT$, one RTT after the second decrease, Swift flows observe the delay around time $2RTT$, which is also around $Target + \frac{n \cdot W_{AI}}{LineRate}$ (note that the first decrease is minimal). Therefore, Swift flow decrease by around $\max\left(\frac{n \cdot \beta \cdot W_{AI}}{LineRate \cdot Target}, max\_mdf\right) \cdot Target$ again. Decrease can be repeated multiple times until the delay falls below the target. In the example shown in Figure 19, we assume that only three decreases are required. An RTT after delay falls below the target, the swift flows start to accelerate, initiating a new cycle.

The delay fluctuations of Swift flows are composed of the part above the target and the part below the target. The part above the target is $\frac{n \cdot W_{AI}}{LineRate}$, while the part below the target is evidently bounded by $\max\left(\frac{n \cdot \beta \cdot W_{AI}}{LineRate \cdot Target}, max\_mdf\right) \cdot Target$. Therefore, the fluctuation of Swift is bounded within $\frac{n \cdot W_{AI}}{LineRate} + \max\left(\frac{n \cdot \beta \cdot W_{AI}}{LineRate \cdot Target}, max\_mdf\right) \cdot Target$.