Flow Scheduling with Imprecise Knowledge

Wenxin Li[1] Xin He[1] Yuan Liu[1] Keqiu Li[1] Kai Chen[2,3] Zhao Ge[1] Zewei Guan[1]
Heng Qi[4] Song Zhang[1] Guyue Liu[5]
[1]*Tianjin Key Laboratory of Advanced Networking, Tianjin University*
[2]*Hong Kong University of Science of Technology* [3]*University of Science and Technology of China*
[4]*Dalian University of Technology* [5]*New York University Shanghai*

Abstract

Most existing data center network (DCN) flow scheduling solutions aim to minimize flow completion times (FCT). However, these solutions either require precise flow information (e.g., per-flow size), which is challenging to implement on commodity switches (e.g., pFabric [7]), or no prior flow information at all, which is at the cost of performance (e.g., PIAS [10]). In this work, we present QCLIMB, a new flow scheduling solution designed to minimize FCT by utilizing *imprecise flow information*. Our key observation is that although obtaining precise flow information can be challenging, it is possible to accurately estimate each flow's lower and upper bounds with machine learning techniques.

QCLIMB has two key parts: i) a novel scheduling algorithm that leverages the lower bounds of different flows to prioritize small flow over large flows from the beginning of transmission, rather than at later stages; and ii) an efficient out-of-order handling mechanism that addresses practical reordering issues resulting from the algorithm. We show that QCLIMB significantly outperforms PIAS (88% lower average FCT of small flows) and is surprisingly close to pFabric (around 9% gap) while not requiring any switch modifications.

1 Introduction

Flow scheduling is an effective scheme for low latency data center network (DCN) transport design [7, 10, 24, 41, 27, 37, 33]. One of the most important goals of flow scheduling is to minimize flow completion times (FCT), which is essential for many critical DCN applications, such as web search [5], key value store [2, 40, 3], and machine learning training [23]. These applications are dominated by small messages and have stringent latency requirements, as a result, even a very small delay can significantly degrade application performance [37].

There are two lines of prior work aimed at minimizing FCT, known as clairvoyant and non-clairvoyant scheduling. The clairvoyant scheduling [7, 41, 37, 24] assumes prior knowledge of precise flow size information and uses it to approximate the Shortest Remaining Processing Time (SRPT). This approach can theoretically achieve optimal performance, but is very challenging to be deployed in current DCNs, e.g., requiring too many priority queues [7] or re-factor the entire TCP/IP stack [41, 37, 24]. The non-clairvoyant scheduling [10, 11, 21, 48] requires no prior flow information and dynamically estimates flow size (e.g., based on the bytes the flow has sent [10]). While this approach is easy to implement in practice, it cannot precisely distinguish between large and small flows at the beginning, thus failing to minimize FCTs for latency sensitive short flows.

To minimize FCT and be practical, we explore a new design space that lies between existing clairvoyant and non-clairvoyant scheduling solutions. Rather than relying on precise flow information or no prior flow information at all, we ask: *Is it possible to use imprecise flow information to minimize FCT with commodity switches*?

Answering this question requires identifying useful, yet imprecise, flow information and incorporating it into flow scheduling. Some existing work [23, 42] employ machine learning (ML) techniques to estimate per-flow size but fail to get high accuracy (§2.2). Utilizing imprecise flow information is also very challenging. Simply feeding imprecise information to existing clairvoyant schedulers significantly degrades their performance [23].

We address these challenges with QCLIMB, a practical flow scheduling solution that uses imprecise flow information to minimize FCT in DCNs. QCLIMB is based on a key observation that although determining per-flow size of DCN applications can be difficult, it is possible to accurately estimate each flow's lower and upper bounds. Our experiments (§2.3) on realistic DCN workloads have shown that the actual sizes of a majority of flows ($> 99.9\%$) fall within their lower and upper bounds estimated by random forest (RF) model. Moreover, we found the actual sizes of small flows are generally close to their lower bounds, while medium and large flows may

have a larger gaps between their actual sizes and lower bounds. These findings provide an opportunity to precisely differentiate small flows from large flows based on their *lower bounds*, making it possible to prioritize small flows over large ones from the start, rather than in later stages of transmission.

Based on these findings, we develop a novel scheduling algorithm consisting of two main phases: *queue-climbing-up* and *queue-climbing-down*. Each flow is initially mapped to a priority queue based on its lower bound. During the *queue-climbing-up* phase, the flow is gradually promoted to higher-priority queues based on its remaining data size relative to the lower bound. If the flow is not yet completed after the first phase, it enters the *queue-climbing-down* phase. During this phase, QCLIMB gradually demotes the flow to lower-priority queues based on its bytes sent, and once its upper bound finishes, it is pulled directly to the lowest priority queue.

QCLIMB's algorithm is effective at prioritizing small flows over large ones for two reasons: i) small flows are close to their lower bounds and thus can finish in the first few higher priority queues during the queue-climbing-up phase; and ii) medium and large flows will be transmitted behind small flows during the queue-climbing-up phase because their lower bounds are relatively larger. They will also be penalized to the last few lower priority queues during the queue-climbing-down phase.

In addition to designing the scheduling algorithm, we must also tackle practical out-of-order (OOO) issues. This is because during the queue-climbing-up phase, the later packets of a flow can enter higher-priority queues than the earlier ones. The default TCP OOO handling mechanism considers this event as a packet loss and triggers unnecessary retransmissions. This results in serious performance degradation, especially for small flows (details in §5.2).

Addressing the OOO issue needs to overcome two practical challenges: First, how to differentiate reorderings caused by QCLIMB's scheduling algorithm from actual packet loss? Second, for packets reordered by QCLIMB, how to efficiently reorder them at the receiver side? Simply relying on the default TCP reordering mechanism will cause redundant retransmissions.

To tackle the first challenge, we present a priority-based loss detection mechanism at the receiver (§3.2.1). The idea is to leverage the fact that the packets carrying the same priority belonging to the same flow should be in order, and a gap within the same priority is identified as a loss. For the second challenge, we take a non-intrusive approach by customizing ACKs for OOO packets at the receiver to bypass the default TCP ACKing mechanism. Through this way, normal TCP ACKs will not be sent for OOO packets, and the senders have no chance to trigger redundant retransmissions (§3.2.2). Moreover, with the customized ACK, the sender can quickly retransmit the lost packet without waiting for timeout (§3.2.3).

We have implemented a QCLIMB prototype (§4), which only requires end host implementation and the built-in function (e.g., strict priority queuing) in existing commodity switches. We implement the scheduling logic and RF model inferencing as Linux kernel modules, which reside between the NIC driver and the TCP/IP stack as a shim layer. Further, OOO handling is implemented within the TCP and IP layers and requires minimal modifications to the kernel source code but does not touch the core TCP congestion control code.

We build a small-scale 25G testbed with eight servers and a Barefoot Tofino switch[1], together with large-scale simulations at 40/100G network, to evaluate the performance of QCLIMB (§5). We find that:

- Compared to PIAS [10] that requires no flow size, QCLIMB reduces the overall average FCT and the average/tail FCT of small flows by up to 49.5% and 88%/97%, respectively. It also improves the query performance by 70%∼97% in a Memcached application.
- Compared to pFabric [7] that assumes precise flow information, QCLIMB can deliver an average gap of 9% for overall performance and even show a 6.8% lower tail FCT of small flows in a PageRank workload.
- QCLIMB's design components are effective for the performance. QCLIMB is resilient to extreme cases with model-application mismatching, coexisting applications, and tiny workload. Yet, it outperforms PIAS even with two queues and low model accuracy.

2 Background and Motivation

2.1 Limitations of Existing Approaches

As shown in Table 1, existing DCN flow scheduling solutions can be classified into two categories:

Clairvoyant solutions: Clairvoyant solutions attempt to approximate the Shortest Remaining Processing Time (SRPT) scheduling based on prior knowledge of precise flow size information. In general, they can provide good performance but have significant limitations in deployability. First, pFabric [7] uses millions of priorities to implement SRPT, whereas such fine-grained priority queues require clean-slate switches and are not supported by existing commodity switches. Second, pHost [24], Homa [37] and EPN [34] use limited number of priority queues, but need to re-factor the network stack or rely on programmable switches. Finally, PDQ [28] and FastPass [41] do not use priorities but are impractical as well. For instance, PDQ [28] requires non-trivial switch modification to adjust the flow rate to implement flow preemption. FastPass [41], as a centralized scheduling design, is challenging to be deployed in a large cluster.

[1]Though the switch is P4-programmable, we use it as a commodity hardware in our evaluation.

Schemes		Requiring no switch changes or advanced hardware	Using limited number of priority queues	Retaining existing TCP/IP network stacks	Using lower bounds for flow scheduling
Clairvoyant	pFabric [7]	No	No	No	No
	PDQ [28]	No	× (priority queues are not required)	No	No
	EPN [34]	No	Yes	Yes	No
	FastPass [41]	Yes (but centralized arbiter is required)	× (priority queues are not required)	No	No
	pHost [24]	Yes	Yes	No	No
	Homa [37]	Yes	Yes	No	No
Non-Clairvoyant	PIAS [10]	Yes	Yes	Yes	No
QCLIMB		Yes	Yes	Yes	Yes

Table 1: Summary of main DCN flow scheduling schemes in literature and comparison to QCLIMB.

Non-clairvoyant solutions: Non-clairvoyant schedulers are agnostic about flow size information and generally use the idea of flow aging to estimate the pending data with the bytes a flow has already sent. For example, PIAS [10] gives the highest priority to new flows and then gradually demotes their priorities as they send more data. It is a readily-deployable solution that works with multiple priority queues available in commodity switches and is compatible with legacy TCP/IP stacks. However, PIAS has limited ability in minimizing FCT (§5).

In short, this short analysis inspires us to think about if there is a middle-point design between clairvoyant and non-clairvoyant schedulers, i.e., scheduling flows with imprecise knowledge and with commodity switches.

2.2 Imprecise Flow Information

Indeed, researchers have shown the possibility to learn flow size information from past traces using prevailing ML techniques [23, 42, 35, 30]. Nevertheless, the estimated flow sizes of ML models are often imprecise. To validate this point, we use a widespread ML technique—RF, over three workloads[2]: K-Means, PageRank, and SGD, provided by [23]. Fig. 1 plots the gap between actual and estimated flow sizes for different-scale RF models we trained (i.e., different *maximum tree depth d*). In general, a larger d leads to higher prediction accuracy. However, the gap between actual and estimated flow sizes always exists. Under $d = 10$, the mean/maximum gap can reach 856KB/296811KB, 820KB/333271KB, and 55KB/54446KB, for the K-Means, PageRank and SGD workloads, respectively. In particular, for PageRank under $d = 10$ RF model, 34% of flows have a gap of over 100KB to their estimated sizes.

Little work can utilize imprecise knowledge well. The only work, FLUX [23], directly takes imprecisely estimated flow sizes as input for clairvoyant schedulers (e.g., pFabric [7]), which, however, suffers degraded FCT performance. Indeed, we did a simulation and the results show that the average FCT of small flows achieved by pFabric with imprecise knowledge can be slowed down by up to 22.8× (appendix A). Even though the estimated flow sizes are imprecise, in what follows we demonstrate that the lower bound part is highly accurate.

[2]We did not use traditional DCN workloads like web search [5] and data mining [25] because they only provide flow size distribution. The workloads of [23] are collected from university clusters running real ML applications and contain enough information for learning.

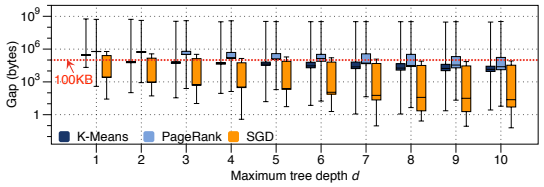


Figure 1: The gap between predicted and actual flow sizes.

2.3 Motivation: Lower Bound on Flow Size

2.3.1 Application Examples

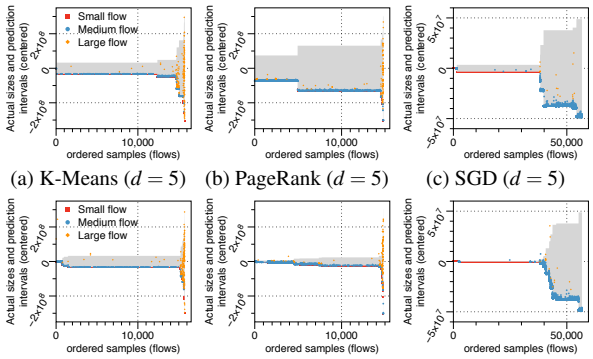
We begin by introducing a few application examples in which lower bounds on flow sizes indeed exist:

Distributed ML: Training ML models in parallel is an increasingly important workload in datacenters. During the model training, each generated flow needs to transfer at least one parameter update (e.g., a 32-bit integer). This means that the flow is at least ∼40 bytes, considering the length of various headers including TCP and IP. As a concrete example, we inspect the flow size distribution of a SGD workload [23] and find that all flows are more than 44 bytes (see §5.1).

Web Search: Large-scale web search application is another example, where a query might be sent to many aggregators and workers, each responsible for a different part of the index. From a networking perspective, it contains query traffic, delay-sensitive short messages for cluster coordination, and background traffic for response-quality-oriented massive data transfer. The smallest flow comes from the query traffic, e.g., transferring the index of at least one page between workers and aggregators, which is typically lager than 1.6KB [5].

2.3.2 Experimental Observations

The above examples only provide application-level identical lower bounds for all flows. These bounds are loose for individual flows. That said, they may be far away from the actual flow sizes, thus limiting the effect of flow scheduling. To provide a tight lower bound for each individual flow at its start, we use the RF technique [15]. Specifically, we keep the full conditional distribution of each RF tree's decisions and use a quantile regression forest (QRF) method [36] to build a prediction interval for each flow's size (see appendix B). Fig. 2 gives an in-



(a) K-Means ($d = 5$) (b) PageRank ($d = 5$) (c) SGD ($d = 5$)

(d) K-Means ($d = 10$) (e) PageRank ($d = 10$) (f) SGD ($d = 10$)

Figure 2: Relationship between actual flow sizes and prediction intervals. For better visualization, flows are ordered according to their prediction intervals.

tuitive visual feeling[3] for the prediction intervals. We have the following observations that inform our design.

Observation 1 (O1): *With RF model, each flow can have a bounded interval on its data size, and the vast majority of flows can be accurately bounded.*

A flow is bounded if its actual size falls exactly within its lower and upper bounds; otherwise, it is an out-of-bound flow. As shown in Table 2, the ratio of bounded flows accounts for 99.9055%∼100%. This result stems from the fact that QRF keeps the full conditional distribution of the estimated size; thus, the obtained prediction intervals can cover each flow with high probability.

Observation 2 (O2): *For small flows, their actual sizes are mostly close to their lower bounds.*

From Fig. 2, we see that small flows are mostly scattered around their lower bounds. The average gap of small flows' actual sizes to their lower bounds can be only 12983B, 12630B, and 213B, for K-Means, PageRank, and SGD workloads, respectively. For the root cause of this phenomenon, an intuitive conjecture is that these workloads exhibit a long-tail distribution where most flows are short. This makes an RF model have many small values in its leaf nodes, thus predicting a lower bound close to the small flow's size.

Observation 3 (O3): *Medium and large flows' lower bounds are relatively larger than those of small ones.*

Across all the cases we examined, the average lower bound of small flows is 52B∼12KB, 1002B∼1198B, and 80B∼664B, for K-Means, PageRank, and SGD workloads, respectively, For medium (large) flows, the average lower bound is 52B∼124KB (52B∼26899KB), 1002B∼2090B (1002B∼19060KB), and 255KB∼ 260KB (260KB∼1109KB) for the three workloads, respectively.

Observation 4 (O4): *For out-of-lower-bound flows, the small flows have a smaller gap to relevant lower bounds than medium and large ones.*

[3]Fig. 2 only shows two settings (i.e., the maximum tree depth $d = 5$ and 10), but our observations are condensed from all settings we tested.

Workloads		The ratio of bounded flows	The ratio of out-of-bound flows
K-Means	$d = 5$	99.9936%	0.0064%
	$d = 10$	99.9301%	0.0699%
PageRank	$d = 5$	100%	0
	$d = 10$	99.9055%	0.0945%
SGD	$d = 5$	100%	0
	$d = 10$	99.9877%	0.0123%

Table 2: The ratio of bounded and out-of-bound flows.

We gather the out-of-lower-bound flows and explore how far their actual flow sizes are from their lower bounds. We observe that this differs in different types of flows. For small flows, the average gap between their actual sizes and lower bounds is only 2.8B∼25KB, across the cases we tested. For medium and large flows, this gap can reach 68KB∼743KB and 6863KB∼77072KB, respectively.

Observation 5 (O5): *For out-of-upper-bound flows, they are primarily medium and large flows and may go beyond their relevant upper bounds a lot.*

We barely see any small flows going beyond their upper bounds in our experiments. The out-of-upper-bound flows are dominated by medium and large ones; yet, their actual flow sizes have a substantial gap to relevant upper bounds, i.e., 54KB∼2322KB and 262KB∼43252KB for medium and large flows, respectively.

Observations **O4** and **O5** can be intuitively caused by different densities of flow size between small and medium/large flows. Compared to small flows, medium/large flows have a more sparse distribution of flow size value; thus, if being out-of-bound, they are more likely to have a larger gap to their bounds.

Discussion: The observations above might not manifest for applications without long-tail flow size distributions, e.g., local file system [22] and archival data [43]. This is expected because regular RF shows high predictability for majority class samples, while small flows in these applications are minority classes. A possible way would be to modify the weighting strategy in regular RF [45] to bias small-value samples and consequently make better predictions for small flows than medium/large ones.

3 QCLIMB Design

3.1 Lower-bound-based scheduling

The above findings offer an opportunity to differentiate small flows from large flows precisely, based on their *lower bounds*. This makes it possible to prioritize small flows over large ones from the start, rather than in later stages of transmission. Based on this insight, we develop a novel flow scheduling algorithm consisting of two main phases: *queue-climbing-up* and *queue-climbing-down*.

Phase 1: Queue-Climbing-Up. The observations **O1** in §2.3 indicate that the lower bound of each flow is highly

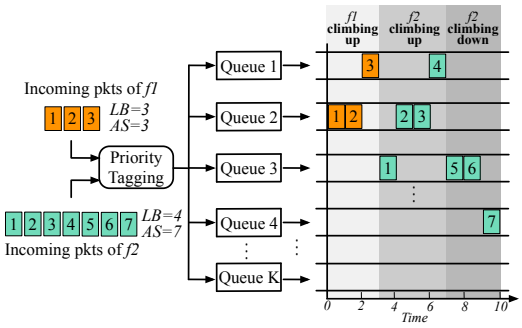


Figure 3: A queue-climbing-up&-down example with two flows: *f1* and *f2*. AS: actual size; LB: lower bound.

credible. That said, each flow's actual size is, with a high probability, larger than its lower bound. So, QCLIMB hypothetically believes that each flow's size equals its lower bound. In particular, QCLIMB lets each flow enter a lower-bound-matched initial priority queue first and then gradually promotes it from this initial queue to the higher-priority ones based on the remaining bytes to its lower bound (see Fig. 3). In this phase, flows with smaller lower bounds but the same bytes sent or larger bytes sent but the same lower bounds are likely to have higher priority.

Phase 2: Queue-Climbing-Down. The hypothetical judgment made above can be wrong for the flow not yet completed after its lower bound part finishes. QCLIMB thus takes an additional queue-climbing-down phase as a remedy. Specifically, QCLIMB pulls back the flow to its initially entered priority queue and punishes it by demoting it to lower-priority queues gradually based on its bytes sent. Moreover, if the bytes sent exceed the upper bound, QCLIMB takes a greater punishment by dragging the flow down to the lowest priority queue.

Why this works. Our scheduling algorithm is able to effectively prioritize small flows over large ones for the following reasons. *First*, if a flow is a small one, its size is near the lower bound (**O2**); thus, it can finish with the first few higher priorities. *Second*, small flows can keep relatively higher priorities than medium and large flows during the queue-climbing-up phase since their lower bounds are somewhat smaller (**O3**). *Third*, despite very few small out-of-lower-bound flows, they will not enter a priority queue lower than the one mapping their actual sizes because of their small gap to lower bounds (**O4**) and the relatively large flow size range of each priority queue. *Fourth*, a medium or large flow will eventually be penalized to lower-priority queues or even the lowest priority queue (**O5**).

In comparison to the non-clairvoyant solutions such as PIAS [10], which demotes flows as they transmit more data, medium/large flows can only be detected and separated into low-priority queues after they coexist with short flows in higher-priority queues for a short period. In contrast, QCLIMB separates flows into different priority queues once they start, using their lower bounds.

Putting it all together: QCLIMB adopts multiple priority queues available in commodity switches. Packets carrying different priorities will enter into different priority queues. Packets in different queues are scheduled with strict priority, while packets in the same queue follow FIFO scheduling. Packet priority tagging (appendix C) is distributed at each end-host, which is triggered whenever a new packet p arrives. It first gets the bytes sent B, lower bound L, and upper bound U for the parent flow of the arriving packet p. It then relies on the following three steps working cooperatively.

1. **Priority Promoting:** If $B < L$, we invoke *get_priority* for determining packet p's priority, which scans queue thresholds bottom-up and returns the first priority having a threshold smaller or equal to $L - B$. Meaning, a flow is initially mapped to a priority queue according to its lower bound. Then, it climbs up from this initial queue to higher-priority queues gradually based on the remaining bytes to its lower bound.
2. **Delay Demoting:** A flow may only contain minor errors: its lower/upper bounds map to the same priority. Such flow can only remain a small amount of data after the first phase. So, we take a delay demoting approach here, i.e., we keep each flow's priority (mostly the highest one) for a short while to transmit a slack size S more data to hope it can finish.
3. **Priority Demoting:** At this point, if a flow is not yet completed, we will change the strategy and gradually demote its packets to lower-priority queues according to its bytes sent. If the flow is not end even after the upper bound portion finishes, we move it to the lowest priority queue directly.

Choice of Slack Size S: A larger slack size S may allow more flows to complete before switching to lower-priority queues while leading more large flows to coexist with small flows. The optimal value of S in terms of average FCT is closely related to the gap between the actual sizes and lower bounds of flows, and varies under different traffic patterns and different RF models. However, as mentioned earlier, almost all small flows are close to their lower bounds and can finish within the queue-climbing-up phase, making QCLIMB beneficial under a wide range of S (see §5.2).

3.2 Out-of-Order Handling

In addition to designing the scheduling algorithm, we must also tackle practical out-of-order (OOO) issues. This is because during the queue-climbing-up phase, the later packets of a flow can enter higher-priority queues than the earlier ones. As a result, the subsequent packets of a flow may arrive at the receiver before previous ones, i.e., out-of-order (OOO). The default TCP OOO

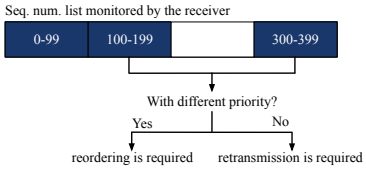


Figure 4: Priority-based loss detection.

handling may treat this event as loss, thus triggering unnecessary retransmissions and significantly hurting the performance, especially for the tail FCT of small flows (see Fig. 15 in §5.2). Addressing the OOO issue needs to overcome two practical challenges: (i) how to differentiate reorderings caused by QCLIMB's scheduling algorithm from actual packet loss? and (ii) for packets reordered by QCLIMB, how to efficiently reorder them at the receiver side?

3.2.1 Loss detection

To detect if an OOO event is due to loss, QCLIMB adopts a priority-based loss detection mechanism at the receiver. This mechanism leverages the fact that for each flow, the packets carrying the same priority traverse the same priority queue in each switch along the transmission path[4] and should be in-order. A gap within the same priority is assumed to be a loss. For a better intuition of this point, we consider a concrete example in Fig. 4. As we can see, there appears a sequence number gap. We check if the left-hand packet of this gap has different priority as the right-hand packet. If yes, the packet may be in-flight, and reordering is required (§3.2.2); Otherwise, the relevant packet must be lost, and the retransmission logic will be invoked (§3.2.3). Note that because a flow may transmit multiple packets using the same priority, there could be a corner case. More specifically, when the gap occurs at the boundary of a priority, it will not be identified as a loss with the above mechanism, but it is still likely to be a loss. In this paper, we leave such loss detection to timeout.

3.2.2 Packet reordering

Packet reordering is conducted by separating OOO packets into slow path, as detailed below. Starting from the bottom to up, packets arrive at the NIC and are read into the kernel as `skbs`. Once a packet is copied to `skb`, it will be pushed up to the TCP layer, where QCLIMB checks if it is in-ordered. If yes, this packet will be sent along the fast path that directly connects to the application layer receiver buffer. Otherwise, this packet will be identified as an OOO packet. As such, QCLIMB will send this packet along the slow path and store it in an OOO receive queue. After the missing packets arrive, QCLIMB will forward it up to the application layer receive buffer.

While the reordering is simple, the OOO event caused by QCLIMB's scheduling may mislead the default TCP stack to trigger duplicate ACKs from receiver to sender, thus leading to redundant retransmissions. To avoid this phenomenon, we let the receiver bypass the default TCP ACKing mechanism by directly sending a customized ACK to the sender on receiving any OOO packet. Meanwhile, we let the sender maintain a scoreboard to track which packets have been cumulatively and selectively acknowledged. Here, each customized ACK carries the cumulative acknowledgment (indicating its expected sequence number), a scoreboard update flag, and a SACK tag (indicating the OOO packet received). Upon receiving such ACK, the QCLIMB sender checks if this ACK contains an update flag. If yes, the sender will update its scoreboard to allow TCP to continue as usual. As such, the sender will not have a chance to trigger redundant retransmissions because the TCP's normal ACKs for OOO packets will not be sent.

3.2.3 Packet retransmission

When a loss is detected, the receiver will not discard the OOO packet, and the sender will retransmit the lost packet. More precisely, the receiver also sends a customized ACK, which has the same format as that in §3.2.2 but has a different (retransmission) flag. Upon receiving such an ACK, the QCLIMB sender checks if this ACK contains a retransmission flag. If yes, it enters loss recovery mode, where the sender retransmits the first packet that corresponds to the cumulative acknowledgment value. Any subsequent packet will be retransmitted if the sender receives another ACK carrying a higher cumulative acknowledgment value and a retransmission signal. Through this way, we can quickly retransmit the lost packets without waiting for the timeout.

Note that to guarantee the delivery of all ACKs, we transmit them at the highest priority. Moreover, we also give the highest priority to retransmitted packets because we want to fill the gap as soon as possible to minimize the resequencing buffer's memory footprint.

4 Implementation

We have implemented a QCLIMB prototype under the Linux 4.15 kernel, as detailed below.

4.1 Sender

Flow monitoring: We implement this module in Linux kernel space and integrate it with the RF inferencing and packet tagging modules. It collects flow information using the Netfilter hook [1]. For each new flow, we collect its start time, flow gap (time since the end of the previous flow), flow sizes for the last 5 flows, and TCP 5-tuple (i.e., source/destination IPs and ports, protocol ID) and feed them into the RF inference module. During the flow lifetime, we also record its bytes sent to guide the packet tagging. For each finished flow, we append its flow size to the features and transfer the collected information to

[4]We consider single-path routing for each flow, i.e., ECMP [29].

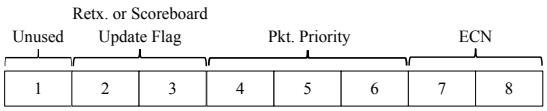


Figure 5: The TOS field format used in QCLIMB.

RF training using the `netlink` channel.

RF training: This module is in user-space. It receives flow information and performs RF training. We transfer the trained RF model to the RF inferencing module, again using the `netlink` channel.

RF inferencing: We implement this module in the kernel space to avoid frequent cross-space communication between packet tagging and RF inferencing. Specifically, we use C to implement the RF model in the kernel. This module performs online inference to obtain the lower/upper bounds for each flow and passes the inferred information to the packet tagging module for flow scheduling.

Packet tagging: This module enforces the scheduling policies of QCLIMB algorithms by marking packets with priorities at end hosts. We implement it as a Linux kernel module, locating between TCP/IP stack and Linux TC. For each outgoing packet, this module has three key operations. *First*, it maintains a hash-based flow table that stores the 5-tuple and bytes sent. *Second*, it identifies the flow the packet belonging to and updates the bytes sent of the relevant entry in the flow table. *Third*, based on the lower/upper bounds and bytes sent of the flow, it calculates a priority for this packet and invokes the `ipv4_change_dsfield` function to tag the priority into this packet's IP header using three bits of the TOS field (as shown in Fig. 5). Note that three bits can represent 8 priorities at most, which can match the number of priority queues in most commodity switches.

Packet retransmission: This module is mainly responsible for retransmitting the lost packet identified by the priority-based loss detection mechanism. Specifically, we have two operations. First, using some newly added codes in the `tcp_rcv_established` function, we check if an arriving ACK contains a flag (the 2nd and 3rd bit in TOS field). If no, the TCP/IP stack continues as usual. Otherwise, we go to the second operation, which first further checks the value of this flag. If the flag equals to 01, retransmission is needed; if it is 10, only scoreboard update is required. For retransmission, we get the sequence number of the lost packet according to the ACK's `ack_seq` field and then invokes the `tcp_transmit_skb` function to retransmit the lost packet.

Rate control: Since our implementation does not touch the congestion control code, QCLIMB is compatible with any TCP-like congestion control implementations (e.g., TCP, DCTCP). Note that when a loss is detected by the priority-based detection mechanism (§3.2.1), we halve the window size of the relevant flow to avoid injecting too many traffic in the network.

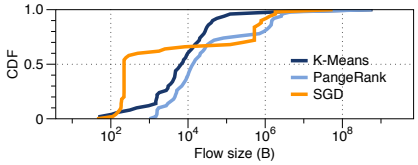


Figure 6: Workloads used for evaluation.

4.2 Receiver

Loss detection: This module contains three operations to enforce the loss detection mechanism described in §3.2.1. First, as packets are read in off the wire and converted to `skbs`, the packet priority is copied to the `skb->priority` field. Second, we compare the `skb->priority` of the first segment in the OOO receive queue with that of the last segment in the in-order TCP receive buffer to check if there is a missing gap. Third, QCLIMB adds a new `skb->flag` field in `skb_buff`. The `skb->flag` equals to 0 by default and `skb->flag=1` means that the lost packet needs retransmission. Note that `skb->flag` is in the receiver's kernel, we need to send it over the network to the relevant sender. To do this, we call a newly added function, `tcp_send_ack_qclimb`, to send a customized ACK and tag the `skb->flag` in this ACK packet. Specifically, we copy the `skb->flag` value to the 2nd&3rd bits of the TOS field in this ACK's IP header. Note here, when an OOO event is not due to loss, we will also send a customized ACK, but set the `skb->flag` value to 2 and copy it to the ACK's header to notify the sender to update the scoreboard solely.

Packet reordering: As OOO packets and in-order packets are separated into different queues, the reordering module is quite simple. Whenever there is a data packet, the receiver scans the OOO receive queue to move any in-sequence packets to the in-order receive buffer. After that, users leverage `recv` function to copy data from the TCP receive buffer to application layer receive buffer.

Remark: On the switch side, QCLIMB only needs to configure strict priority queuing (SP); if TCP/ECN transport is in use, ECN is also required. Both SP and ECN are standard features in existing commodity switches.

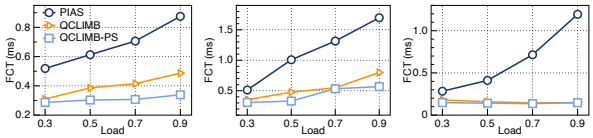
5 Evaluation

We evaluate QCLIMB through a combination of testbed experiments and large-scale simulations and show that

- QCLIMB achieves lower FCTs than PIAS (§5.1).
- QCLIMB's design components are effective (§5.2).
- QCLIMB works well in large datacenters (§5.3).

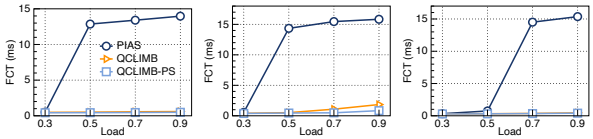
5.1 Testbed Experiments

Testbed: We build a small-scale testbed consisting of 8 servers connected to a Barefoot Tofino switch using 25Gbps links. Each server is equipped with a 16-core CPU (Intel Xeon Silver 4314@2.4GHz), 64G memory, a 25G NIC (Mellonax CX5), and installed with Ubuntu (kernel version 4.15.1). The switch runs SONIC and is



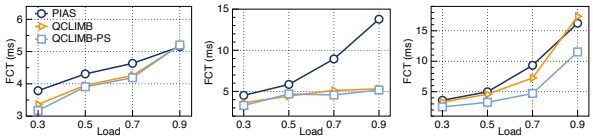
(a) K-Means (b) PageRank (c) SGD

Figure 7: Average FCT of 0-100KB small flows.



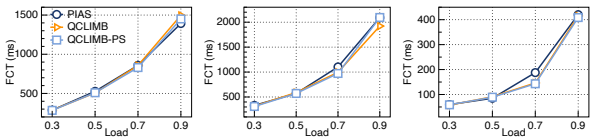
(a) K-Means (b) PageRank (c) SGD

Figure 8: 99th percentile tail FCT of 0-100KB small flows.



(a) K-Means (b) PageRank (c) SGD

Figure 9: Average FCT of 100KB-10MB medium flows.



(a) K-Means (b) PageRank (c) SGD

Figure 10: Average FCT of >10MB large flows.

configured with strict priority queuing with at most 8 queues. We use the DCTCP as the rate control.

Comparisons: We compare QCLIMB with PIAS [10] and QCLIMB-PS. We chose PIAS because it has the same practical features as QCLIMB but falls short in FCT minimization. QCLIMB-PS is a variant of QCLIMB that takes precise flow sizes as input and promotes a flow to higher-priority queues based on its remaining bytes. For OOO handling, it uses the same component as in §3.2. QCLIMB-PS is essentially a clairvoyant scheduler and is used for quantifying how far QCLIMB is from the scheme that has precise knowledge.

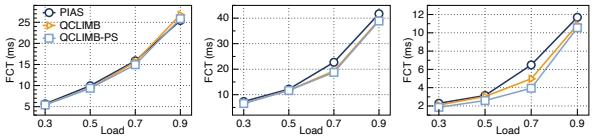
Workloads: As mentioned in §2, we mainly use three workloads provided by the paper [23]: K-Means, PageRank and SGD. Their distributions are shown in Fig. 6. All the three workloads exhibit a heavy-tailed distribution, where most flows are short and most of the traffic are dominated by a small percent of large flows. To replay these workloads in our testbed, we strictly keep the message arrival order and let message sizes follow the original testing traces. We reset the inter-arrival time of requests to match a particular network load.

Setup: For each workload, we first train an RF model (with the maximum tree depth $d = 10$ by default) for $\sim$3 minutes using 80% of the dataset and then use the trained RF model for testing with the remaining dataset to conduct our evaluation. We use 8 priority queues by default. We use the same queue thresholds as the PIAS paper [10] for all workloads. We set slack size S to 100KB by default. We set the RTO_{min} to 10ms and use a per-port buffer of 350KB at each switch.

Performance of small flows: Fig. 7 and Fig. 8 first show the average and tail FCT of small flows, respectively, for the K-Means, PageRank and SGD workloads with the network load varying from 0.3 to 0.9. We have the following two observations. First, for all workloads, QCLIMB outperforms PIAS in both the average and tail FCT of small flows. Compared to PIAS, it reduces the average/tail FCT of small flows by up to 44%/96%, 58%/96%, and 88%/97%, for the K-Means, PageRank, and SGD workloads, respectively. This is because QCLIMB promotes a flow's priority until its lower bound part finishes, while small flows are mostly scattered around their lower bounds. Therefore, unlike PIAS, QCLIMB will not let small flows suffer from the queue-climbing-down phase (see §5.2). Second, QCLIMB shows a comparable average and tail FCT of small flows with QCLIMB-PS. Particularly, compared to QCLIMB-PS at 0.7 network load, QCLIMB has a 3% gap for the small average FCT in PageRank workload and a 5% gap for the small tail FCT in SGD workload.

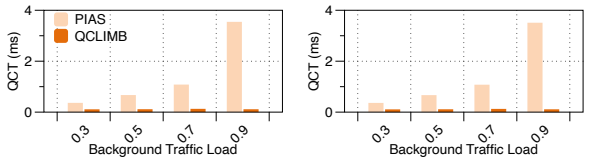
Performance of other flows: Fig. 9, Fig. 10 and Fig. 11 depict the FCT statistics for the other flows. The first observation is that QCLIMB outperforms PIAS for medium flows. Under a moderate 0.5 load, QCLIMB reduces the average FCT of medium flows over PIAS by 8%, 24%, and 8% for the K-Means, PageRank, and SGD workloads, respectively. This is expected because medium flows are more likely to leave a small amount of data after their lower bound parts finish (see Fig. 2), thus are more likely to complete in QCLIMB's delay demoting step. As the second observation, we find that QCLIMB even performs slightly better than PIAS for large flows in some cases. For instance, compared to PIAS, QCLIMB reduces the average FCT of large flows by 5% on average across all network loads in the SGD workloads. The last observation is for the average FCT of all flows. More precisely, QCLIMB shows up to 6%, 15%, and 23% reductions in overall average FCT over PIAS for the K-Means, PageRank, and SGD workloads, respectively.

Results with the Memcached application: We further build a Memcached application to evaluate QCLIMB's performance. We use one host as the client and the remaining 7 hosts as servers. Before sending each GET query (i.e., *memcached_get()*), the client will first send a SET (i.e., *memcached_set()*) request to pre-populate each server with a key-value pair. The key is flow id. The value is flow size which is randomly chosen according to a Facebook Memcached workload [8]. Once the SET request is completed, the client immediately sends a GET query to each server to get the value just populated by the SET request. We repeat this process

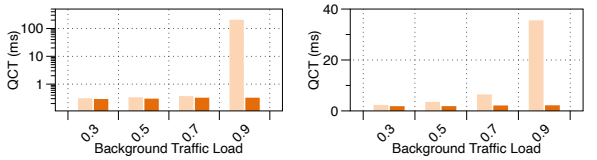


(a) K-Means (b) PageRank (c) SGD

Figure 11: Average FCT of all flows.



(a) Overall average QCT (b) Avg. QCT of <100KB queries

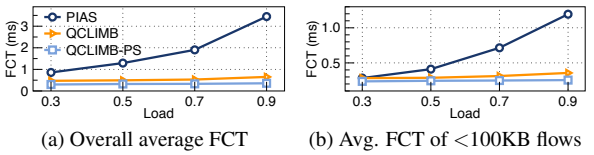


(c) Tail QCT of <100KB queries (d) Avg. QCT of >100KB queries

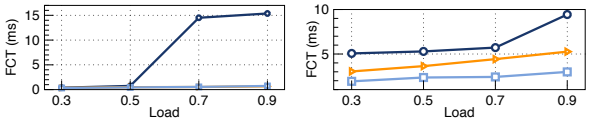
Figure 12: QCT statistics across different query sizes.

roughly 140000 times and thus generate around 140000 GET queries. A GET query completes only when the client receives all servers' responses. We only consider GET queries and measure the query completion time (QCT) as the application performance metric. We also inject background traffic, which is a mix of small and large flows generated according to a Web Search workload [5]. The background traffic load varies from 0.3 to 0.9. We calculate each query's size as the sum of its individual flows' size. Fig. 12 compares the QCT of QCLIMB with that of PIAS, across different query sizes and different background traffic loads. Compared to PIAS, QCLIMB reduces the overall average QCT by 70%∼97%, the average/tail QCT of 0-100KB small queries by 70%∼97%/6%∼99.8%, and the average QCT of >100KB queries by 21%∼94%. Note that QCLIMB's QCT does not increase with background traffic load. This is because background traffic is transmitted with the lowest priority, while all queries' traffic is in flows less than 100MB and will not overlap with background traffic in the same queue under QCLIMB's scheduling logic.

Tiny workload: We further evaluate QCLIMB in a Tensorflow workload from the Flux dataset [23]. This workload contains a significant proportion (i.e., 77%) of very tiny flows (i.e., with less than 100B) and thus is more likely to cause congestion than the other workloads. Despite this, we can observe from Fig. 13 that QCLIMB still shows superior performance. Compared to PIAS, it reduces the overall average FCT by up to 81.1%. For the average/tail FCT of small flows, the improvement of QCLIMB over PIAS is up to 70%/96.4%. The reductions of QCLIMB in small flows do not penalize other flows. In fact, QCLIMB even reduces the average FCT of >100KB flows by 22.7%∼44.2%, compared to PIAS.



(a) Overall average FCT (b) Avg. FCT of <100KB flows



(c) Tail FCT of <100KB flows (d) Avg. FCT of >100KB flows

Figure 13: FCT statistics across different flow sizes under Tensorflow workload.

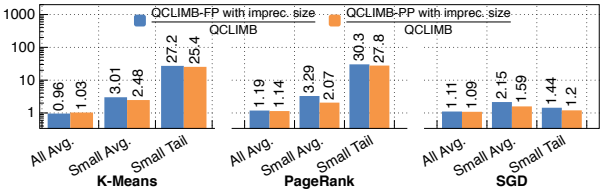


Figure 14: Effectiveness of error-tolerant scheduling.

5.2 QCLIMB Deep Dive

5.2.1 Effectiveness of the design components

Effect of lower-bound-based scheduling: We first check the effectiveness of QCLIMB's scheduling design. We construct two baselines, both of which directly use the estimated flow sizes for scheduling. The first one, QCLIMB-FP, determines a flow's priority only once by comparing its estimated flow size with queue thresholds. The second one, QCLIMB-PP, has two more steps than QCLIMB-FP. More precisely, it gradually promotes a flow's priority according to its remaining bytes to the estimated size. Afterward, if the flow is not yet ended and the sent bytes exceeds the estimated size, it directly pulls this flow to the lowest priority queue. Fig. 14 shows the results for different workloads under load 0.5. We observe that compared to QCLIMB-FP and QCLIMB-PP, respectively, QCLIMB brings up to 16% and 12% improvements in overall average FCT, and especially, it brings up to 70%/97% and 60%/96% improvements in average/tail FCT of small flows. Since QCLIMB is at most 1.63× slower than QCLIMB-PS for the average FCT of small flows (see 0.7 load in Fig. 7a), the 3.29× speedup of QCLIMB over QCLIMB-PP means that QCLIMB reduces the impact of prediction size errors on small flows by $\frac{3.29-1.63}{3.29-1} = 72\%$.

Effect of priority-based loss recovery: To quantify the effectiveness of our priority-based loss recovery mechanism, we construct a variant of QCLIMB, which abandons the proposed OOO design (§3.2) and uses the default TCP loss recovery mechanism. Fig. 15 compares the FCT performance achieved by this variant and QCLIMB for the K-Means, PageRank and SGD workloads under load 0.7. As we can see from this figure, QCLIMB outperforms the constructed variant significantly, with the overall average FCT and the average/tail

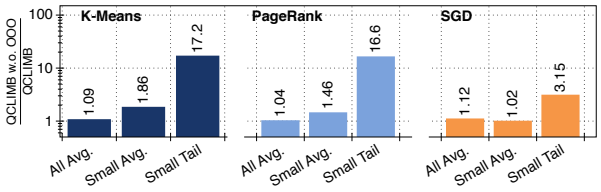
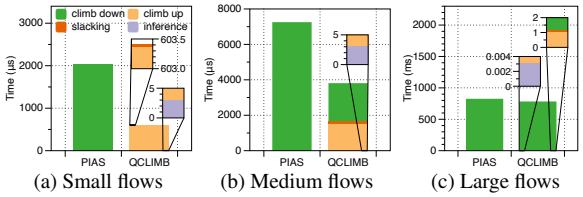


Figure 15: Effectiveness of priority-based loss recovery.



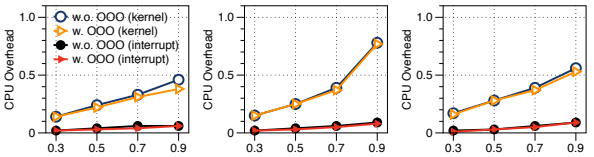
(a) Small flows (b) Medium flows (c) Large flows

Figure 16: Performance breakdown.

FCT of small flows being reduced by up to 1.12× and 1.86×/17.2×, respectively. Meaning, the priority-based loss recovery design brings up to 11% and 46%/94% improvements for the two flow types, respectively.

Performance breakdown: QCLIMB differs from PIAS because it has an additional scheduling step—queue-climbing-up, making it complete small flows with the first few higher-priority queues while separating medium/large flows into relatively low-priority queues. To confirm this, we conduct a performance breakdown evaluation for QCLIMB and PIAS under the PageRank workload at load 0.7. As we can see from Fig. 16, though QCLIMB has more processing steps for flow scheduling, it delivers significantly lower FCTs than PIAS. We further observe that for small flows in QCLIMB, only 0.52% of the FCT is spent for RF model inference, and the queue-climb-up step accounts for 99.47%, the remaining 0.01% is for the slacking step. By contrast, PIAS only contains a queue-climbing-down step and will keep demoting small flows to lower-priority queues, thus delivering poor performance. Medium and large flows in QCLIMB first go through model inference step and then climb up to higher-priority queues. They will stay in the highest-priority queue for short while using the slacking step, and finally will be penalized to lower-priority queues using the queue-climbing-down step. Owing to the queue-climbing-up step, QCLIMB achieves lower FCTs of medium/large flows than PIAS. Note that across all flow types, the model inference time is ∼3.1 μs, which is negligible as compared to the FCT of flows (603μs∼784 ms). This means QCLIMB can quickly obtain the lower/upper bounds for each flow and has the potential to apply to high-speed datacenter networks.

OOO handling Overhead: We conduct a testbed evaluation and consider two types of OOO handling overheads at receivers: the average CPU utilization (including software interrupts) and the OOO buffer space. Fig. 17 first compares the CPU overheads of QCLIMB with that of a QCLIMB variant using default TCP OOO. We observe that QCLIMB incurs up to 17.4% less CPU overhead



(a) K-Means (b) PageRank (c) SGD

Figure 17: CPU overhead and software interrupts for kernel packet processing.

Load	K-Means		PageRank		SGD	
	QCLIMB (TCP OOO)	QCLIMB	QCLIMB (TCP OOO)	QCLIMB	QCLIMB (TCP OOO)	QCLIMB
0.3	1338.70	591.66	1204.25	604.39	1793.06	936.81
0.5	2776.05	1548.54	1924.27	769.42	2006.02	1812.35
0.7	5752.96	4979.13	3133.81	2620.44	2316.89	1888.72
0.9	8077.62	6096.89	3436.34	2787.49	3333.34	2016.14

Table 3: Buffer length (bytes) for storing OOO packets.

than the baseline. Yet, the software interrupts only occupy 6.2%∼17% of the total CPU overhead. Table 3 further shows the buffer length for OOO packet storing. We observe that QCLIMB incurs 194B∼623B less OOO buffer than the default TCP OOO, equaling a 10%∼60% reduction. The reason why QCLIMB incurs less CPU overhead and OOO buffer space is that QCLIMB can quickly detect the loss and retransmit the lost packets to fill the gap in the receiver's sequence number list. Thus, OOO packets do not need to stay too long in the OOO buffer and can be quickly removed from this buffer.

5.2.2 Never-seen-before flow handling

One can either use QCLIMB or simply fall-back to PIAS for handling never-seen-before flows.

Using QCLIMB: Using QCLIMB for never-seen-before flow scheduling forms a model-application mismatching scenario. To test QCLIMB in this case, we evaluate the performance of the PageRank workload at 0.7 network load while using the RF models trained based on the K-Means and SGD workloads. Fig. 18 shows the FCT statistics. We observe that when using the SGD model for the PageRank workload, the average FCT of all flows and small flows can be prolonged by 3.63× and 2.33×, respectively. This is expected because the PageRank and SGD workloads have very different flow size distributions (see Fig. 6). By contrast, when using the RF model derived from a workload having a similar flow size distribution, the FCTs will not be affected too much. For example, the average FCT of small flows is slowed down by only 0.5% when using the K-Means RF model for the PageRank workload.

Using PIAS: Using PIAS for never-seen-before flows leads to QCLIMB flows coexisting with PIAS ones. To show how these flows affect each other, we conduct an experiment where PIAS traffic uses the WebSearch workload [5] at load 0.3, and QCLIMB traffic is the PageRank workload at load 0.5. We compare this mixing solution with a baseline that uses PIAS for all traffic. Fig. 19 shows the results. We see that for both workloads,

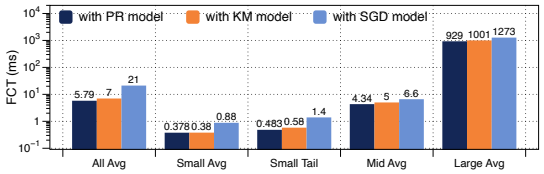


Figure 18: PageRank workload with mismatched models.

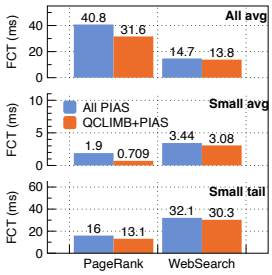


Figure 19: Mixing PIAS with QCLIMB.

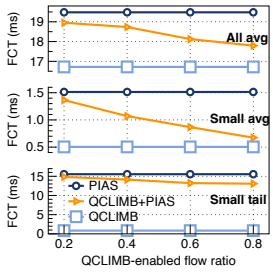


Figure 20: Impact of QCLIMB flow ratio.

the mixing solution achieves lower FCTs than the baseline. Particularly, the average FCT of small flows can be reduced by 62.7% and 10.1% for the PageRank and WebSearch workloads, respectively. This implies that mixing QCLIMB traffic in the network helps improve the performance of non-QCLIMB traffic. Moreover, as the PIAS flows does not use QCLIMB, the reduction of their FCTs under the mixing solution is less than that of QCLIMB-enabled PageRank flows.

Varying QCLIMB-enabled flow ratio: We proceed to investigate the impact of the QCLIMB-enabled flow ratio. We use PageRank workload at network load 0.7 and divide the traffic into two parts. The first part is non-QCLIMB flows that use PIAS for scheduling, while the second is QCLIMB-enabled flows. We vary the QCLIMB-enabled flow ratio from 0.2 to 0.8. We compare PIAS, QCLIMB, and the QCLIMB+PIAS mixing solution. Fig. 20 depicts the results. As expected, the performance of the mixing solution is between PIAS and QCLIMB. Even when there are 20% QCLIMB flows, the mix solution delivers ∼10% lower average FCT of small flows than PIAS. We further observe that the higher the QCLIMB-enabled flow ratio, the more flows can depart quicker, thus leaving more room for non-QCLIMB flows and consequently delivering lower FCTs.

5.2.3 Sensitivity to parameter settings

Impact of number of queues: To validate the impact of the number of queues on QCLIMB, we further conduct the testbed evaluation with 2 and 4 priority queues under the PageRank workload at 0.7 load. Fig. 21 shows the results. We observe that in general, QCLIMB outperforms PIAS under all shown settings. Even with two queues, QCLIMB works well and reduces the overall average FCT and the average/tail FCT of small flows by 28.6% and 74.3%/96.3%, respectively, compared to PIAS.

Impact of model accuracy: To understand the impact of the flow size predictor, we tested QCLIMB in the PageRank workload at 0.5 load, under different RF models with varying maximum tree depth d. Fig. 15 shows the results. It is clear that the overall average FCT decreases as d grows. This means that the higher the RF model's accuracy, the better performance QCLIMB achieves. We can further find that even under $d = 5$, QCLIMB still performs significantly better than PIAS, with the overall average FCT reduced by 16.3% and the average/tail FCT of small flows by 50%/93.8%. Note that for the extreme case of $d = 1$, the RF model may predict a wide size range for each flow, thus making QCLIMB achieve a similar overall average FCT with PIAS. But the predicted lower bounds of this $d = 1$ model might help QCLIMB complete most small flows during the queue-climbing-up phase. This makes QCLIMB achieve a dramatically lower average/tail FCT of small flows than PIAS (see Fig. 22b and Fig. 22c).

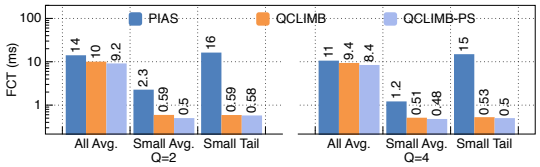
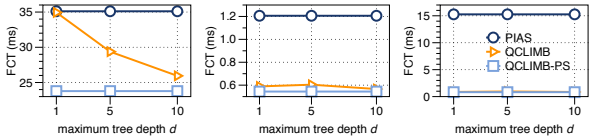


Figure 21: Impact of number of queues.



(a) Overall Avg. (b) Small Avg. (c) Small Tail

Figure 22: PageRank workload with different RF models.

Impact of slack size S: QCLIMB allows each flow to transmit a slack size S more data after its lower bound part finishes. QCLIMB outperforms PIAS under a wide range of S, and setting S to 100KB achieves the best performance for QCLIMB (appendix D).

5.3 Large-scale Simulations

Settings: In our simulations, we use the same topology as prior evaluations of PIAS [10] and pFabric [7], consisting of 144 hosts divided among 9 racks with a 2-level switching fabric; the difference is that we use 40Gbps host links and 100Gbps core links. Again, we use the K-Means, PageRank, and SGD workloads as above. We vary the load from 0.3 to 0.9.

Comparison with PIAS: Fig. 23 compares the FCT statistics of QCLIMB with that of PIAS. Compared to PIAS, QCLIMB reduces the overall average FCT by 17.4%∼47.9%, 9.1%∼38.3%, and 37.9%∼49.5%, for the K-Means, PageRank and SGD workloads, respectively. QCLIMB also achieves 1%∼30.1%/1%∼62%, 1%∼35.1%/1%∼55%, and 3%∼46.5%/2%∼42.5% lower average/tail FCT of small flows than PIAS, in the K-Means, PageRank and SGD workloads, respectively. Such good results demonstrate the effectiveness of QCLIMB's scheduling. Note

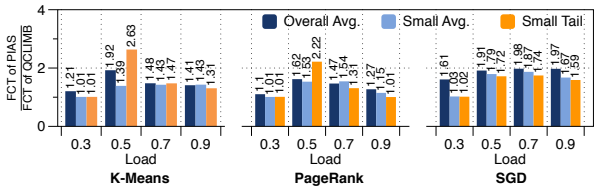
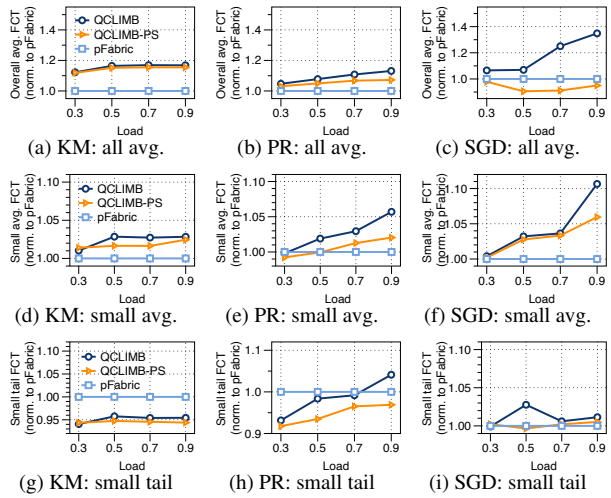


Figure 23: Comparison with PIAS in simulations.



(a) KM: all avg. (b) PR: all avg. (c) SGD: all avg.

(d) KM: small avg. (e) PR: small avg. (f) SGD: small avg.

(g) KM: small tail (h) PR: small tail (i) SGD: small tail

Figure 24: Comparison with QCLIMB-PS and pFabric in simulations (KM: K-Means; PR: PageRank).

that QCLIMB's overall improvement over PIAS in SGD workload is relatively larger than those in other workloads. The reason is that the SGD workload is more skewed. Around ∼95% and ∼74% flows in the K-Means and PageRank workloads, respectively, are smaller than 100KB, while only ∼66% flows in the SGD workload, are smaller than 100KB. In such case, large flows are more likely to coexist with short flows under PIAS. QCLIMB is less likely to be affected by such problems, as medium and large flows have relatively higher lower bounds and take less time in queue-climbing-up phase, thus exhibiting less time to coexist with small flows.

Comparison with precise-knowledge-aware schemes: We compare QCLIMB, QCLIMB-PS (§5.1) and pFabric [7]. Note that QCLIMB-PS and pFabric take precise flow sizes as scheduling input. Fig. 24 depicts the results and reveals the following findings. First, QCLIMB does not show severely worse overall performance in the K-Means and PageRank workloads. Specifically, under K-Means, QCLIMB achieves 0.4%∼1.4% and 12%∼17% higher overall average FCT than QCLIMB-PS and pFabric, respectively. For PageRank, this gap is 1.6%∼5.9% to QCLIMB-PS and 4.6%∼13.1% to pFabric. The average gap of QCLIMB to pFabric is 9% in PageRank workload. Second, for the SGD workload, QCLIMB's overall performance has a relatively large gap to QCLIMB-PS and pFabric. This is because that the SGD workload is more skewed, leading large flows in QCLIMB to be more likely to coexist with small flows. Third, QCLIMB can achieve comparable performance

Flow bins	Small flows	Medium flows	Large flows	All flows
QCLIMB	0	0	124	124 (83 by loss detection)
pFaric	10	25	414	414 (all by timeout)

Table 4: Loss events comparison (K-Means, 0.7 load).

with QCLIMB-PS for small flows.For the average FCT of small flows, QCLIMB has 1%∼2.8%, −0.2%∼5.7%, and 0.4%∼10.5%, for the K-Means, PageRank and SGD workloads, respectively. Fourth, we find that the performance gap between QCLIMB and QCLIMB-PS in the K-Means workload is smaller than that in the other workloads. This is because the RF model trained over K-Means has a higher prediction accuracy than that over the PageRank and SGD workloads. Note that QCLIMB does not severely penalize large flows. In fact, Compared to PIAS, it achieves up to 48.2% lower average FCT of large flows. Further, it has at most 15% gap to pFabric in the average FCT of large flows. For more details on the performance of large flows, please see appendix D.

Remark: Fig. 24g and Fig. 24h reveal that QCLIMB can even deliver a slightly lower tail FCT of small flows than pFabric. This is because QCLIMB incurs fewer packet loss events than pFabric. As a concrete example, Table 4 shows that QCLIMB has 70% fewer loss events than pFabric and incurs no packet loss for small and medium flows, under the K-Means workload at network load 0.7. Moreover, 83 of these 124 loss events can be quickly detected by QCLIMB's priority-based loss detection mechanism (§3.2.1), whereas the packet loss events in pFabric can only be detected by timeout.

6 Conclusion

Scheduling flows with imprecise knowledge is an important and practical problem that has been neglected by prior work in this field. QCLIMB bridges this gap with a key observation that it is possible to accurately estimate each flow's lower bounds with ML techniques. QCLIMB employs a lower-bound-based flow scheduling scheme to ensure small flows can be prioritized over medium/large ones from the beginning of transmission. It also contains fast priority-based lost recovery and packet reordering mechanisms to handle OOO issues resulting from the scheduling. We have implemented a QCLIMB prototype using all commodity hardware, and evaluated it through small-scale testbed experiments and large-scale simulations. The results show that QCLIMB is a viable solution for scheduling flows with imprecise size information.

Acknowledgments

We thank the anonymous NSDI reviewers and our shepherd Prof. Rachit Agarwal for their constructive feedback and suggestions. This work was supported in part by the NSFC Grant (62202325, 62062005), the Hong Kong RGC TRS T41-603/20-R. Keqiu Li is the corresponding author: keqiu@tju.edu.cn.

References

[1] Linux netfilter. http://www.netfilter.org/.

[2] memcached: a distributed memory object caching system. http://www.memcached.org/, 2011.

[3] Redis. https://redis.io/, 2021.

[4] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, et al. Conga: Distributed congestion-aware load balancing for datacenters. In *Proc. of ACM SIGCOMM*, 2014.

[5] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proc. of ACM SIGCOMM*, 2010.

[6] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is more: trading a little bandwidth for ultra-low latency in the data center. In *Proc. of USENIX NSDI*, 2012.

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

[8] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proc. of ACM SIGMETRICS*, 2012.

[9] Wei Bai, Kai Chen, Shuihai Hu, Kun Tan, and Yongqiang Xiong. Congestion control for high-speed extremely shallow-buffered datacenter networks. In *Proc. of the Asia-Pacific Workshop on Networking (APNet)*, 2017.

[10] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Information-agnostic flow scheduling for commodity data centers. In *Proc. of USENIX NSDI*, 2015.

[11] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Pias: Practical information-agnostic flow scheduling for commodity data centers. *IEEE/ACM Transactions on Networking*, 2017.

[12] Wei Bai, Li Chen, Kai Chen, and Haitao Wu. Enabling {ECN} in multi-service multi-queue data centers. In *Proc. of USENIX NSDI*, 2016.

[13] Wei Bai, Shuihai Hu, Kai Chen, Kun Tan, and Yongqiang Xiong. One more config is enough: Saving (dc) tcp for high-speed extremely shallow-buffered datacenters. In *Proc. of IEEE INFOCOM*, 2020.

[14] Steven Blake, David Black, Mark Carlson, Elwyn Davies, Zheng Wang, and Walter Weiss. An architecture for differentiated services. 1998.

[15] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[16] Qizhe Cai, Mina Tahmasbi Arashloo, and Rachit Agarwal. dcpim: Near-optimal proactive datacenter transport. In *Proc. of ACM SIGCOMM*, 2022.

[17] Li Chen, Kai Chen, Wei Bai, and Mohammad Alizadeh. Scheduling mix-flows in commodity datacenters with karuna. In *Proc. of ACM SIGCOMM*, 2016.

[18] Li Chen, Shuihai Hu, Kai Chen, Haitao Wu, and Danny HK Tsang. Towards minimal-delay deadline-driven data center tcp. In *Procs. of ACM HotNets*, 2013.

[19] Li Chen, Justinas Lingys, Kai Chen, and Feng Liu. Auto: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization. In *Proc. of ACM SIGCOMM*, 2018.

[20] Inho Cho, Keon Jang, and Dongsu Han. Credit-scheduled delay-bounded congestion control for datacenters. In *Proc. of ACM SIGCOMM*, 2017.

[21] Mosharaf Chowdhury and Ion Stoica. Efficient coflow scheduling without prior knowledge. In *Proc. of ACM SIGCOMM*, 2015.

[22] Allen B Downey. The structural cause of file size distributions. In *Proceedings of the 2001 ACM SIGMETRICS*, 2001.

[23] Vojislav Dukic, Sangeetha Abdu Jyothi, Bojan Karlas, Muhsen Owaida, Ce Zhang, and Ankit Singla. Is advance knowledge of flow sizes a plausible assumption? In *Proc. of USENIX NSDI*, 2019.

[24] Peter X Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. phost: Distributed near-optimal datacenter transport over commodity network fabric. In *Proc. of ACM CoNext*, 2015.

[25] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta

Sengupta. Vl2: a scalable and flexible data center network. In *Proc. of ACM SIGCOMM*, 2009.

[26] Matthew P Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert NM Watson, Andrew W Moore, Steven Hand, and Jon Crowcroft. Queues dont matter when you can {JUMP} them! In *Proc. of USENIX NSDI*, 2015.

[27] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proc. of ACM SIGCOMM*, 2017.

[28] Chi-Yao Hong, Matthew Caesar, and P Brighten Godfrey. Finishing flows quickly with preemptive scheduling. *ACM SIGCOMM Computer Communication Review*, 42(4):127–138, 2012.

[29] Christian Hopps et al. Analysis of an equal-cost multi-path algorithm. Technical report, RFC 2992, November, 2000.

[30] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[31] Shuihai Hu, Wei Bai, Gaoxiong Zeng, Zilong Wang, Baochen Qiao, Kai Chen, Kun Tan, and Yi Wang. Aeolus: A building block for proactive transport in datacenters. In *Proc. of ACM SIGCOMM*, 2020.

[32] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. Hpcc: High precision congestion control. In *Proc. of ACM SIGCOMM*, 2019.

[33] Ziyang Li, Wei Bai, Kai Chen, Dongsu Han, Yiming Zhang, Dongsheng Li, and Hongfang Yu. Rate-aware flow scheduling for commodity data center networks. In *Proc. of IEEE INFOCOM*, 2017.

[34] Yuanwei Lu, Guo Chen, Larry Luo, Kun Tan, Yongqiang Xiong, Xiaoliang Wang, and Enhong Chen. One more queue is enough: Minimizing flow completion time with explicit priority notification. In *Proc. of IEEE INFOCOM*, 2017.

[35] Larry Medsker and Lakhmi C Jain. *Recurrent neural networks: design and applications*. CRC press, 1999.

[36] Nicolai Meinshausen and Greg Ridgeway. Quantile regression forests. *Journal of Machine Learning Research*, 7(6), 2006.

[37] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proc. of ACM SIGCOMM*, 2018.

[38] A. Munir, I. A. Qazi, Z. A. Uzmi, A. Mushtaq, S. N. Ismail, M. S. Iqbal, and B. Khan. Minimizing flow completion times in data centers. In *Proc. of IEEE INFOCOM*, 2013.

[39] Ali Munir, Ghufran Baig, Syed M Irteza, Ihsan A Qazi, Alex X Liu, and Fahad R Dogar. Friends, not foes: synthesizing existing transport strategies for data center networks. In *Proc. of ACM SIGCOMM*, 2014.

[40] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, et al. The ramcloud storage system. *ACM Transactions on Computer Systems (TOCS)*, 33(3):1–55, 2015.

[41] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: a centralized" zero-queue" datacenter network. In *Proc. of ACM SIGCOMM*, 2014.

[42] Pascal Poupart, Zhitang Chen, Priyank Jaini, Fred Fung, Hengky Susanto, Yanhui Geng, Li Chen, Kai Chen, and Hao Jin. Online flow size prediction for improved network routing. In *Proc. of IEEE ICNP*, 2016.

[43] Vaidyanathan Ramaswami, Kaustubh Jain, Rittwik Jana, and Vaneet Aggarwal. Modeling heavy tails in traffic sources for network performance evaluation. In *Proceedings of ICC3*, 2014.

[44] Alon Rashelbach, Ori Rottenstreich, and Mark Silberstein. A computational approach to packet classification. In *Proc. of ACM SIGCOMM*, 2020.

[45] Marko Robnik-Šikonja. Improving random forests. In *Proceedings of Springer ECML*, 2004.

[46] Erico Vanini, Rong Pan, Mohammad Alizadeh, Parvin Taheri, and Tom Edsall. Let it flow: Resilient asymmetric load balancing with flowlet switching. In *Proc. of USENIX NSDI*, 2017.

[47] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better never than late: Meeting deadlines in datacenter networks. *ACM SIGCOMM Computer Communication Review*, 41(4):50–61, 2011.

[48] Hong Zhang, Li Chen, Bairen Yi, Kai Chen, Mosharaf Chowdhury, and Yanhui Geng. Coda: Toward automatically identifying and scheduling coflows in the dark. In *Proc. of ACM SIGCOMM*, 2016.

[49] Hong Zhang, Junxue Zhang, Wei Bai, Kai Chen, and Mosharaf Chowdhury. Resilient datacenter load balancing in the wild. In *Proc. of ACM SIGCOMM*, 2017.

[50] Junxue Zhang, Wei Bai, and Kai Chen. Enabling ecn for datacenter networks with rtt variations. In *Proc. of ACM CoNEXT*, 2019.

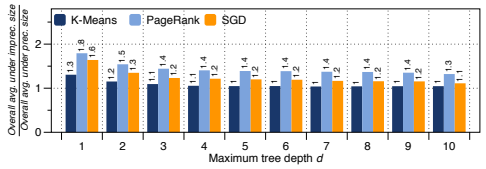
[51] Yiwen Zhang, Gautam Kumar, Nandita Dukkipati, Xian Wu, Priyaranjan Jha, Mosharaf Chowdhury, and Amin Vahdat. Aequitas: admission control for performance-critical rpcs in datacenters. In *Proceedings of ACM SIGCOMM*, 2022.

A The Performance of pFabric under Imprecise Knowledge

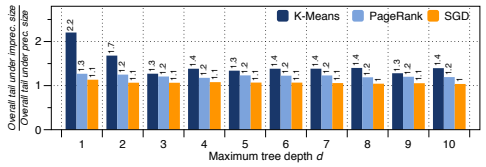
In this section, we want to know how pFabric [7] performs when scheduling flows with imprecise flow sizes. We use the same topology and workloads as §5.3. The load is 0.7. Fig. 25 shows the FCT statics across different flow sizes when using imprecise flow sizes for pFabric [7] as compared to using precise knowledge. From this figure, we observe that due to the errors in flow sizes, the FCT achieved by pFabric can be slowed down significantly. More specifically, the overall average/tail FCT achieved by pFabric with imprecise flow sizes can be increased by up to 1.8×/2.2×. Moreover, the average FCT slowdown of small flows can even reach 22.8×.

B Obtaining Prediction Intervals

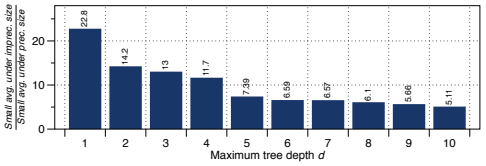
For brevity, let us consider an RF model with T trees, trained over a set of flows, i.e., $\mathcal{N} = \{1, \ldots, n\}$. Each flow i is identified by (X_i, y_i), where X_i contains the flow features and y_i is the flow size. For a new (testing) flow with its $X = x$, each tree t will drop it down to a specific leaf node l. Let $\mathcal{N}_l$ denote the set of flows falling in l. The tree t will predict a size for this flow as the weighted average over the sizes of all flows in $\mathcal{N}_l$, i.e., $y_t = \sum_{i=1}^{n} w_i(x,t) \cdot y_i$, where $w_i(x,t) = 1/|\mathcal{N}_l|$ if flow $i \in \mathcal{N}_l$ and 0 otherwise. For the entire RF, the predicted size is thus the average prediction of all trees, i.e., $y = \frac{1}{T}\sum_{t=1}^{T} y_t = \sum_{i=1}^{n} w_i(x) \cdot y_i$, where $w_i(x) = \frac{1}{T}\sum_{t=1}^{T} w_i(x,t)$ represents the weight of flow i in predicting the new flow's size. This implies that the conditional mean of y, given $X = x$, is approximated by a weighted mean over the sizes of all flows in $\mathcal{N}$, namely, $E(y|X = x) = \sum_{i=1}^{n} w_i(x) y_i$.



(a) Slowdown in overall average FCT



(b) Slowdown in overall tail FCT



(c) Slowdown in avg. FCT of small flows with PageRank

Figure 25: FCT slowdown across different flow sizes when using imprecise flow sizes for pFabric [7] as compared to using precise knowledge.

Unlike RF focusing on the conditional mean, QRF considers the conditional distribution of y under $X = x$ by defining it as the probability that for $X = x$, y is smaller than y', i.e., $F(y'|X = x) = P(y \leq y'|X = x) = E(1_{\{y \leq y'\}}|X = x)$. Just as the way approximating $E(y|X = x)$, $E(1_{\{y \leq y'\}}|X = x)$ can be approximated by the weighted mean over the observations of $1_{\{y \leq y'\}}$. This leads to $F(y'|X = x) = \sum_{i=1}^{n} w_i(x) \cdot 1_{\{y \leq y'\}}$. With this function, the α-quantile $Q_\alpha(x)$ is defined such that the probability of y being smaller than $Q_\alpha(x)$ is, for a given $X = x$, exactly equal to α, i.e., $Q_\alpha(x) = \inf\{y' : F(y'|X = x) \geq \alpha\}$. Using $Q_\alpha(x)$, one can build a prediction interval. For instance, a β $(0 \leq \beta \leq 1)$ prediction interval for the value of y is $[Q_{(1-\beta)/2}(x), Q_{(1+\beta)/2}(x)]$. Thus, $Q_{(1-\beta)/2}(x)$ and $Q_{(1+\beta)/2}(x)$ are taken as the lower and upper bounds, respectively, for the estimated size of the new flow. We set β to 1 unless otherwise specified.

C QCLIMB's Scheduling Algorithm

Algorithm 1 shows the pseudocode of QCLIMB's lower-bound-based scheduling. To understand this procedure, we use an illustrating example in Fig. 26. In this example, there are two flows (i.e., f1 and f2) and one switch port capable of transmitting 1 unit of data at each time. This port has three priority queues (i.e., Q1, Q2, and Q3), with queue thresholds being ≤ 2, $[3,5]$, and ≥ 6, respectively. We consider $S = 0$ for ease of presentation. As shown in Fig. 26(a), f1 is actually larger than f2, but its predicted size is smaller than f2's. Purely performing queue-climbing-up according to predicted size will make

Load	K-Means			PageRank			SGD		
	PIAS	QCLIMB-PS	pFabric	PIAS	QCLIMB-PS	pFabric	PIAS	QCLIMB-PS	pFabric
0.3	1.21/1.67	0.99/1.0	0.89/1.07	1.11/1.41	0.99/1.0	0.96/1.09	1.25/1.11	1.0/1.01	1.06/1.06
0.5	1.93/1.53	0.99/1.0	0.86/0.95	1.60/1.84	0.99/1.0	0.95/1.07	1.22/1.16	1.02/1.05	1.07/1.08
0.7	1.48/1.37	0.99/0.99	0.85/1.07	1.47/1.56	0.99/1.0	0.92/1.10	1.29/1.23	1.05/1.04	1.10/1.10
0.9	1.41/1.25	0.99/1.01	0.86/1.08	1.34/1.58	0.99/1.01	0.91/1.09	1.36/1.27	1.06/1.05	1.08/1.07

Table 5: Average/tail FCT of large flows achieved by different schemes in simulation (norm. to QCLIMB).

Algorithm 1 QCLIMB's Priority Tagging Algorithm

Require: An incoming packet p; A slack size S; Queue priorities $P = \{P_1, \ldots, P_k\}$ and thresholds $\alpha = \{\alpha_1, \ldots, \alpha_k\}$

1: **procedure** PRIORITYTAGGING
2: $B \leftarrow bytes\_sent\_of\_parent\_flow(p)$
3: $L \leftarrow lower\_bound\_of\_parent\_flow(p)$
4: $U \leftarrow upper\_bound\_of\_parent\_flow(p)$
5: **if** $B < L$ **then** ▷ priority promoting
6: $v \leftarrow get\_priority(L-B, P, \alpha)$
7: **else if** $B < L+S$ **then** ▷ delay demoting
8: $v \leftarrow$ the value of the highest priority P_1
9: **else if** $B < U$ **then** ▷ priority demoting
10: $v \leftarrow get\_priority(B, P, \alpha)$
11: **else**
12: $v \leftarrow$ the value of the lowest priority P_K
13: **end if**
14: Tag v into the packet p's header
15: **end procedure**

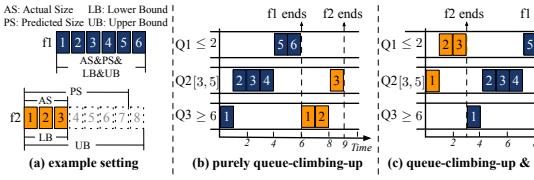


Figure 26: An example for illustrating QCLIMB's error-tolerant scheduling.

f1 complete faster than f2 and result in an average FCT of $\frac{6+9}{2} = 7.5$. Conversely, QCLIMB lets f1 and f2 initially enter Q3 and Q2, respectively, according to their lower bounds. As a result, f2 has a higher priority than f1 to transmit data and will be promoted to Q1 when its lower bound portion reduces to 2. At time 3, since its lower bound portion has finished, f2 will be moved back to the queue it initially entered, i.e., Q2, where it finishes its last packet transmission. After that, f1 starts data transmission until time 10. Thus, the average FCT is reduced to $\frac{3+9}{2} = 6$.

D Supplementary Experiment Results

Impact of slack size S [Testbed]: QCLIMB allows each flow to transmit a slack size S more data after its lower bound part finishes. To understand the impact of S, we measure the FCT statistics of QCLIMB with different S in the PageRank workload at load 0.7. As we can see from Fig. 27, with various settings of S, QCLIMB reduces the overall average FCT by 4.1%∼18.3% and the average/tail FCT of small flows by 48.6%∼62.2%/15.6%∼93.4%, compared to PIAS. We further observe that the overall average FCT of QCLIMB reduces as S increases at the beginning (e.g., before S reaches 100KB). This is reasonable because a larger S makes more flows (especially small ones) complete at the delay demoting step. However, as S keeps increasing, this step will introduce more medium and large flows to coexist with small flows in high-priority queues. That's why the FCTs of QCLIMB go up after S=100KB. In general, QCLIMB shows performance improvements over PIAS under a wide range of S, and we consider setting S to 100KB (§5.1) is a feasible choice.

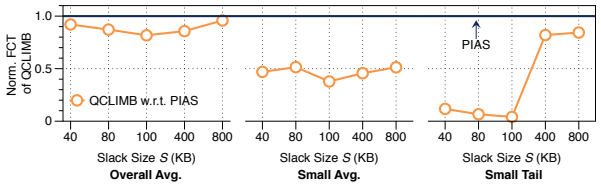
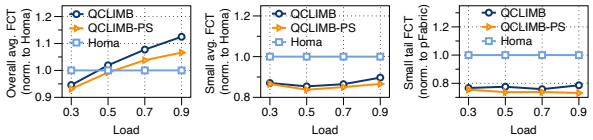


Figure 27: Sensitivity to the parameter of slack size S.

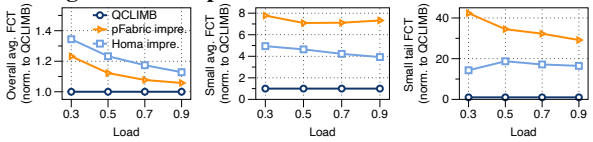
Large flow performance [Simulation]: Table 5 further summarizes the average and tail FCT of large flows achieved by different schemes in simulations. We have the following observations. First, QCLIMB achieves 10%∼48.2%/10%∼45.7% lower average/tail FCT of large flows than PIAS, across all workloads at all loads. Second, for the average FCT of large flows, QCLIMB is worse than pFabric by 14%∼4% across the K-Means and PageRank workloads, and outperforms pFabric by 6%∼10% in the SGD workload. Third, QCLIMB maintains a lower tail FCT of large flows than pFabric for most of the tested cases. In particular, it cuts the tail FCT of large flows by 10% in both PageRank and SGD workloads at 0.7 load, compared to pFabric. The reason is again that pFabric detects packet loss too late with timeout, deferring the retransmission of lost packets and prolonging tail FCTs accordingly.

Comparison with Homa [Simulation]: We compare QCLIMB with Homa [37] under the PageRank workload. The results are shown in Fig. 28. We observe that QCLIMB has up to a 12.5% gap to Homa



(a) All avg. (b) Small avg. (c) Small tail

Figure 28: Comparison with Homa in simulation.



(a) All avg. (b) Small avg. (c) Small tail

Figure 29: Comparison with pFabric and Homa using imprecise knowledge in simulation.

for the overall average FCT. This is expected because QCLIMB uses the reactive DCTCP for rate control. By contrast, Homa can proactively allocate bandwidth as "grants" to senders, who can then send scheduled packets at the correct rate to realize high utilization. Homa's improvements mainly come from medium/large flows. In fact, our experiments show that Homa can achieve up to 24.6%/6.7% lower average FCT of medium/large flows (for brevity, we omit the detailed figures). For small flows, Homa performs worse than our QCLIMB. As shown in Fig. 28b and Fig. 28c, the reduction of QCLIMB over Homa in the average/tail FCT of small flows is up to 14.6%/24.2%. This is mainly because of Homa's proactive nature, which enforces Homa to blindly transmit unscheduled packets in the first RTT, causing traffic bursts, non-trivial queuing delay, and eventually, packet losses. Such first RTT issue will seriously affect the performance of small flows [31].

Comparison with pFabric and Homa using imprecise knowledge [Simulation]: Readers may wonder if QCLIMB can outperform pFabric and Homa with imprecise knowledge as scheduling input. To answer this, we conduct an experiment over PageRank workload with varying network load. The results are shown in Fig. 29. Compared to pFabric and Homa with imprecise knowledge, QCLIMB can reduce the average FCT of all flows by up to 18.8% and 25.6%, respectively. For the average/tail FCT of small flows, the reductions of QCLIMB are even larger: 85.9%∼87.1%/96.6%∼97.6% and 74.5%∼79.8%/93%∼94.7% over pFabric and Homa with imprecise knowledge, respectively.

E Discussion

The QCLIMB testbed measurements in this paper were based on 25 Gbps link speeds, but QCLIMB may still work in higher-speed networks. The reason is that the dominant overhead of an end-to-end QCLIMB flow is the end-host processing delay. Such processing delay is independent of the line-rates and is mainly caused by kernel network stack processing. In our evaluation, we observe that a 100KB flow requires roughly 200μs for the end-host processing on both sender and receiver sides. By contrast, QCLIMB's model inference takes only 3μs, which is negligible as compared to such processing delay. One can further reduce this inference latency to 1μs with advanced FPGA hardware [23] or may even reduce it to tens of nanoseconds by carefully pipelining RF's decision tree on hardware [44].

F Other Related Work

We have discussed the closely related works [7, 28, 10, 34, 41, 24, 37, 23] extensively in §2.1. Here, we only review some other DCN flow scheduling ideas that have not been discussed elsewhere. For example, NDP [27] uses receiver-driven scheduling but can only account for fair sharing rather than SRPT. Moreover, it relies on special hardware support from switches to provide receivers a full view of the traffic demand. Aeolus [31] adds selective dropping and loss recovery on top of receiver-driven transports (e.g., Home [37], NDP [27], ExpressPass[20]), suffering the same issues as above. dcPIM [16] takes multiple rounds of matching to compute conflict-free pairings of sender and receiver to achieve high utilization, but relies on precise flow size information and also needs to refactor the network stack. QJUMP [26] applies Internet QoS-like techniques (e.g., DiffServ [14]) to schedule flows of datacenter applications but requires the application itself to specify the priorities. Karuna [17] schedules a mix of flows with and without deadlines. It uses PIAS-like mechanisms for non-deadline flows and hence suffers from the same effectiveness problems as PIAS. PASE [39] synthesizes exiting transport designs to provide good performance, but requires non-trivial switch modification or complex control plane for arbitration. Auto [19] applies deep reinforcement learning (DRL) techniques to flow scheduling and traffic optimization. Aequitas [51] uses weighted fair queuing (WFQ) to guarantee RPC-level service-level objectives (SLOs), while QCLIMB employs strict priority queuing for prioritizing small flows to minimize FCTs. Above all, there exist no existing schedulers in the literature that can address the flow scheduling problems under imprecise flow sizes.

There are other DCN research efforts such as congestion control (e.g., DCTCP [5], D3 [47], MCP [18], L2DCT [38], HULL [6], HPCC [32], BCC [13, 9], MQECN [12], and ECN$^\sharp$ [50]) and multi-path load balancing (e.g., CONGA[4], Flowlet [46], and Hermes [49]). These designs are insufficient for FCT minimization as in-network priority queues are not used.