Analysis of Pyrrha: Congestion-Root-Based Flow Control Is Most Cost-Effective to Eliminate Head-of-Line Blocking

Zhaochen Zhang, Kexin Liu, Peirui Cao, Chang Liu, Yizhi Wang, Vamsi Addanki, Stefan Schmid, Qingyue Wang, Wei Chen, Xiaoliang Wang, *Senior Member, IEEE*, Jiaqi Zheng, *Senior Member, IEEE*, Wenhao Sun, *Member, IEEE*, Tao Wu, Ke Meng, Fei Chen, Weiguang Wang, Bingyang Liu, Wanchun Dou, Guihai Chen, *Fellow, IEEE*, Chen Tian, *Senior Member, IEEE*, Hao Yin, and Fu Xiao, *Senior Member, IEEE*

***Abstract*—In modern datacenters, the effectiveness of end-to-end congestion control (CC) is quickly diminishing with the rapid bandwidth evolution. Per-hop flow control (FC) can react to congestion more promptly. However, a coarse-grained FC can result in Head-Of-Line (HOL) blocking. A fine-grained, per-flow FC can eliminate HOL blocking caused by flow control, however, it does not scale well. This paper presents Pyrrha, a scalable flow control approach that provably eliminates HOL blocking while using a minimum number of queues. In Pyrrha, flow control first takes effect on the root of the congestion, i.e., the port where congestion occurs. And then flows are controlled according to their contributed congestion roots. A prototype of Pyrrha is implemented on Tofino2 switches. Compared with state-of-the-art approaches, the average FCT of uncongested flows is reduced by 42%-98%, and 99th-tail latency can be $1.6\times$-$215\times$ lower, without compromising the performance of congested flows.**

***Index Terms*—Data center network, congestion management, flow control.**

I. Introduction

GIVEN the increasingly stringent performance requirements on datacenter networks, avoiding congestion and the resulting delays has become critical for many applications. Indeed, measurement studies show that congestion events are frequent in today's datacenters, *e.g.*, bursty key-value stores [1], [2], web search services with massive queries [3], and data-parallel/machine-learning systems with partition/aggregation traffic patterns [4], [5], [6], [7], [8], [9], [10], [11]. Generally, congestion occurs at an output port when the arrival rate of traffic exceeds its link bandwidth. Queues build up at congested ports. With an inflated queue, flows could endure a long queuing delay or even face packet loss, hence flows' completion times (FCT) can be prolonged [3], [12].

State-of-the-art approaches to handle congestion is end-to-end congestion control (CC) [3], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23]. Congestion can be detected by senders when congestion signals are sent back or feedback delay is observed. Usually, it costs senders at least one Round-Trip Time (RTT) to be aware of the congestion, and then senders may take several RTTs to converge to an appropriate transmission rate. In modern datacenters, the effectiveness of end-to-end CC is quickly diminishing with the rapid bandwidth evolution [24], [25], [26], [27](§ II-A).

With the increasing link speed and performance requirements of datacenter networks, an intriguing and emerging alternative is per-hop flow control (FC), which can react to congestion much more promptly. It suppresses the transmission of the upstream entity before overwhelming the downstream queue, which avoids a large buffer occupancy. Generally, traffic in the same queue is controlled as a whole. Once the queue length exceeds a given threshold, a pause frame can be sent to pause the upstream entity [28], [29], avoiding further buffer build-up where congestion occurs. However, a coarse-grained flow control might spread the congestion to the whole network, inducing Head-Of-Line (HOL) blocking, and hurting the performance of victim flows [30], [31], [32], [33]. Here HOL blocking refers to flows being paused innocently (II-B). A naïve approach to eliminate HOL blocking could be to isolate each flow into different queues and control each of them separately. However, such a per-flow granularity flow control is not scalable since the hardware resources of switches are limited. State-of-the-art flow control approaches hence aim at reducing the number of queues required by compromising the granularity of isolation [24]. Thus, the HOL blocking can not be eliminated entirely (II-C).

This paper explores how to eliminate HOL blocking in a scalable manner, *i.e.*c, minimizing the required number of queues. We observe that when congestion occurs, flow control first takes effect on the root of the congestion, *i.e.*c, the port

Received 20 November 2024; revised 27 July 2025; accepted 16 November 2025; approved by IEEE Transactions on Networking Editor R. La. Date of publication 10 December 2025; date of current version 8 January 2026. This work was supported in part by the National Key Research and Development Program of China under Grant 2024YFB2906700; in part by NSF China under Grant 62325205, Grant 62172276, Grant 62441236, and Grant 62172204; and in part by the Postgraduate Research and Practice Innovation Program of Jiangsu Province under Grant KYCX25_0310. *(Corresponding author: Peirui Cao.)*

Zhaochen Zhang, Kexin Liu, Peirui Cao, Chang Liu, Yizhi Wang, Qingyue Wang, Wei Chen, Xiaoliang Wang, Jiaqi Zheng, Wanchun Dou, Guihai Chen, and Chen Tian are with the State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China (e-mail: caopeirui@nju.edu.cn).

Vamsi Addanki and Stefan Schmid are with TU Berlin, 10623 Berlin, Germany.

Wenhao Sun, Tao Wu, Ke Meng, Fei Chen, Weiguang Wang, and Bingyang Liu are with Huawei, Nanjing 210012, China.

Hao Yin is with Tsinghua University, Beijing 100084, China.

Fu Xiao is with Nanjing University of Posts and Telecommunications, Nanjing 210003, China.

Digital Object Identifier 10.1109/TON.2025.3636161

where congestion occurs. Then several congestion hotspots (*i.e.*c, output port with buffer build-up) can appear along the back pressure path of flow control. These involved hotspots form a congestion tree, where the root of the tree is the root of the congestion. Simply controlling the transmission of flows based on hotspots could induce HOL blocking. Instead, if we separately control the transmission of flows according to the congestion root they participate in, flows will not be paused innocently. As the number of concurrent congestion roots observed on each port is moderate, only a reasonably small number of queues on each port are required for traffic isolation in each switch [34].

Based on these insights, this paper presents Pyrrha, a congestion-root-based per-hop flow control protocol (III). In Pyrrha, each switch maintains a snapshot of the congestion status of the downstream network. Flows that would pass through the same congestion root in their downstream paths could be pushed into a dedicated congestion queue locally in each upstream switch. Then, flows passing through different congestion roots can be controlled separately as soon as possible. Pyrrha solves a set of challenges.

- ***How to identify a hotspot's role?*** When congestion occurs, multiple congestion roots may be claimed in succession. This could occur when an upstream congestion hotspot claims itself root while its downstream hotspot disagrees, and vice versa. With a local view, it is hard to tell which root is the exact root in the upcoming congestion. Intuitively, given the behavior of flow control, congestion roots are always locate on the most downstream ports that flows pass through. Inspired by the root-selection procedure of classic spanning tree algorithms [35], [36], Pyrrha employs a distributed self-stabilizing *merge* mechanism. A port can claim itself a *self-nominated* congestion root independently when it detects its queue buildup for the first time. It then abdicates its claim in favor of a *self-nominated* downstream root if (part of) its flows pass through the downstream claimer. Quickly, participating hotspots can converge to a congestion tree. Naturally, the congestion root is detected. (§ IV-A).
- ***How to identify a congested flow upon its arrival?*** For each arriving flow to a switch, its entire following path should be deterministic to the switch to identify a congested flow. Inspired by recent industrial path control practice [37], [38], Pyrrha proposes a hash-function-aware design for switches. Every switch can determine the path that a flow will take. With that information, the switch could match the path against the congestion status snapshot of the downstream network to determine whether it is a congested flow (§ IV-B).
- ***How to handle events-tangling scenarios?*** Congestion trees could overlap with each other which could result in a congested flow traversing several congestion roots. Besides, the congestion root in networks may vary with transient bursty traffic. Without careful scheduling among congestion queues, such flows could be mistakenly paused/resumed or delivered out-of-order. Pyrrha proposes a resource-efficient hierarchical queue structure corresponding to the physical topology. The design ensures both correct flow control semantic and in-order delivery even in highly dynamic scenarios (§ IV-C).

A framework is constructed for analyzing egress-based flow control protocols. We analytically prove that Pyrrha is a HOL-blocking-free per-hop flow control protocol with the minimum queue requirements (§ VI). Moreover, we demonstrate Pyrrha is deadlock-free in scenarios where other per-hop flow control protocols fail (§ VII).

A prototype of Pyrrha is implemented on Tofino2 [39]. Testbed evaluations and large-scale NS-3 simulations have been performed. We compare Pyrrha with existing flow control protocols (*e.g.*, Priority Flow Control (PFC) [40], and BFC [24]). And we also incorporate Pyrrha with existing congestion control protocols (*e.g.*, DCQCN [13], TIMELY [14], and HPCC [12]). We find that the average FCT of uncongested flows is reduced by 42.8%-98.2%, and 99th-tail latency can be $1.6\times$-$215\times$ lower, without compromising the performance of congested flows. In addition, Pyrrha reduces the maximum buffer occupancy by up to $1.8\times$-$6.2\times$ (§ VIII). *This work does not raise any ethical issues.*

II. Background and Motivation

A. CC Is Falling and FC Is Rising

A variety of datacenter applications produce bursty traffic, which can result in different types of congestion, *e.g.*, incast, and load imbalance. To handle congestion, existing efforts focus on developing end-to-end congestion control (CC). CC can be classified into *reactive* and *proactive*. With reactive CC, congestion can be detected by switches (*e.g.*, ECN in DCTCP [3] and DCQCN [13], INT measurements in HPCC [12], PINT [15], PowerTCP [20] and Poseidon [22]) or end-hosts (*e.g.*, Timely [14], Swift [16], and On-Ramp [17]). After receiving congestion signals or if packet delays are observed, senders adjust the transmission rate. It may cost a flow several RTTs to converge to an appropriate rate even in a stable network condition. With proactive CC, bandwidth is allocated before the transmission (*e.g.*, ExpressPass [23], [41], Homa [42], NDP [43], Aeolus [25], and pHost [44]). However, whether to transmit in the first RTT is a dilemma, and proactive CC either wastes the first RTT or risks reintroducing congestion. Recently there are CCs [21], [45] which detect congestion at sub-RTT by leveraging switches to send back control frames directly. However, they cannot quickly react to congestion especially when congestion occurs at the last hop (*e.g.*, incast).

Several trends. The control loop of CC is too long to handle transient congestion, given the fast evolution of datacenter networks: (i) The high port bandwidth allows to send out more flows within the first RTT, even before congestion control could step in [24] and [25]. Transient bursty traffic results in a large buffer occupancy and at the same time mislead the rate adjustment of CC. (ii) The buffer size cannot catch up with the increased speed of its high bandwidth per port [24], [27]. It becomes harder for switches to buffer transient congestion and wait for end-to-end CC's intervention. (iii) The growing scale of datacenter networks and the emerging workloads (*e.g.*, distributed training) lead to more bursty traffic (*e.g.*, a larger scale incast) [26], [46], [47].

Our vision. To handle bursty traffic, per-hop FC protocols should step in. As shown in Figure 1, we propose a labor division between CC and FC:

- **(i) Per-hop FC handles transient congestion.** A switch can control the traffic transmission quickly by per-hop flow control frames. It is in a unique position to quickly manage flows that have already been injected into the network to avoid performance downgrades.

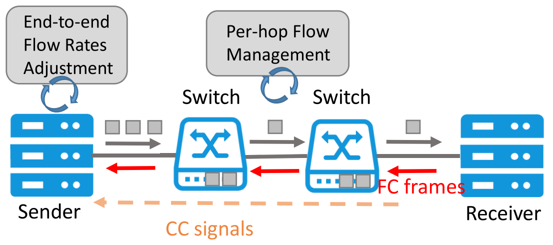


Fig. 1. Our vision of labor division between CC/FC.

- **(ii) CC takes its part to handle persistent congestion.** CC can adjust the flow rates to increase/decrease the forthcoming traffic injection into the network when congestion occurs and to handle fairness issues.

B. Hol Blocking Problem of Simple FC

In RoCEv2 [28], PFC [40] ensures that the buffer does not overflow. PFC pauses the upstream entity at a per-port or per-priority-class queue granularity when the ingress queue length exceeds a given threshold. Further, when the upstream ingress queue exceeds the threshold, a pause frame will be sent to its upstream entities.

However, the intervention of PFC could spread congestion. When PFC is applied, flows that do not contribute to downstream congestion could be paused when they share the same queue with congested flows. The congestion scope can spread from congested ports to piles of innocent ports. Hence, it could downgrade the performance of uncongested flows. We define the above congestion spreading phenomenon as the **HOL blocking caused by flow control**, *i.e.*c, a flow is paused innocently by the congested port that it does not pass through. HOL blocking could cause a throughput downgrade. More severely, PFC is vulnerable to deadlock with routing loops [30], [31], [32], [33].

Typical incast workloads. To demonstrate the HOL blocking problem, we conduct a simulation where incast flows are mixed with non-incast flows. 720-to-1 incast flows are generated with an average size of four Bandwidth-Delay Product (BDP) and non-incast flows are generated with a load of 0.8 following the Poisson arrival process (setting details in § VIII). Figure 2(a) depicts the real-time throughput. To make it more clear, we use ***vulnerable flows*** to denote uncongested flows sharing paths with congested flows in the remainder of the paper since they are more likely to be hurt by congested flows. Other uncongested flows are denoted as ***background flows***. Hence, the throughput of vulnerable flows is severely hurt since they are paused by downstream switches with congested flows as a whole, leading to a large queuing delay. Besides, since a PFC pause frame storm occurs, congestion is spread to the whole network. Consequently, background flows suffer a throughput downgrade from 1ms to 4ms.

MoE workloads. We investigate the performance of Pyrrha under the traffic of a popular type of pre-trained large language model called Mixture-of-Expert (MoE). According to [46], the traffic pattern can be characterized by an imbalanced all-to-all where a significant portion of the traffic is sent to a few 'hot' experts. Owning to the synchronized nature of the training process, the traffic exhibits a periodic on-off pattern [47], [48]. Following [47], two groups of periodic traffic are generated.

Figure 3 shows the performance of flows with collided phase, *i.e.*c, where their phase overlaps (the results of the interleaved phase are detailed in our conference paper [49]). In the case of DCQCN+PFC, two groups of all-to-all suffer from HOL-blocking as they compete for bandwidth, which in turn triggers PFC. The peak bandwidth lasts for approximately 1 ms during which non-hot experts complete their traffic reception, followed by hot experts continue receiving their traffic. The throughput of the all-to-all-1 group is notably suppressed, even dropping to zero upon the arrival of the all-to-all-2 group. Once a portion of the all-to-all-2 flows finishes, all-to-all-1 begins to grasp some of the bandwidth, as indicated by the red rectangle in the figure. While for DCQCN+Pyrrha, benefiting from the rapid reaction to the congestion, two groups of flows do not disturb each other, accelerating the tail latency by a factor of 1.46.

C. State-of-the-Art Flow Control Is Flawed

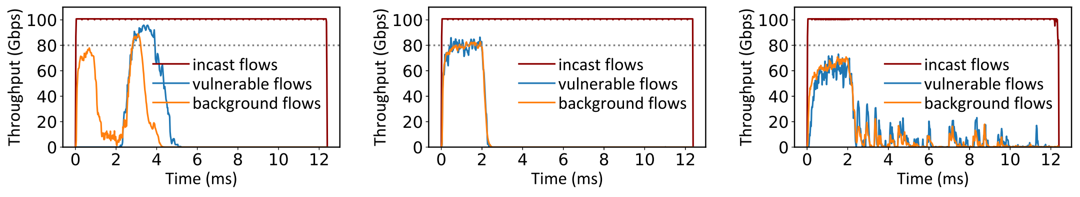
To overcome the HOL blocking problem caused by coarse-grained control on queues, a naïve scheme is per-flow queue FC scheme. Figure 2(b) demonstrates the simulation results under the same settings as in Figure 2(a) when the switch assigns a dedicated queue to each flow passing through it. Vulnerable and background flows fully utilize the link, at the same time the throughput of incast flows does not degrade. However, tens of thousands of flows can be observed on ports [50]. Hence a per-flow granularity scheme is non-scalable.

Existing flow control approaches try to reduce the number of queues by compromising the isolation granularity.

Destination-based flow control. This line of work tries to isolate congestion by separating flows transmitting to different destinations. Revisiting super-computing literature decades ago, per-destination Virtual Output Queues (VOQs) [51], [52], [53] are assigned to separate flows with different destination addresses [54], [55]. However, per-destination VOQs are not scalable since the number of VOQs required scales with the number of hosts in networks. Floodgate [56] is a per-hop flow control leveraging per-destination windows to identify incast traffic in datacenter networks. Then, incast traffic can be isolated from non-incast traffic. However, it should maintain a per-destination state of the remaining sending window which demands much memory resources on switches. In summary, they only aim at eliminating HOL blocking caused by the last-hop incast and cannot handle other types of congestion such as load imbalance. In addition, they require per-destination resources which build barriers to deployment at scale.

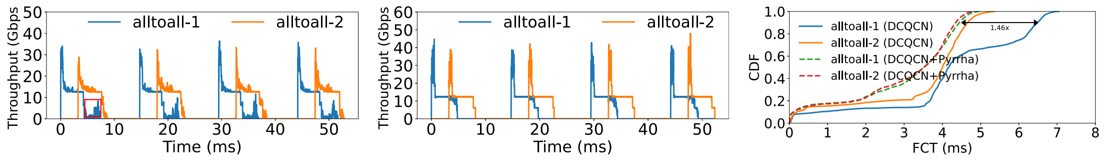
Queue-based flow control. A second line of work targets assigning flows into a limited number of queues to alleviate HOL blocking. Since it cannot isolate congested flows from uncongested flows entirely, HOL blocking cannot be avoided. In BFC [24], flows are assigned to a number of queues (*i.e.*c, 32-128 queues per port) according to their *flow-id*s and hash functions. A flow is assigned to an empty queue if possible and could share it with other flows when all queues are occupied. As shown in Figure 2(c), with relatively large incast flows, both vulnerable and background flows maintain a very low throughput from 2ms to 12ms. The tail latency of flows is prolonged by 6× compared with the per-flow queue scheme. This is because incast flows can occupy queues for a long time. Vulnerable and background flows sharing the same queue with incast flows are severely hurt, and their transmission rate is mistakenly controlled by the network bottleneck (*i.e.*c, the destination ToR of incast).

To sum up, existing solutions cannot totally avoid HOL-blocking, and some of them are impractical.



(a) PFC. (b) per-flow queue flow control. (c) BFC.

Fig. 2. Performance when incast flows are mixed with non-incast flows.



(a) Throughput (DCQCN). (b) Throughput (DCQCN+Pyrrha). (c) FCT comparison.

Fig. 3. Performance comparison under MoE workloads (collided phase).

III. Pyrrha Overview

Our target is to eliminate HOL blocking caused by flow control in the most cost-effective way. In this section, we first illustrate why congestion root is the appropriate granularity of flow control along with the basic idea of Pyrrha, followed by a list of challenges.

A. Basic Idea

Before illustrating our basic idea, we introduce several concepts in flow control when congestion occurs. Figure 4 shows a typical congestion tree rooted at P5.

- **Congestion Hotspot.** When the congestion occurs, a flow control scheme starts to pause upstream, inducing several hotspots along its back-pressuring path. A congestion hotspot is an output port whose input rate exceeds its output rate and its queue accumulates.
- **Congestion Root.** As its name implies, a congestion root is the root cause of the congestion, where congested flows finally aggregate. Meanwhile, it is the root of the corresponding congestion tree.
- **Congestion Tree.** A congestion tree can be made up of a root (*e.g.*, P5), non-root hotspots (*e.g.*, P1-P3) and leaf ports (already controlled by the root but have not paused its upstream yet). In our paper, a congestion tree is named after its root (*e.g.*, T5 denotes the tree whose root is P5).

Why congestion-root-based FC? A non-differentiating treatment of flows passing through congestion roots and hotspots could result in HOL-blocking since flows passing through hotspots might not contribute to the congestion. To avoid involving innocent flows, flow control should decide the right scope of flows to control. Intuitions are that if a flow control only applies pause to flows contributing to the congestion root, HOL-blocking can be eliminated. Meanwhile, flows passing through different congestion roots should be handled separately to avoid interfering with others.

Quick reaction to congestion: identify congested flows upon arrival. Once a congestion root is detected, the congestion root information is propagated to its upstream switches when it is detected. Then each switch maintains a congestion status snapshot of its downstream networks. A congested flow can be detected upon its arrival in networks by checking whether its path matches existing congestion roots. Hence, its transmission can be controlled several hops earlier before it arrives at congestion roots. It reduces the occurrence of severe congestion and relieves the buffer pressure on following hops, especially on congestion roots.

Fine-grained isolation: manage traffic according to its contributed congestion roots. By default, flows are pushed into a physical output queue (OQ). For separate control, congested flows passing through the same congestion root should be pushed into a dedicated isolation queue (IQ) assigned to the corresponding congestion root. Different categories of flows, *e.g.*, congested flows contributing to different congestion roots, and uncongested flows are isolated respectively. Then, the transmission of congested flows can be controlled precisely by pausing the exact queue assigned to the congestion root, which avoids congestion spreading.

Based on the above ideas, we propose Pyrrha, a practical fine-grained flow control scheme based on congestion roots. Pyrrha achieves good properties with the formal proof is placed in § VI and § VII:

- **HOL-blocking-free.** Pyrrha has no HOL blocking in any scenario (Theorem 1).
- **Minimal queue usage.** Pyrrha is a HOL-blocking-free flow control protocol requiring the minimal number of queues (Theorem 2).
- **Deadlock-robust.** Pyrrha is deadlock-free in some deadlock-prone scenarios where other per-hop flow control protocols fail (§ VII).

B. Design Challenges

Identify congestion roots. Correctly identifying the congestion root is the prerequisite of proper flow control. In tree T5 of Figure 4, P1-P3, and P5 should agree that P5 is the root of incast. Otherwise, if P1 incorrectly identifies itself as an independent root, vulnerable flow VS→VR becomes a victim.

Identify congested flows. To control the transmission of flows separately, congested flows should be identified precisely. When an uncongested flow is mistakenly identified (*e.g.*, VS→VR) as a congested flow, it can be paused incorrectly. To recognize a congested flow quickly, a switch should obtain the flow's path to be traversed in its downstream

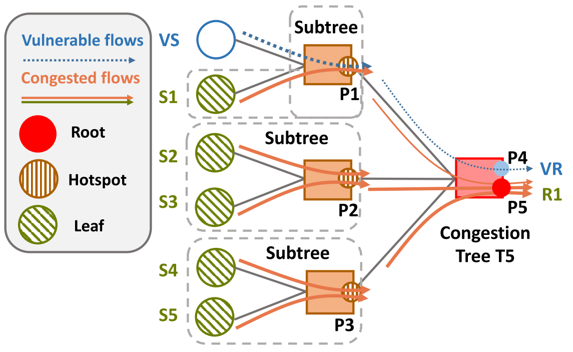


Fig. 4. Congestion tree illustration.

network so that it can check the path against its downstream network snapshot.

Handle tree-tangling scenarios. Several concurrent congestion trees can be intertwined among themselves. If non-root ports of trees are overlapped, these ports can play different roles in different congestion trees. These ports should not entangle transmission control received from different trees. When a tree is covered by another one, it results in a flow contributing to several congestion roots, which appear frequently in real systems. One may wonder why congestion still occurs at a port after part of its passing flows is controlled by its downstream congestion roots. This is because the congestion at the port is caused by itself other than its downstream ports. Figure 5(b) depicts an example that the path of a flow can first match a congestion root (P6) and then match another congestion root (P5). The transmission of the congested flow should be controlled by both congestion roots, *i.e.*c, only when both congestion roots send a RESUME frame can the flow be transmitted. Besides, the transmission control of the flow should not interfere with other flows sharing one of the congestion roots.

In addition, when the competing traffic changes, the congestion tree can shrink or expand, leading to congestion roots varying over time. For P3 in Figure 4, initially, packets of flows {S4-S5} →R1 are controlled in the IQ$_{P5}$. If flows {S1-S3} →R1 finish, P3 is likely to become the new congestion root later. Flows {S4-S5} →R1 are no longer the congested flows of P5. Instead, they are congested flows of P3. Later-arrived packets of these flows should be carefully scheduled in case they are transmitted before packets previously queued into the IQ$_{P5}$, which results in out-of-order delivery.

IV. Pyrrha Design

Figure 5 demonstrates the architecture of Pyrrha. The bottom part of the figure denotes the packet propagation among Pyrrha switches. And the upper part depicts the three major components of a Pyrrha switch. Congestion Root Identification (§ IV-A) responds to downstream switches to detect congestion and identify the corresponding root. The congestion information is carried in flow control frames (*e.g.*, PAUSE) and propagated to upstream switches in a hop-by-hop manner. Congested Flow Identification (§ IV-B) maintains the snapshot of downstream network congestion states to help quickly recognize congested flows through path matching. Isolation queues (IQs) are structured in a hierarchy corresponding to the topology by Congested Flow Management (§ IV-C). Details and discussions are put in § IV-D and § IV-E.

A. Congestion Root Identification

Initial detection. Inspired by the root-election process of spanning tree protocols, a congested port can claim itself a *self-nominated* congestion root candidate independently. Initially, each port is attached with a OQ and flows are pushed into the OQ by default. Hence the queue length increase on the OQ can be regarded as an indication of congestion. When a data packet arrives at the OQ, a switch checks whether the queue length exceeds a given threshold K$_{pause}$ (*e.g.*, several per-hop BDPs). If so, a hotspot is detected and the hotspot regards itself as a congestion root. Subsequent packets that arrive at the hotspot trigger a PAUSE frame to the corresponding upstream port from which the packet arrived.

Congestion root identification. According to behavior of flow control, a root is always the most downstream hotspot in a tree. Hence, we can identify the real congestion roots by merging upstream congestion tree into a more downstream one. As shown in Figure 5(a), when a congestion-root-candidate hotspot (P1) receives a PAUSE frame from a downstream root (P5), it indicates that part of its passing-through flows also traverses this downstream hotspot. Hence it recognizes itself as a false-positive congestion root. A new IQ for this new congestion root (P5) is assigned. All following packets matching the new root will enter the corresponding IQ. Then, the old congestion tree is canceled and merged into the new congestion tree. Note that this process can be iterative when there exist multiple layers of hotspots in a congestion tree. The false-positive congestion roots are eventually merged and the root of the new congestion tree is the real congestion root.

Merging process. To start merging, the false-positive congestion root notifies all its child nodes by sending a control message MERGE. MERGE is sent to all its upstream entities belonging to the (old) false-positive congestion root, carrying the ID of both old and new congestion roots. As shown in Figure 6, switches receiving the MERGE frame change the state of the corresponding IQ to soft-merging and propagate the notification to its upstream further. Soft-merging means that the old IQ now belongs to no congestion tree and can be unassigned once empty. The packets queuing in the old IQ are not controlled by the false-positive congestion root. Instead, only packets passing through the real congestion root are controlled (§ IV-C). The merging process finishes within a one-way delay, hence false-positive congestion roots have a negligible impact on performance.

B. Congested Flow Identification

Intuitions are that a congested flow passes through at least one congestion root.

Determining a flow's exact path. To determine whether a data packet belongs to a congested flow, the entire onward-path of each arriving packet at a switch should be deterministic. Pyrrha is compatible with traffic load balancing protocols that can locally get deterministic onward paths for flows [38], [57], [58], [59], [60]. Among those load balancing protocols, hash-based protocols (*e.g.*, per-flow ECMP and PLB [37]) are most widely deployed given its no-reordering properties [61]. Pyrrha proposes a hash-function-aware design. Pyrrha's switch calculates a packet's onward-path by using its IP tuples, routing hash functions, and seeds of its downstream switches, together with flow labels carried in its header if necessary as input. Besides, source routing is compatible with Pyrrha

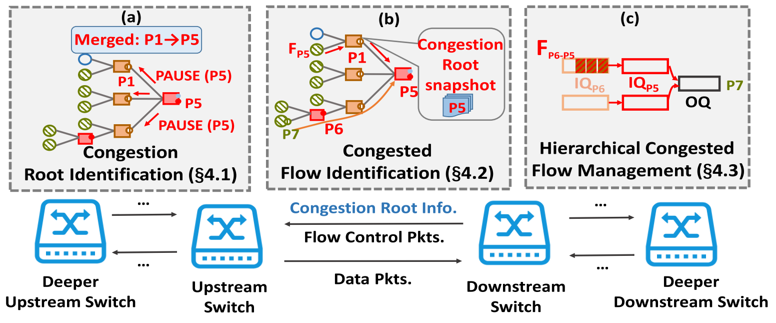


Fig. 5. Pyrrha architecture.

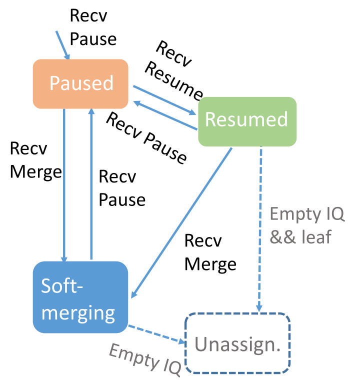


Fig. 6. IQ state transitions.

naturally since Pyrrha's switch can derive the onward-path of a packet by parsing its header.

The memory and computation resource to get a flow's path is moderate, and following optimizations are facilitated by leveraging the industry-standard practices in datacenters. (i) Given that switches in datacenter only support limited types of hash functions (*e.g.*, CRC or XOR) to conduct efficient calculation [61], [62], [63], Pyrrha switch only needs to store the type of the hash function of other switches along with their hash seeds. (ii) In many widely Fdeployed topologies [32], [64], [65], [66], the multiple equal path property and the up-down routing strategy can be leveraged to optimize the overhead. In those topologies, the forwarding tables of all core switches are identical. Hence, Pyrrha switch only needs to store one replica for core switches' forwarding tables. Furthermore, for fat-tree, the path from a core switch to a given destination is unique, hence Pyrrha switch only conducts hash function calculation for the first two hops.

For other adaptive load balancing approaches, *e.g.*, per-packet spraying and DRILL [42], [67], which determine the path of flows through dynamic states, Pyrrha is a complementary solution to handle the hotspots caused by destination collision (*i.e.*c, incast), where load balancing falls short. Experiments in our conference paper [49] show that Pyrrha is compatible with DRILL and further improves the tail latency by 18.3% compared to pure Pyrrha. Pyrrha also handles corner cases where flows are re-routed to different paths due to link failures (for further details on the evaluation of this feature, see our conference paper [49]).

Matching and maintaining snapshot. Pyrrha's switch maintains the *congestion-root table* of the downstream networks. As shown in Figure 5(b), the table maintains a snapshot of the congestion states of its downstream networks. When a PAUSE frame indicating a new congestion root is received, the congestion root is recorded in the table. When a packet arrives at the switch, its path can be obtained via above mentioned methods. The switch checks whether its path matches any entry in the congestion-root table. For port P1 in Figure 5(b), packets that will traverse P5 in its downstream path are identified as belonging to a congested flow. Hence, it's enqueued to a separate IQ, which is paused/resumed based on the state of the corresponding root. Otherwise, the packet belongs to an uncongested flow and is put in the OQ.

C. Congested Flow Management

To handle tangled scenarios where congestion trees are intertwined among each other or congestion roots vary over time, Pyrrha leverages a hierarchical methodology to manage congested flows corresponding to that of the topology. It can be supported by a Hierarchical Isolation Queue (HIQ) architecture, which manages congested flows in a hierarchy. Pyrrha installs HIQ during compilation according to its location in networks and manages the usage of queues dynamically during runtime through a mapping table. We also provide single-tier IQs prototype to fully support the function of HIQ.

Handling congested flow in hierarchy. Congestion trees are intertwined when non-root ports of trees are overlapped or a tree is covered by another one. For the first scenario, IQs on ports can naturally isolate control from different congestion roots on non-root ports. For the latter one, a congested flow could pass through multiple congestion roots. To ensure precise control isolation, a congested flow should match all corresponding IQs before it is forwarded. A hierarchical organization of IQs based on the location of their corresponding congestion roots in the topology enables a congested flow to match appropriate IQs *sequentially*. Especially, when per-flow load balancing is used, a flow at most encounters one congestion root among switches at the same level. It ensures in-order delivery when a congested flow alters its matched IQs.

Hierarchical Isolation Queue (HIQ) architecture. HIQ consists of several levels of IQs. Each IQ is positioned in a hierarchy according to its distance to the corresponding congestion root in the physical topology. Hence, the number of layers of the HIQ is determined by its location in the network. As shown in Figure 5(c), in a two-tier network, an uplink port of a ToR switch maintains two levels of IQs, since the farthest potential congestion root is two hops from it. Figure 5(c) depicts the HIQ architecture on P7, *i.e.*c, the leaf port in Figure 5(b). P7 fully utilizes the two-level architecture of the HIQ, since the farthest congestion root P5 is two hops from it. Especially, the OQ is connected to the last level of the HIQ architecture. A dedicated scheduler is equipped for each level of queues to schedule the traffic transmission. Only when an IQ is in a resumed or soft-merging state can its packet be dequeued and pushed into the next-level IQs/OQ. From the perspective of a packet, this process is performed iteratively until it reaches the OQ. In the scenario depicted in Figure 5, when a flow that will traverse congestion root P6 and P5 arrives at the upstream switch of port P6, it matches the HIQ *from-near-to-far*. After a packet is dequeued from IQ$_{P6}$, it is pushed into a next-level IQ$_{P5}$. When there is no more matched IQ, it is pushed into the OQ. Hence, the packet can be forwarded to the next hop only after all matched congested roots are resumed. In this way, congested flows are controlled locally precisely.

IQs in HIQ are arranged by levels, supporting in-order delivery naturally. Considering the merging procedure in Figure 5(a), the congestion root is changed from P1 to P5. IQ$_{P1}$ is in the soft-merging state and packets in it can be mixed with congested and uncongested traffic. Pyrrha should handle them separately to avoid HOL blocking. Congested flows of root P5 are dequeued from IQ$_{P1}$ and then pushed into the next level IQ$_{P5}$. Uncongested flows are forwarded to OQ. In this way, precise isolation is achieved without inducing re-ordering.

Handling secession of the congestion root. Once the OQ length of the congestion root decreases below the resume threshold, it sends back RESUME to its upstream. Likewise, upstream switches could resume their upstream when their own IQ decreases below the resume threshold. It is an iterative process that the congestion tree eliminates starting from the leaf switches to the root. A congestion port becomes a leaf when it has not sent PAUSE yet, or when the status of all its upstream entities is set to *unassigned*. The IQ of the leaf switch is unassigned when it becomes empty. When all the upstream IQs of a congestion root are marked unassigned and the queue length of OQ of the congestion root is below the resume threshold, the congestion root disappears naturally. Figure 6 depicts the state transition of the IQ usage. Only when an empty IQ is in a resumed or soft-merging state can it be marked as unassigned.

D. Miscellaneous Detailed Design

Congestion information propagation. There are three types of flow control frames in Pyrrha, *e.g.*, PAUSE, RESUME, and MERGE. These control frames carry the congestion information and control the transmission of congested flows accurately in a hop-by-hop manner. Correspondingly, there are three states for an IQ, *i.e.*c, paused, resumed, or soft-merging. The state transitions of an IQ are shown in Figure 6.

When a congestion root is detected, a PAUSE frame is sent back to the upstream port through the ingress port of which the data packet is just received. Once a data packet is pushed into the OQ which is attached to the congestion root, Pyrrha switch checks the packet's ingress port and sends back the PAUSE frame. A PAUSE frame carries the ID of the congestion root (*i.e.*c, identified as *switch-id:port-id*). Likewise, when the queue length of IQ exceeds the threshold K$_{pause}$, a PAUSE frame carrying the root ID is sent back to its upstream switch.

Cooperation on end-hosts. To handle persistent congestion, end-hosts should control the upcoming traffic into networks. Pyrrha can cooperate with congestion control protocols. And Pyrrha can perform better if end-hosts can respond to PAUSE (or RESUME and MERGE) frames. To pause and resume at a per-flow granularity, end-hosts could leverage a pull-based transmission model, which can be implemented by

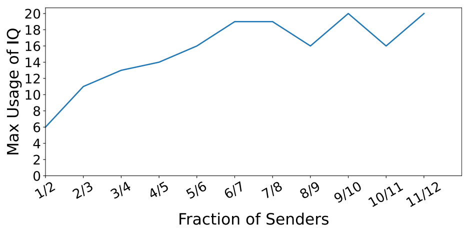


Fig. 7. Queue usage analysis.

programmable smart NICs in RDMA networks. Especially, Pyrrha can also handle end-host congestion (*e.g.*, PCIe congestion) by backpressuring the traffic it receives.

Handling rare packet loss. Pyrrha reduces queue lengths significantly. Hence, buffer overflow rarely occurs. However, Pyrrha cannot guarantee lossless in all scenarios. In an $m:1$ incast, the victim egress port will accumulate $K_{pause}+m\times$per-hop BDPs of data. In extreme cases, such as when each egress port of the switch alternately becomes the victim of a $k-1:1$ incast (k denotes the number of ports on switches), total buffer accumulation may exceed switch capacity, causing packet loss. In this case, Pyrrha can leverage IRN [68] for fast retransmission.

E. Design Discussions

Queue consumption. In the most extreme scenario, the number of congestion roots can be the number of ToRs in the network. However, the concurrent amount of congestion in networks is usually moderate. It is reported that only 3% of the links in edge and aggregation layers appear as a hotspot for more than 0.1% of time intervals [34], [69]. Moreover, the concurrent roots can be much less than that of hotspots.

To investigate the IQ usage, evaluations under stressful workloads where $(m-1)$ out of m ToRs send traffic simultaneously to the left $1/m$ of the ToRs are conducted. A $k=12$ fat-tree topology is employed and each host sends 40 one-BDP flows continuously to create a substantial network burst. Figure 7 illustrates the IQ usage of Pyrrha as the fraction of senders varies. The IQ utilization ranges from 6-20, approximately proportional to the number of switch ports k, which is considered to be relatively moderate. Since commodity switches can support thousands of VOQs [51], [52], [53], assigning a dedicated queue to each downstream root is feasible. To handle corner cases where IQs are not enough, similar to BFC, Pyrrha leverages hash functions [70], [71] to choose an IQ according to the congestion root ID, at a cost of sacrificing precise isolation.

Deadlock prevention. Pyrrha is deadlock robust since OQs never get paused. To prevent cyclic buffer dependencies (CBD) caused by routing loops, Pyrrha switch checks whether the congestion root carried in the PAUSE frame is identical to its own identifier. If so, it ignores the PAUSE frame directly (§ VII).

Incremental Deployment. To support incremental deployment, Pyrrha-enabled switches can operate in a hybrid environment with legacy PFC switches by enabling PFC for interoperability with legacy switches. In this mode, PFC provides a lossless fabric between Pyrrha and PFC switches. Meanwhile, traffic traversing multiple Pyrrha-enabled switches benefits from its head-of-line blocking-free flow control, substantially improving performance.

V. Implementation and Testbed Experiments

We implement a Pyrrha prototype on Tofino2, a state-of-the-art programmable switch ASIC [39] with Reconfigurable Match Table (RMT) architecture. In this section, we briefly describe the key modules of the prototype, followed by the overhead analysis. Testbed evaluations show that Pyrrha can achieve good performance (§ V-B). More implementation details are placed in our conference paper [49].

A. Prototype of Pyrrha

We implement a Pyrrha prototype on Tofino2 with 2.5k lines of P4 code and 2k lines of Python code. The operations of Pyrrha is implemented entirely in the data plane at line rate.

Key modules and the pipeline. The Pyrrha prototype is mainly composed of several modules, *i.e.*c, (i) congestion root matcher, (ii) queue manager, (iii) queue state detector, and (iv) signal packet module.

(i) Upon arrival, the data packet first undergoes a standard processing procedure, including forwarding and admission control. Subsequently, the data packet is forwarded to the congestion root matcher, an integral component of the path calculation unit and a congestion root table, facilitating the congested flow identification. Specifically, the path calculation unit calculates the packet's egress ports of its onward path. The congestion root table records the status of ports, indicating whether the port is congested or not. (ii) Then packets enter the queue manager for queue assignment. The queue manager assigns queues to traffic based on the congestion roots it would pass through. Central to its design is a multi-segment stack, wherein each segment manages the available queues for a specific egress port.

(iii) The queue state detector checks whether the length of the assigned queue exceeds the pause threshold or decreases below the resume threshold, and then triggers appropriate signal packets. The queue length is retrieved by utilizing *ghost threads* in Tofino2.

(iv) When it is necessary to send a signal packet, the signal packet module leverages the packet trigger functionality to construct signal packets, such as PAUSE and RESUME. Upon receiving a PAUSE or RESUME, the module engages Tofino2's *AFC (Advanced Flow Control)* mechanism to pause (or resume) the queue.

Feasibility of HIQ The architecture of HIQ is supported in current Metro Ethernet (MetroE) service routers [72], [73]. And a recent work of implementing multi-level scheduler on ASIC [74] also verifies its feasibility. According to private talks with chip vendors, they consider it possible to implement HIQ in their next-generation switching chips. For instance, the two-layer HIQ can be obtained by connecting two traffic manager models in series and specifying the next-level IQ to be pushed in when a packet is dequeued. Although HIQ is not supported by the architecture of Tofino2 currently, the features of HIQ can be fully supported via single-tier queues (see our conference paper [49] for details).

Complexity and overhead. Tofino2 adopts pipeline architecture, wherein the resource allocation is determined at compile time. It enables us to ascertain Pyrrha's resource requirements without running it in a large-scale cluster. According to the statistics reported in megascale [75], the scale of current data centers can reach up to 10,000 hosts. Therefore, we use a k=36 fat-tree topology with 11,664 hosts as a representative case. Pyrrha prototype can easily

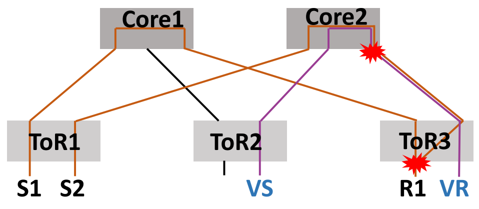
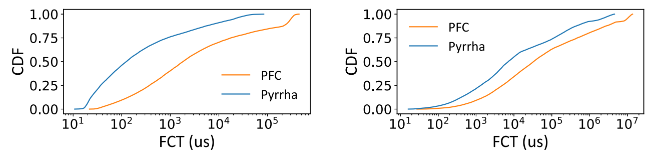


Fig. 8. Testbed topology.



(a) Vulnerable flows (Web Server). (b) Vulnerable flows (Web Search).

Fig. 9. FCT of testbed experiments.

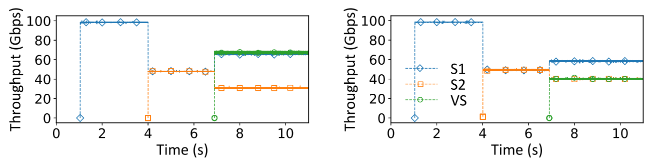
scale to it, with around 11 MB (*i.e.*c, 44.5% of Tofino2) of the memory resource consumption. Specifically, the memory usage of Pyrrha prototype is mainly composed of three units, *i.e.*c, path calculation unit, congestion root table, and queue manager, overall consuming 9.25 MB. And the processing logic consumes around 1.88 MB. (i) As analyzed in § IV-B, the memory consumption of the path calculation can be optimized by leveraging the industry standard, occupying 0.44MB SRAM. (ii) The congestion root table is organized hierarchically, where the nth table records ports that are n hops away from the switch, and the port is identified as <switch-id, port-id>. Hence, the storage usage of the congestion root table is proportional to the number of ports in networks, occupying 176 KB SRAM. (iii) The queue manager firstly checks whether the congested flow is assigned a queue through a *IsAssigned Table* and assigns a queue if necessary by looking up a multi-segment *QueueId stack* which records the available queue. Then it updates the *QueueId record table* to record the queue assignment status. This module overall consumes 8.64MB SRAM. Furthermore, anticipating the rapid maturation of HIQ technology [74], we pre-design a HIQ-compatible pipeline. Our analysis shows that the reduced queue requirements in HIQ significantly lower system overhead, enabling Pyrrha to scale to topologies an order of magnitude larger when HIQ becomes available (detailed analysis is placed in our conference paper [49]).

B. Testbed Evaluation

Topology. We use a 2-level leaf-spine topology as shown in Figure 8, consisting of three ToR, two core switches, and two hosts per rack, all connected via 100 Gbps links.

Workloads. We evaluate Pyrrha under incast-mix scenarios. Incast flows are generated by letting hosts S1 and S2 transmit flows to host R1 simultaneously. Vulnerable flows are generated by letting host VS send flows to host VR. Flows S2 →R1 and VS→VR share the same port on the core switch. Web Server and Web Search flows are generated following a Poisson arrival process (§ VIII).

Pyrrha reduces the FCT of vulnerable flows. Figure 9 demonstrates the FCT performance of vulnerable flows. For PFC, vulnerable flows are HOL blocked by incast flows, suffering a large queuing delay. Pyrrha quickly detects the



(a) Pyrrha (b) PFC

Fig. 10. Throughput of testbed experiments.

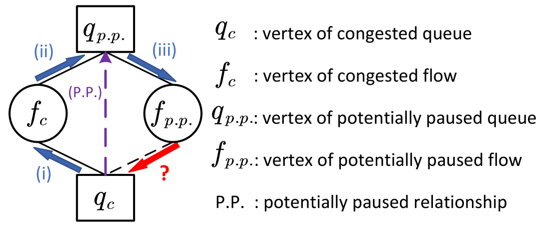

$q_{p.p.}$
f_c
$f_{p.p.}$
q_c
(i)
(ii)
(iii)
(P.P.)
?
q_c : vertex of congested queue
f_c : vertex of congested flow
$q_{p.p.}$: vertex of potentially paused queue
$f_{p.p.}$: vertex of potentially paused flow
P.P. : potentially paused relationship

Fig. 11. The pattern of flow control and HOL blocking in the framework.

congestion on the destination ToR switch and isolates incast flows into a dedicated queue. Thus, the FCT of vulnerable flows is greatly reduced.

Pyrrha improves the throughput. Figure 10(a) shows the throughput when flows on three hosts S1, S2, and VS start to arrive at 1s, 4s, and 7s, respectively. Pyrrha improves the throughput of vulnerable flows to 66.7 Gbps without compromising the overall throughput of incast flows (100 Gbps). The network throughput is improved by 26.7 Gbps.

VI. Head-of-Line Blocking Analysis

Overview. In this section, we propose a framework for analyzing the existence of HOL blocking and the queue usage of flow control protocols. We first construct a formal model based on a bipartite graph of flows and queues and provide the necessary definitions for the analysis (§ VI-A). Then, we establish a formal criterion to determine whether a flow control protocol is HOL-blocking-free (§ VI-B). Finally, we apply this framework to formally prove that Pyrrha is not only HOL-blocking-free but also achieves this with the minimum usage of queues (§ VI-C).

Insight. Inspired by the gradient graph [76], [77], we represent flows and queues as vertices, and their traversing relationship as edges. Figure 11 visually summarizes the core concept and the logic of our proof, where rectangles represent queues and circles represent flows. Congestion at the congested queue (q_c) is caused by a set of congested flows (f_c) passing through it, which are presented as neighbors in the graph (relationship i). A flow control protocol pauses these congested flows by pausing the upstream queues they traversed, which are referred to as potentially paused queues $q_{p.p.}$ (relationship ii). However, this action risks HOL blocking if $q_{p.p.}$ is also shared by other innocent flows, which are referred as potentially paused flows $f_{p.p.}$ (relationship iii). In order to prove that Pyrrha is HOL-blocking-free, the core of our proof is to formally demonstrate that for any congested queue in Pyrrha, the set of potentially paused flows ($F_{p.p.}$) is identical to the set of congested flows (F_c). Furthermore, we prove that any flow control protocol using fewer queues than Pyrrha cannot avoid HOL blocking through Dirichlet's box principle and proof by contradiction.

TABLE I
Notations Used in Pyrrha Analysis

Notation	Description
S	Scenario representing the state of the data center network at a given time, including topology, traffic matrix, congested ports, and congested flows.
$G = (V, E)$	Graph describing a scenario
f	Flow in the network (vertex in G)
F	Set of flows
q	Queue in the network (vertex in G)
p	Port
q_{p-qid}	Queue at a port p with a queue ID qid [1]
Q	Set of queues
$N(\cdot)$	Set of neighborhood in graph G
$R_p(f, i)$	i-th port in the route of flow f
$R_i(f, p)$	Order of the port p in the route of flow f ($\perp$ if f does not pass through p).
$P(q)$	Port that q belongs to
$Q(p)$	Set of queues at port p
$Q_c(p)$	Set of congested queues at port p (empty set if the port is not congested)
$Q(p, f)$	Queue that flow f passes through at port p
$f_c \mid F_c(\cdot)$	Congested flow | Set of *congested flows* for a congested queue or a congested port
$p_{p.p.} \mid P_{p.p.}(\cdot)$	Potentially paused port | Set of *potentially paused ports* with respect to a congested queue or a congested port
$P_{p.p.}(q, f)$	Set of *potentially paused ports* for a flow f which passes congested queue q
$q_{p.p.} \mid Q_{p.p.}(\cdot)$	Potentially paused queue | Set of *potentially paused queues* with respect to a congested queue or a congested port
$f_{p.p.} \mid F_{p.p.}(\cdot)$	Potentially paused flow | Set of *potentially paused flows* with respect to a congested queue or a congested port

[1] In different flow control protocols, the qid is often presented in different ways. This will be explained in detail below.

A. Model and Definitions

We first present the notations used in the analysis and then formally define the analysis framework. For clarity, notations are gathered into Table I.

We denote by S, a scenario representing the state of the datacenter network at a specific time instance. The scenario S captures the topology, traffic matrix, congested ports, and congested flows in the datacenter network. We define an undirected graph G to describe each congestion scenario S. Definition 1 gives the formal definition. Specifically, the graph consists of a vertex set V divided into two sets: the set of flows F and the set of all queues Q in the network, similar to the gradient graph in [76] and [77]. Each flow and each queue is then a vertex in our graph G. The edge set consists of edges denoted by $e = (f, q)$, if and only if a flow f traverses queue q. There is only one edge between f and q.

Definition 1: (Graph:) A scenario S is described by an undirected graph $G = (V, E)$, where $V = F \cup Q$ is the set of vertices consisting of two disjoint subsets: flows F and queues Q. The edges E represent the pass-through relationship between flows and queues, *i.e.*c, $E = \{(f, q) \mid f \text{passes through } q\}$. G is a bipartite graph.

In our analysis, we are interested in the interactions between queues and flows traversing them under a specific flow control protocol. To this end, we define the neighborhood set $N(v)$ as the set of neighborhoods of vertex v. Definition 2 gives our formal definition.

Definition 2: (Neighborhood set:) The neighborhood set of a vertex v, denoted by $N(v)$, is the set including all vertices adjacent to v. For a vertex set $U \subseteq V$, we denote by $N(U)$ the union of neighborhood sets of all vertices $v \in U$.

$$N(v) = \{v' \mid (v, v') \in E\} \tag{1}$$

$$N(U) = \bigcup_{v \in U} N(v) \tag{2}$$

Flow control protocol. It is difficult to model all the protocols within one framework. Among all flow control protocols, the most prevalent type is the *per-hop* flow control protocol. To the best of our knowledge, all lossless flow control protocols operate in a per-hop manner [24], [29], [40]. In contrast, non-per-hop flow control protocols need a large headroom to be lossless, which is unaffordable on commodity switches. Therefore, we only analyze per-hop flow control protocols in our analysis.

Further, we only analyze egress-based flow control protocols. There are many works based on the ingress queue. However, most of the commodity datacenter switches have only egress queues, and the ingress queues are only counters rather than buffers. This leads to the fact that ingress-based works can be analyzed using the egress-based framework. Consider a congestion scenario where two flows from different ingress queues converge at an egress queue. Under PFC, an ingress-based protocol, each congested ingress queue pauses its corresponding upstream egress queue. In our analysis framework, this is equivalent to the egress queue being congested and then pausing the two upstream egress queues according to the flows' ingress direction.

The fundamental control entity of the flow control protocol is a *queue*. Thus, we use a queue as the basic unit to define the flow control protocol. In our framework, we consider congested ports and congested flows to be pre-determined and flow control protocols only decide the queue assignment and queue behaviors.

Definition 3: (Congested flows:) Given a congested queue q_c, the set of congested flows $F_c(q_c)$ is defined as the set of all flows passing through the congested queue q_c *i.e.*c$F_c(q_c)$ is the neighborhood set of q_c in the graph G.

$$F_c(q_c) = N(q_c) \tag{3}$$

In flow control protocols, the pause to a congested flow f_c is achieved by pausing queues that f_c passed *earlier* than the corresponding congested queue q_c.

Definition 4: (Potentially paused ports:) The potentially paused ports denoted by $P_{p.p.}(q_c, f_c)$ are the ports that each f_c passed through earlier than q_c. Let $R_i(f, p)$ denote the order of port p along the path of flow f, then $P_{p.p.}(q_c, f_c)$ is given by,

$$P_{p.p.}(q_c, f_c) = \{p \mid R_i(f_c, p) \neq \perp \wedge \\ R_i(f_c, p) < R_i(f_c, P(q_c))\} \tag{4}$$

$$P_{p.p.}(q_c) = \bigcup_{f_c \in F_c(q_c)} P_{p.p.}(q_c, f_c) \tag{5}$$

Based on our definition of potentially paused ports $P_{p.p.}(q_c)$ (Definition 4), we can now define potentially paused queues $Q_{p.p.}(q_c)$ as the set of queues that a congested flow f_c passes through at each port in $P_{p.p.}(q_c)$. In addition, as indicated by relationship II in Figure 11, all queues that each f_c passes through earlier than q_c are potentially paused by per-hop flow control protocols.

Definition 5: (Potentially paused queues:) Given a specific congested queue q_c, potentially paused queues $Q_{p.p}(q_c)$ are queues traversed by the congested flows passing through q_c. Specifically, let $Q(p, f)$ denote the queue that a flow f passes through at port p. Then, the set of potentially paused queues $Q_{p.p}(q_c)$ is the union of queues traversed by every congested flow $f_c \in F_c(q_c)$ (Definition 3), before reaching q_c:

$$Q_{p.p.}(q_c) = \bigcup_{f_c \in F_c(q_c)} \{Q(p, f_c) \mid p \in P_{p.p.}(q_c, f_c)\} \tag{6}$$

Alternatively, we can write $Q_{p.p}(q_c)$ as,

$$Q_{p.p.}(q_c) = \{q_{p.p.} \mid \exists f_c \in F_c, s.t., q_{p.p.} \in N(f_c) \wedge \\ R_i(f_c, P(q_{p.p.})) < R_i(f_c, P(q_c))\} \tag{7}$$

where $P_{p.p}(q_c, f_c)$ is the set of potentially paused ports (Definition 4), $R_i(f, p)$ is the order of port p along the path of flow f. The set of potentially paused queues w.r.t. a congested port p is the union of the potentially paused queues w.r.t each queue belonging to the port p *i.e.*c

$$Q_{p.p.}(p) = \bigcup_{q_c \in Q_c(p)} Q_{p.p.}(q_c) \tag{8}$$

Similar to the Definition 4, 5, as indicated by relationship III in Figure 11, the potentially paused flows are the flows that pass through potentially paused queues, *i.e.*c, the neighborhood of potentially paused queues in graph G. In the following, we formally define potentially paused flows.

Definition 6: (Potentially paused flows:) Given a congested queue q_c, potentially paused flows $F_{p.p}(q_c)$ is the set of flows traversing the potentially paused queues $Q_{p.p}(q_c)$, *i.e.*c, $F_{p.p}(q_c)$ is the neighborhood of $Q_{p.p}(q_c)$:

$$F_{p.p.}(q_c) = N(Q_{p.p.}(q_c)) \tag{9}$$

Given a congested port p, potentially paused flows $F_{p.p}(p)$ is the set of flows traversing the potentially paused queues $Q_{p.p}(p)$, *i.e.*c, $F_{p.p}(p)$ is the neighborhood of $Q_{p.p}(p)$:

$$F_{p.p.}(p) = N(Q_{p.p.}(p)) \tag{10}$$

Based on the model and definitions introduced in this section, we next study the Head-of-line blocking properties of flow control protocols.

B. Head-of-Line Blocking

In this subsection, we introduce the pattern of HOL blocking. As mentioned in the background section of the paper, HOL blocking caused by flow control is defined as a flow being paused innocently by the congested port that it does not pass through. According to the definition, we use a two-step approach to detect HOL blocking. First, we identify all flows potentially paused by flow control triggered by a congested port. Then we check whether these flows are innocently paused by a congested port that they do not traverse.

Recall that Section VI-A introduces the flow control pattern and derives the set of potentially paused queues $Q_{p.p.}(q_c)$. We then determine whether potentially paused flows traverse the congested port. If a potentially paused flow traverses the congested port, it is not subject to HOL blocking. If all potentially paused flows for a congested port traverse that port, we can conclude that no HOL blocking occurs for this congested port.

Definition 7: (No HOL blocking:) There is no head-of-line blocking due to a congested port p if and only if the following conditions are satisfied: all the potentially paused flows $F_{p.p}(p)$ w.r.t port p traverse a queue at port p. This can be formally defined as:

$$\forall f \in F_{p.p.}(p), f \in N(Q(p)) \tag{11}$$
$$\iff F_{p.p.}(p) \subseteq N(Q(p)) \tag{12}$$

If there is no HOL blocking for all congested ports, then there is no HOL blocking in the network.

Since the vertices in the graph G are mapped one-to-one with the flows and queues in the network, the HOL blocking criterion (Definition 7) based on the graph G is *sufficient and necessary*. Specifically, if there is a potentially paused flow $f_{p.p.} \in F_{p.p.}(p)$ not adjacent to any queue of the congested port in the graph G, then the flow can be paused by congested port p but not traverse the congested port p, *i.e.*c, the HOL blocking occurs, and vice versa.

C. Analysis of Pyrrha

In this section, we first formally define Pyrrha's mechanism and then present a formal proof. We will prove two theorems related to Pyrrha.

- Theorem 1: Pyrrha has no HOL blocking in any scenario.
- Theorem 2: Pyrrha is a HOL-blocking-free flow control protocol requiring the minimal number of queues.

Discussion about Pyrrha with different queue organizations. We only analyze Pyrrha with single-tier IQs for now, and leave the generalization for future work. The main challenge for analyzing Pyrrha with HIQ is that it is hard to model all the feasible behaviors of HIQ. However, we believe that theorems of Pyrrha are not affected by the way it is implemented.

For Theorem 1, the property of no HOL blocking comes from the isolation of different congested flows. Thus, the property can be maintained as long as the flows can be isolated, regardless of the queue organization.

For Theorem 2, the number of required queues equals the number of congested flow groups that need to be isolated, as each group needs a separate queue for isolation. In Pyrrha, the number of congested flow groups relates to the number of congestion roots, which is independent of queue organization. In fact, Pyrrha with HIQ will use fewer queues than Pyrrha with single-tire IQ.

Formal definition of Pyrrha. Recalling the design of Pyrrha (§ IV), a *congested port* can claim itself as a self-nominated *congestion root* and begin to send out PAUSE frames. Once the congested port receives a PAUSE frame from a downstream congested port, it *abdicates its claim* and turns into a *hotspot* caused by downstream congested port. A *hotspot* could still claim itself a *congestion root* if it becomes a *congested port* again (*i.e.*c, congestion detected in OQ again). Summarizing the above process, we find that when a port becomes a *congested port*, it claims itself as a *congestion root*; and when it *abdicates its claim*, it turns into a *hotspot* simultaneously. Therefore, the two terms *congested port* and *congestion root* are equated when describing the mechanism of Pyrrha in the following analysis.

In a congested port of Pyrrha, there must be a congested OQ and perhaps some congested or uncongested IQs. Recalling that in the analysis framework, when we analyze a congested queue, we consider all queues that would be paused by the hop-by-hop pause of the congested queue as potentially paused queues. Thus, the pause behavior of an IQ can be seen as the hop-by-hop relay of the pause behavior of the OQ of the congested port corresponding to this IQ, which is contained by the analysis of the OQ of the congested port. Therefore, we only need to analyze the OQ of each congested port to cover flow control behaviors of Pyrrha.

For a congested queue q_c, Pyrrha pauses the congested flows passing through it by sending PAUSE frames to upstream ports. Each port receiving a PAUSE frame attempts to place congested flows into the corresponding IQ for the congested queue they traverse, creating the IQ if it does not exist. Therefore we use the set of OQs the IQ corresponding to as the qid. As an example, we denote the IQ at the upstream port p traversed by a flow that already passed through two congested OQs q_1 and q_2 as $q_{p\text{-}\{q_1,q_2\}}$. Especially, the qid of an OQ is an empty set $\emptyset$ (or denoted as $\{\}$).

The formal definition of the mechanism of Pyrrha is:

$$Q_{p.p.}(q_c) = \{q_{p\text{-}\{\cdots,q_c,\cdots\}} \mid p \in P_{p.p.}(q_c)\} \tag{13}$$
$$f \in N(q_{p\text{-}\{\cdots,q_c,\cdots\}}) \iff q_c \in N(f) \tag{14}$$

Equation 13 comes from the fact that the congestion signal from congested queue q_c only pauses the IQ corresponding to itself. And Equation 14 comes from the fact that the flow is pushed in IQ corresponding to q_c if and only if the flow passes through congested queue q_c. The queue assignment of Pyrrha is dynamic, and these two equations hold after every assignment operation.

No HOL blocking analysis Then we prove that there is no HOL blocking in Pyrrha.

Theorem 1: Pyrrha has no HOL blocking in any scenario.

Proof: From Equation 1, we can deduce that:

$$\begin{aligned} v_1 \in N(v_2) &\iff \exists e \in E, e = \{v_1, v_2\} \\ &\iff v_2 \in N(v_1) \end{aligned} \tag{15}$$

Combining Equation 14 and Equation 15, we get:

$$f \in N(q_{p\text{-}\{\cdots,q_c,\cdots\}}) \iff f \in N(q_c) \tag{16}$$

Then, we have:

$$\forall f \in N(Q_{p.p.}(q_c)), f \in N(q_c) \tag{17}$$

Finally, we can deduce that:

$$F_{p.p.}(q_c) = N(Q_{p.p.}(q_c)) \subseteq N(q_c) \tag{18}$$

For any port p, we have:

$$\begin{aligned} F_{p.p.}(p) &= \bigcup_{q_c \in Q_c(p)} F_{p.p.}(q_c) \\ &\subseteq \bigcup_{q_c \in Q_c(p)} N(q_c) \subseteq N(Q(p)) \end{aligned} \tag{19}$$

That is, the no HOL blocking criterion holds for any port in Pyrrha. Thus Pyrrha has no HOL blocking in any scenario.□

Queue usage analysis. Then we prove that Pyrrha is a HOL-blocking-free flow control protocol requiring the minimal number of queues. There are three kinds of queues in the network when flow control protocol has been deployed: congested queue, potentially paused queue, and normal queue that is neither congested nor potentially paused. The number of normal queues is irrelevant to the behavior of the flow control protocol. Therefore, we only consider *the number of congested queues and potentially paused queues*.

To improve readability, we state in advance the insights and notations used in the following proof. In the following proof, we use the pigeonhole principle twice, once applied on the queues (Lemma 1) and once on the ports (Theorem 2), and we use * to denote the port containing more queues or the queue containing more flows. To accomplish the proof, we first prove the following helper lemmas.

Lemma 1: Assume there are two one-tier schemes to organize the queues and flows at port p, namely X and Y. If the number of queues of X is less than that of Y, at least one queue in X contains flows that belong to at least two queues in Y.

Proof: Upon examining the queues for port p under Y, it is clear that this port cannot have only one queue; otherwise, this port would not contain any queue in scheme X. Therefore, there must be n ($n \geq 2$) queues in Y. In the single-tier queue scheme, neighborhood sets of these n queues are pair-wise disjoint, *i.e.*c, a flow will not pass through two queues on the same port. Thus, we can use the pigeonhole principle on sets of flows and queues in protocol X. Putting n pair-wise disjoint sets of flows in less than n queues, there must be a queue q_X^* containing flows from two sets of flows. □

Denoting the set of queues as $Q_X()$ and $Q_Y()$, and the two queues in Y as q_{Y1} and q_{Y2},the formal expression of Lemma 1 is:

$$\begin{aligned}&|Q_X(p)| < |Q_Y(p)| \Rightarrow\\&\exists q_X^* \in Q_X(p), \exists q_{Y1}, q_{Y2} \in Q_Y(p), s.t.\\&N(q_X^*) \cap N(q_{Y1}) \neq \emptyset, N(q_X^*) \cap N(q_{Y2}) \neq \emptyset\end{aligned} \tag{20}$$

Lemma 2: If there exists a flow control protocol that uses fewer queues than Pyrrha on a port, the flow control protocol must incur HOL-blocking on that port.

Proof: Denote the flow control protocol as A and the port is p^*. We use $Q_A(p^*)$ to denote the set of queues on port p^* in protocol A, and $Q_P(p^*)$ for that in Pyrrha. It is clear that the number of queues for Pyrrha on port p^* must be larger than 1. Otherwise, A can not use any queue on port p^*. Then, we can apply Lemma 1 on port p^*. After substitute A to X and P for Pyrrha to Y,we obtain that:

$$\begin{aligned}&\exists q_A^* \in Q_A(p^*), \exists q_{p^*\text{-}Q_{c1}}, q_{p^*\text{-}Q_{c2}} \in Q_P(p^*), s.t.\\&N(q_A^*) \cap N(q_{p^*\text{-}Q_{c1}}) \neq \emptyset, N(q_A^*) \cap N(q_{p^*\text{-}Q_{c2}}) \neq \emptyset\end{aligned} \tag{21}$$

In the equation, q_A^* stands for the queue at port p^* in A that contains flows that belong to at least two queues in Pyrrha. $q_{p^*\text{-}Q_{c1}}$ and $q_{p^*\text{-}Q_{c2}}$ stand for two queues at port p^* in Pyrrha that some of the flows belong to them are contained by q_A^*, where Q_{c1} and Q_{c2} denote the sets of congested OQ the queues corresponding to.

Without loss of generality, we assume that the difference $Q_{c1}/Q_{c2} \neq \emptyset$ (if not, just swap the notation of the two sets). For a congested OQ $q_{p'\text{-}\{\}} \in Q_{c1}/Q_{c2}$ which is at p', there must be a flow f_c' passing through q_A^* and congested port p' in protocol A:

$$\exists q_{p'\text{-}\{\}} \in Q_{c1}/Q_{c2}, s.t., \exists f_c' \in N(q_A^*) \cap N(Q(p')) \tag{22}$$

Thus, q_A^* is a potentially paused queue with respect to p' in protocol A.

$$q_A^* \in Q_{p.p.A}(p') \tag{23}$$

However, because q_A^* contains flows which belong to $q_{p^*\text{-}Q_{c2}}$ and $q_{p'\text{-}\{\}} \not\subseteq Q_{c2}$, there must exist a flow passing through q_A^* but not passing through p'.

$$\begin{aligned}&\exists f' \in N(q_A^*) \cap N(q_{p^*\text{-}Q_{c2}}), s.t.\\&\quad f' \in N(Q_{p.p.A}(p')), f' \notin N(Q(p'))\end{aligned} \tag{24}$$

It can be further deduced that.

$$\begin{aligned}&N(Q_{p.p.A}(q_c')) \not\subseteq N(Q(p'))\\&\qquad \Rightarrow F_{p.p.A}(q_c') \not\subseteq N(Q(p'))\end{aligned} \tag{25}$$

That is, the no HOL blocking criterion does not hold for congested port p'. And the HOL blocking occurs at port p^* in flow control protocol A. □

Note that the above proof does not restrict q_A^* to be a potentially paused queue or a congested queue, nor does it require both $q_{p^*\text{-}Q_{c1}}$ and $q_{p^*\text{-}Q_{c2}}$ to be IQs (potentially paused queues). Therefore, Lemma 2 holds for *both congested queues and potentially paused queues*.

Theorem 2: Pyrrha is a HOL-blocking-free flow control protocol requiring the minimal number of queues.

Proof: We prove this by contradiction. We first assume the existence of a lossless flow control protocol A that can use fewer queues than Pyrrha to achieve HOL-blocking-free. According to the pigeonhole principle, protocol A must use fewer queues at a port p^* than Pyrrha. According to Lemma 2, protocol A must incur HOL-blocking on port p^*. This implies that our hypothesis does not hold. Therefore, Pyrrha is a HOL-blocking-free flow control protocol requiring the minimal number of queues. □

VII. Deadlock Analysis

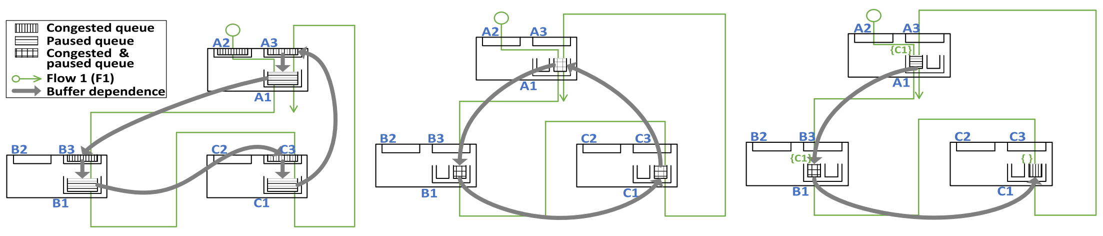
In this section, we analyze the ability of Pyrrha to prevent deadlocks. We use two deadlock-prone scenarios to show the deadlock-robustness of Pyrrha. In each scenario, we first show how deadlocks occur in PFC and the state-of-art flow control algorithm (*e.g.*, BFC). Then we illustrate why Pyrrha prevents deadlock in these scenarios. Finally, we discuss the intuition why Pyrrha provides deadlock-robustness on other scenarios.

Scenarios where a flow's routing forms a loop. In this scenario, a flow first passes through an egress port and then is forwarded back to this port again, which could be attributed to incorrect routing table entries.

An example of this scenario is shown in Figure 12. There are three switches in the figure, where boxes on the top side represent ingress ports and boxes on the bottom side represent egress ports in each switch. In each egress port, there are one or more egress queues, depending on the flow control protocol. We use vertical shading to denote queues that are congested (*i.e.*c, the queues that pause other queues), and horizontal shading to denote queues that are paused. If a queue is both paused and congested, it is marked by grid shading. The flow stuck in the routing loop is drawn in a green line. We use a hollow circle to represent the starting position of the flow, and an arrow to represent the direction of the flow.

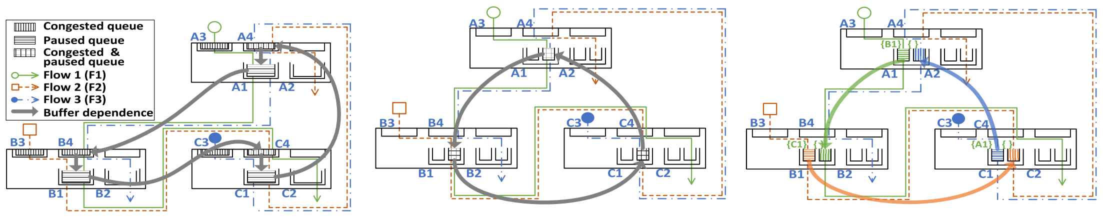
The flow first enters the upper switch from $A2$, exits from $A1$, and is then successively forwarded to port $B1$ at the left switch, to port $C1$ at the right switch, and finally to port $A1$ again at the upper switch. We assume that port $C1$ becomes congested first in our following analysis.

In PFC (Figure 12(a)), the ingress buffer counter of port $C3$ rises and eventually triggers a PAUSE frame to port $B1$. Then the same process is repeated between $B1$ and $B3$, and then



(a) PFC. (b) BFC. (c) Pyrrha.

Fig. 12. The scenario where a single flow forms a loop.



(a) PFC. (b) BFC with **worst-case hash collision**. (c) Pyrrha.

Fig. 13. The scenario where multiple flows sequentially form a loop.

between $A1$ and $A3$. Finally, port $C1$, the initially congested port, is paused by port $A3$, which represents the formation of a deadlock. The buffer dependencies of PFC in this scenario are marked with thick gray arrows in the figure. We can clearly see the Cyclic Buffer Dependency (CBD) among ports $C1$, $C3$, $B1$, $B3$, $A1$, and $A3$.

As shown in Figure 12(b), a similar situation occurs in BFC. BFC assigns a dedicated queue to the flow $F1$ at each egress port. The queue of port $C1$ pauses the queue of port $B1$ once the queue length reaches K_{pause}. Cascadingly, the queue of port $B1$ pauses the queue of $A1$, and the queue of $A1$ pauses the queue of $C1$. As a result, there is a CBD between the queue for $F1$ at ports $A1$, $B1$, and $C1$.

As shown in Figure 12(c), CBD does not exist in Pyrrha. In the figure, we use the same notation to denote OQ and IQ as in Section VI. The congestion that occurs at port $C1$ pauses the corresponding IQ at port $B1$. Then the IQ $q_{B1-\{C1\}}$ pauses IQ $q_{A1-\{C_1\}}$. When the queue length of $q_{A1-\{C_1\}}$ reaches k_{pause}, it sends a pause frame with root ID $C1$ to port $C1$. However, congestion root $C1$ does not react to the PAUSE frame that carries the root ID identical to itself. Thus, for Pyrrha, CBD never forms in this scenario. Although this may result in packet loss on port $A1$, it is much less a threat to the network performance than deadlock.

Scenarios where multiple flows' routing forms a loop sequentially. In this scenario, none of the flow forms a loop by itself. But when multiple flows pass through different ports one after another, a loop can occur. This kind of loop could be attributed to rerouting due to link failure in tree-shape topologies such as fat-tree and spine-leaf [78].

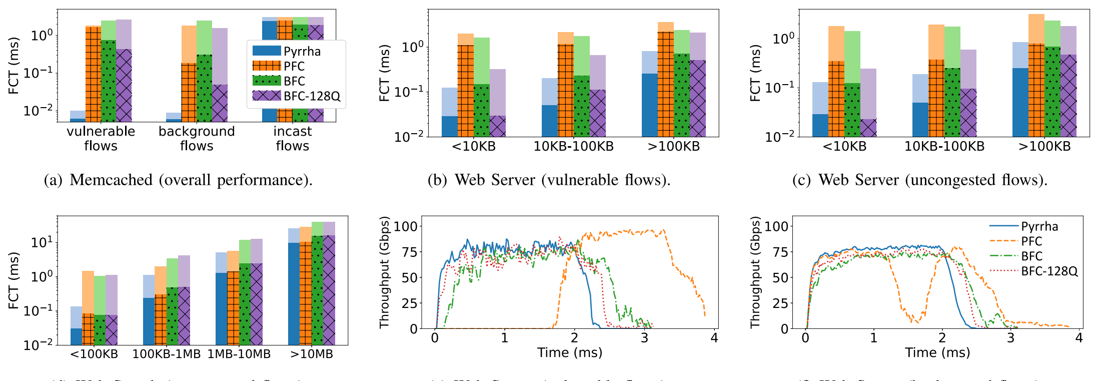
As shown in Figure 13, there are three flows in this scenario, where $F1$, drawn by green solid line, passes through egress ports $A1$, $B1$, and $C2$; $F2$, drawn by orange dashed line, passes through egress ports $B1$, $C1$, and $A2$; $F3$, drawn by green dotted-dashed line, passes through egress ports $C1$, $A1$, and $B2$. Assuming that port $C1$ becomes congested first.

The reaction of PFC in this scenario is shown in Figure 13(a). Considering $F2$ passes through port $C4$ and $C1$ sequentially, the egress congestion at port $C1$ leads to the ingress buffer counter rising at port $C4$. And a PAUSE frame is sent to port $B1$ once the ingress buffer counter reaches $XOFF$. Then the ingress buffer counter of port $B4$ increases due to port $B1$ being paused. Similarly, the ingress buffer counter of port $A4$ increases due to the pause of $A1$, and finally a PAUSE frame is sent back to port $C1$. As we can see in the figure, the CBD is formed among ports $C1$, $C3$, $B1$, $B3$, $A1$, and $A3$, which indicates the existence of deadlock.

For BFC, the idea of isolating different flows in different queues would have avoided deadlocks in this scenario. However, considering the limited number of available queues, the impact of the hash collision on BFC cannot be ignored. Figure 13(b) shows the worst-case hash collision of BFC, where all critical flows are hashed into a collision queue. It is clear that the collision queues at ports $A1$, $B1$, and $C1$ form a CBD.

Figure 13(c) shows the behavior of Pyrrha. In the beginning, $F2$ and $F3$ are queued in OQ at port $C1$ (referred to as $q_{C1-\{\}}$). Once $q_{C1-\{\}}$ is congested, it sends PAUSE frames to its upstream ports. Then a corresponding IQ $q_{B1-\{C1\}}$ is assigned at port $B1$, and following arrival packets of $F2$ are pushed into $q_{B1-\{C1\}}$. $F1$ is pushed into the OQ $q_{B1-\{C1\}}$ at port $B1$ directly since it does not pass through a downstream congested root. Because of the competition for bandwidth between IQ $q_{B1-\{C1\}}$ and OQ $q_{B1-\{\}}$, both queues are subject to queue buildup. A pause frame with root ID $B1$ is sent to port $A1$ when the length of $q_{B1-\{\}}$ exceeds K_{pause}. Then $F1$ is pushed into the IQ $q_{A1-\{B1\}}$ since it passes through a downstream congestion root $B1$. Like $q_{B1-\{\}}$, $q_{A1-\{\}}$ then becomes congested and sends PAUSE frames to its upstream port $C1$. Finally, the system converges to the state shown in Figure 13(c). In the figure, we use colored arrows to represent the buffer dependencies of the corresponding colored queues. As we can see, there is no CBD in Pyrrha since every paused queue eventually have buffer dependency on an OQ, which is never dependent on other queues.

Other scenarios. Generally, to prove there is no deadlock means to prove there is no CBD. In Pyrrha, there are two kinds of queues, OQ and IQ. Only IQ could be paused, and each



(a) Memcached (overall performance). (b) Web Server (vulnerable flows). (c) Web Server (uncongested flows). (d) Web Search (uncongested flows). (e) Web Server (vulnerable flows). (f) Web Server (background flows).

Fig. 14. Performance of FC. The deep/light color in the figures represents the average/99th-tail value for each bar.

IQ has its corresponding OQs. For any paused IQ in Pyrrha, we can work out its buffer dependency chain and eventually find an OQ at the end of the chain. Since OQ would not be paused, *i.e.*c, it does not depend on any queues, any buffer dependency chain in Pyrrha does not form a loop. In other words, in general cases, there is no CBD (*i.e.*c, no deadlock) in Pyrrha. And we are still trying to formalize the proof and extend it to all scenarios.

VIII. Simulation Evaluation

Topology. A non-blocking clos-network is used. It contains 4 core switches, 10 ToRs, and 160 hosts, similar to the topology used in [42]). Each ToR is connected to its hosts and cores via 100/400 Gbps links, respectively. The per-hop propagation delay is 600ns. The base RTT is 5.1μs, and the base BDP is 64KB. A 3-tier fat-tree topology with 1024 hosts is also leveraged to investigate the scalability of Pyrrha.

Workloads. Under incastmix scenarios, flows following a Poisson arrival process with a load of 0.8 and periodic incast flows each composed of 30-40 MTUs with a load of 0.5 are generated. An incast destination does not receive Poisson arrival flows hence the traffic load does not exceed link bandwidth. The incast degree is 720-to-1. For Poisson arrival flows, three workloads are used [3], [42], [79], where Memcached is composed of small flows, where most of the flows are smaller than 1KB, and Web Server and Web Search are large flows mixed with small flows where a small ratio of large flows dominate the average flow size.

Parameters. There are two parameters of Pyrrha, *i.e.*c, the threshold to pause (*i.e.*c, K$_{pause}$ times one-hop BDP) and the threshold to resume (*i.e.*c, K$_{resume}$ times one-hop BDP). In our evaluations, K$_{pause}$ is set to 2, and K$_{resume}$ is set to 1. Besides, the maximum number of IQs can be used is set to 100, but Pyrrha only uses a dozen of IQs in most cases. (We discuss why these values are used in our conference paper [49]). The switch buffer capacity is 20MB. Pyrrha uses shared buffer mode. PFC uses dynamic threshold and $\alpha = 2$.

Metrics. Average/99th-tail FCTs are evaluated. We monitor the maximum buffer on each hop to investigate the composition of buffer reallocation that Pyrrha brings.

A. Comparing With Flow Control

In this section, we use large-scale NS3 simulations to compare Pyrrha with existing flow control protocols (*e.g.*, PFC and BFC). For BFC, two versions with 32 and 128 queues per port are used, as in its paper (*e.g.*, BFC is used to denote BFC-32). Figure 14 shows the FCT and throughput.

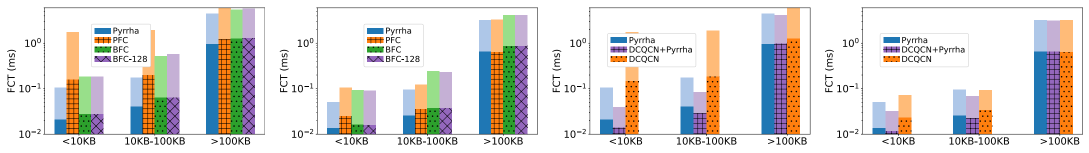
1) PFC: PFC hurts the performance of uncongested flows since they can be paused innocently by their downstream ports when incast occurs. This especially hurts the performance of workloads that are composed of small flows (*e.g.*, Memcached). For Web Server workload, PFC even spreads the congestion to the whole network thus background flows that do not share the same destination pods of incast flows also get hurt. The throughput performance depicted in Figure 14(f) is a side note to this issue. For background flows, it achieves stable high throughput until PFC pause frame storm occurs at 1.5ms. Then background flows endure a significant throughput loss that lasts for about milliseconds.

2) BFC: BFC assigns flows to multiple queues according to flow identifiers (FID) and hash functions. It can partially alleviate HOL blocking caused by congested flows and improve the performance of uncongested flows to an extent compared to PFC. However, HOL blocking occurs when congested flows and uncongested flows share the same queue, or flows are hashed into the same flow FIDs. Hence, BFC can not obtain extremely low latency as Pyrrha does. Along with the number of queues used by BFC increases, *i.e.*c, from 32 to 128, the performance of BFC is improved. For Web Search workload, the tail latency of BFC is not good because BFC sets a relatively smaller threshold to detect congestion compared to PFC. It can risk spreading congestion.

The third group of bars in Figure 14(a) shows the performance of incast (congested) flows. More results of incast flows are deferred to our conference paper [49]. Pyrrha does not compromise the performance of incast flows compared to PFC. BFC reduces the average FCT of incast flows because BFC splits different flows into different queues and incast flows may use several queues simultaneously. Incast flows can use more bandwidth during the resume phase via the round-robin scheduling mechanism among queues.

B. Additional Evaluations and Discussions

Performance under Large-scale Topology. To investigate the scalability of Pyrrha, evaluations under $k = 16$ fat-tree topology (*i.e.*c, 1024 hosts) is conducted. Figure 15 shows the performance of different protocols under Web Server. We analyze the results from two perspectives: (i) When comparing FC with CC protocols, especially for flows smaller than 100KB, fine-grained FC protocols such as BFC and Pyrrha



(a) Web Server (vulnerable flows). (b) Web Server (uncongested flows). (c) Web Server (vulnerable flows). (d) Web Server (uncongested flows).

Fig. 15. FCT performance under k=16 fat-tree.

outperform PFC as well as CC protocols. This advantage is attributed to the rapid response of fine-grained FC to network congestion, which provides a degree of mitigation against HOL blocking. (ii) In terms of cooperation between FC and CC, as depicted in Figure 15(c), when Pyrrha is integrated with CC, small flows can achieve the lowest latency. This is because Pyrrha rapidly reacts to the network congestion, preventing HOL blocking for traffic that has already been injected into the network. Concurrently, CC addresses the persistent congestion associated with larger flows.

Cooperating with congestion control. We evaluate the performance of Pyrrha when integrated with congestion control algorithms in detail. Our experiments demonstrate that Pyrrha significantly improves the performance of uncongested flows by quickly identifying and isolating incast flows. Specifically, Pyrrha reduces the FCT of small flows (less than 100KB) by up to 32.7× and substantially decreases buffer occupancy at congestion roots through rapid and precise flow control. For detailed experimental results, including DCQCN, HPCC, TIMELY, and comprehensive results analysis in different scenarios, please refer to our conference paper [49].

Additional evaluations. In our conference paper [49], we further evaluate Pyrrha under varying incast flow sizes, multiple congestion roots, interleaved MoE traffic, and dynamic congestion root scenarios. We also compare Pyrrha with per-flow queuing and CC without sending windows, analyze parameter selection and merging mechanisms, and investigate a single-tier queue variant. Additionally, we demonstrate that Pyrrha complements adaptive load balancing scheme (*e.g.*, DRILL [67]) by addressing incast-induced congestion while load balancing schemes handle path collision-induced congestion.

Discussions. Discussions regarding how Pyrrha handles link failures, as well as a review of related works, are put in our conference paper [49].

IX. Conclusion

This paper was motivated for a labor division between congestion control and flow control. It is time to embrace per-hop flow control in datacenter networks to react to congestion promptly. We presented Pyrrha, a congestion-root-based per-hop flow control. It controls the transmission of flows at a fine granularity without congestion spreading, requiring a minimum number of queues. The performance of flows can be significantly improved. We are currently discussing with a major vendor the implementation of Pyrrha in its products.

Acknowledgment

The authors would like to thank editors and anonymous reviewers for their valuable comments. When this paper was written, Zhaochen Zhang, Kexin Liu, Peirui Cao, Chang Liu, Yizhi Wang, Qingyue Wang, Wei Chen, Xiaoliang Wang, Jiaqi Zheng, Wanchun Dou, Guihai Chen, Chen Tian were affiliated with State Key Laboratory for Novel Software Technology, Nanjing University. Wenhao Sun, Tao Wu, Ke Meng, Fei Chen, Weiguang Wang, Bingyang Liu were affiliated with Huawei. Vamsi Addanki, Stefan Schmid were affiliated with TU Berlin. Hao Yin was affiliated with Tsinghua University. Fu Xiao was affiliated with Nanjing University of Posts and Telecommunications.

References

[1] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, “Workload analysis of a large-scale key-value store,” in *Proc. 12th ACM SIGMETRICS/Perform. Joint Int. Conf. Meas. Model. Comput. Syst.*, Jun. 2012, pp. 53–64.
[2] B. Fitzpatrick. (2011). *Memcached: A Distributed Memory Object Caching System*. [Online]. Available: http://www.memcached.org/
[3] M. Alizadeh et al., “Data center TCP (DCTCP),” in *Proc. ACM SIGCOMM Conf.*, Aug. 2010, pp. 63–74.
[4] J. Dean and S. Ghemawat, “Mapreduce: Simplified data processing on large clusters,” in *Proc. Commun. ACM*, 2008, pp. 107–113.
[5] M. Zaharia et al., “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,” in *Proc. NSDI*, 2012, pp. 15–28.
[6] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark: Cluster computing with working sets,” in *Proc. HotCloud*, 2010, p. 10.
[7] G. Ananthanarayanan et al., “Reining in the outliers in map-reduce clusters using mantri,” in *Proc. OSDI*, 2010, pp. 265–278.
[8] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, “Managing data transfers in computer clusters with orchestra,” *ACM SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, pp. 98–109, Oct. 2011.
[9] M. Abadi et al., “TensorFlow: A system for large-scale machine learning,” in *Proc. OSDI*, 2016, pp. 265–283.
[10] Y. Peng et al., “A generic communication scheduler for distributed DNN training acceleration,” in *Proc. 27th ACM Symp. Operating Syst. Princ.*, Oct. 2019, pp. 16–29.
[11] E. Ghabashneh, Y. Zhao, C. Lumezanu, N. Spring, S. Sundaresan, and S. Rao, “A microscopic view of bursts, buffer contention, and loss in data centers,” in *Proc. 22nd ACM Internet Meas. Conf.*, Oct. 2022, pp. 567–580.
[12] Y. Li et al., “HPCC: High precision congestion control,” in *Proc. ACM Special Interest Group Data Commun.*, Aug. 2019, pp. 44–58.
[13] Y. Zhu et al., “Congestion control for large-scale RDMA deployments,” in *Proc. ACM Conf. Special Interest Group Data Commun.*, Aug. 2015, pp. 523–536.
[14] R. Mittal et al., “Timely: RTT-based congestion control for the datacenter,” in *Proc. SIGCOMM*, 2015, pp. 537–550.
[15] R. Ben Basat, S. Ramanathan, Y. Li, G. Antichi, M. Yu, and M. Mitzenmacher, “PINT: Probabilistic in-band network telemetry,” in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Architectures, Protocols Comput. Commun.*, Jul. 2020, pp. 662–680.
[16] G. Kumar et al., “Swift: Delay is simple and effective for congestion control in the datacenter,” in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Archit., Protocols Comput. Commun.*, Jul. 2020, pp. 514–528.
[17] S. Liu, A. Ghalayini, M. Alizadeh, B. Prabhakar, M. Rosenblum, and A. Sivaraman, “Breaking the transience-equilibrium nexus: A new approach to datacenter packet transport,” in *Proc. NSDI*, 2021, pp. 47–63.
[18] V. Arun, M. Alizadeh, and H. Balakrishnan, “Starvation in end-to-end congestion control,” in *Proc. ACM SIGCOMM Conf.*, Aug. 2022, pp. 177–192.
[19] P. Goyal, A. Narayan, F. Cangialosi, S. Narayana, M. Alizadeh, and H. Balakrishnan, “Elasticity detection: A building block for internet congestion control,” in *Proc. ACM SIGCOMM Conf.*, Aug. 2022, pp. 158–176.

[20] V. Addanki, O. Michel, and S. Schmid, "PowerTCP: Pushing the performance limits of datacenter networks," in *Proc. NSDI*, 2021, pp. 51–70.
[21] S. Arslan, Y. Li, G. Kumar, and N. Dukkipati, "Bolt: Sub-RTT congestion control for ultra-low latency," in *Proc. NSDI*, 2023, pp. 219–236.
[22] W. Wang et al., "Poseidon: Efficient, robust, and practical datacenter CC via deployable INT," in *Proc. NSDI*, 2023, pp. 255–274.
[23] H. Lim, J. Kim, I. Cho, K. Jang, W. Bai, and D. Han, "FlexPass: A case for flexible credit-based transport for datacenter networks," in *Proc. 18th Eur. Conf. Comput. Syst.*, May 2023, pp. 606–622.
[24] P. Goyal, P. Shah, N. K. Sharma, M. Alizadeh, and T. E. Anderson, "Backpressure flow control," in *Proc. NSDI*, 2022, pp. 1–2.
[25] S. Hu et al., "Aeolus: A building block for proactive transport in datacenters," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Archit., Protocols Comput. Commun.*, Jul. 2020, pp. 422–434.
[26] RackSolutions.(2020). *How Many Servers Does a Data Center Have?*. [Online]. Available: https://shorturl.at/rpi2a
[27] Broadcom.(2019). *BCM56990 Series*. [Online]. Available: https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56990-series
[28] *Supplement to InfiniBand Architecture Specification Volume 1 Release 1.2.2 Annex A17: RoCEv2 (IP Routable RoCE)*, I. T. Association, InfiniBand, Beaverton, OR, USA, 2014.
[29] (2020). *InfiniBandTM Architecture Specification Volume 1 Release 1.4. (2020)*. [Online]. Available: https://cw.infinibandta.org/document/dl/8567
[30] C. Guo et al., "RDMA over commodity Ethernet at scale," in *Proc. ACM SIGCOMM Conf.*, Aug. 2016, pp. 202–215.
[31] M. J. Karol, S. J. Golestani, and D. Lee, "Prevention of deadlocks and livelocks in lossless backpressured packet networks," in *Proc. ToN*, vol. 11, 2003, pp. 923–934.
[32] V. Liu, D. Halperin, A. Krishnamurthy, and T. E. Anderson, "F10: A fault-tolerant engineered network," in *Proc. NSDI*, 2013, pp. 399–412.
[33] H. Lim, W. Bai, Y. Zhu, Y. Jung, and D. Han, "Towards timeout-less transport in commodity datacenter networks," in *Proc. 16th Eur. Conf. Comput. Syst.*, Apr. 2021, pp. 33–48.
[34] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. 10th ACM SIGCOMM Conf. Internet Meas.*, Nov. 2010, pp. 267–280.
[35] Y. Afek, S. Kutten, and M. Yung, "Memory-efficient self stabilizing protocols for general networks," in *Proc. Int. Workshop Distrib. Algorithms*, 1991, pp. 15–28.
[36] F. C. Gärtner, "A survey of self-stabilizing spanning-tree construction algorithms," in *Proc. EPFL*, 2003, pp. 1–12.
[37] M. A. Qureshi et al., "PLB: Congestion signals are simple and effective for network load balancing," in *Proc. ACM SIGCOMM Conf.*, Aug. 2022, pp. 207–218.
[38] A. Kabbani, B. Vamanan, J. Hasan, and F. Duchene, "FlowBender: Flow-level adaptive routing for improved latency and throughput in datacenter networks," in *Proc. 10th ACM Int. Conf. Emerg. Netw. Exp. Technol.*, Dec. 2014, pp. 149–160.
[39] Intel.(2020). *Intel Tofino2—A 12.9tbps P4-Programmable Ethernet Switch*. [Online]. Available: https://ieeexplore.ieee.org/document/9220636
[40] (2011). *802.11qbb. Priority Based Flow Control*. [Online]. Available: https://1.ieee802.org/dcb/802-1qbb/
[41] I. Cho, K. Jang, and D. Han, "Credit-scheduled delay-bounded congestion control for datacenters," in *Proc. ACM SIGCOMM*, 2017, pp. 1–11.
[42] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout, "Homa: A receiver-driven low-latency transport protocol using network priorities," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2018, pp. 221–235.
[43] M. Handley et al., "Re-architecting datacenter networks and stacks for low latency and high performance," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2017, pp. 29–42.
[44] P. X. Gao, A. Narayan, G. Kumar, R. Agarwal, S. Ratnasamy, and S. Shenker, "PHost: Distributed near-optimal datacenter transport over commodity network fabric," in *Proc. 11th ACM Conf. Emerg. Netw. Experiments Technol.*, Dec. 2015, pp. 1–12.
[45] R. Pan, B. Prabhakar, and A. Laxmikantha, "QCN: Quantized congestion notification," NVIDIA, USA, Tech. Rep. 1, 2007.
[46] J. He et al., "FasterMoE: Modeling and optimizing training of large-scale dynamic pre-trained models," in *Proc. 27th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, Apr. 2022, pp. 120–134.
[47] S. Rajasekaran, M. Ghobadi, and A. Akella, "CASSINI: Network-aware job scheduling in machine learning clusters," in *Proc. NSDI*, 2023, pp. 1403–1420.
[48] S. Rajasekaran, M. Ghobadi, G. Kumar, and A. Akella, "Congestion control in machine learning clusters," in *Proc. 21st ACM Workshop Hot Topics Netw.*, Nov. 2022, pp. 235–242.
[49] L. Kexin et al., "Pyrrha: Congestion-root-based flow control to eliminate head-of-line blocking in datacenter," in *Proc. NSDI*, 2025, pp. 379–405.
[50] J. Sonchack, A. J. Aviv, E. Keller, and J. M. Smith, "Turboflow: Information rich flow record generation on commodity switches," in *Proc. 13th EuroSys Conf.*, Apr. 2018, pp. 1–16.
[51] Broadcom.(2021). *Bcm88800 Traffic Management Architecture*. [Online]. Available: https://docs.broadcom.com/doc/88800-DG1-PUB
[52] Cisco.(2010). *Cisco Nexus 5548p Switch Architecture*. [Online]. Available: https://shorturl.at/hXEgN
[53] Juniper.(2017). *Understanding Cos Virtual Output Queues (VOQS) on Qfx10000 Switches*. [Online]. Available: https://shorturl.at/ReBJR
[54] P. C. W. Dally and L. Dennison, "Architecture of the Avici terabit switch/router," in *Proc. 6th Hot Interconnects*, 1998, pp. 1–12.
[55] T. Nachiondo, J. Flich, and J. Duato, "Buffer management strategies to reduce HoL blocking," *IEEE Trans. Parallel Distrib. Syst.*, vol. 21, no. 6, pp. 739–753, Jun. 2010.
[56] K. Liu et al., "Floodgate: Taming incast in datacenter networks," in *Proc. CoNEXT*, 2021, pp. 30–44.
[57] (2018). *Source Routing*. [Online]. Available: http://en.wikipedia.org/wiki/Source
[58] S. Hu et al., "Explicit path control in commodity data centers: Design and applications," in *Proc. NSDI*, vol. 24, 2015, pp. 2768–2781.
[59] E. Rosen, A. Viswanathan, and R. Callon, "Rfc3031: Multiprotocol label switching architecture," Cisco Syst., USA, Tech. Rep. 1, 2001.
[60] N. McKeown et al., "Openflow: Enabling innovation in campus networks," in *Proc. SIGCOMM Comput. Commun. Rev.*, Jul. 2008, pp. 69–74.
[61] Y. Xu et al., "Hashing design in modern networks: Challenges and mitigation techniques," in *Proc. ATC*, 2022, pp. 565–579.
[62] Z. Zhang et al., "Hashing linearity enables relative path control in data centers," in *Proc. ATC*, 2021, pp. 855–862.
[63] K. Qian et al., "Alibaba HPN: A data center network for large language model training," in *Proc. ACM SIGCOMM Conf.*, Aug. 2024, pp. 691–706.
[64] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proc. ACM SIGCOMM Conf. Data Commun.*, Aug. 2008, pp. 63–74.
[65] A. Greenberg et al., "VL2: A scalable and flexible data center network," in *Proc. ACM SIGCOMM Conf. Data Commun.*, Aug. 2009, pp. 51–62.
[66] A. Singh et al., "Jupiter rising: A decade of clos topologies and centralized control in Google's datacenter network," in *Proc. ACM Conf. Special Interest Group Data Commun.*, Aug. 2015, pp. 183–197.
[67] S. Ghorbani, Z. Yang, P. B. Godfrey, Y. Ganjali, and A. Firoozshahian, "DRILL: Micro load balancing for low-latency data center networks," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2017, pp. 225–238.
[68] R. Mittal et al., "Revisiting network support for RDMA," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2018, pp. 313–326.
[69] T. Benson, "Understanding data center traffic characteristics," *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 1, pp. 92–99, 2010.
[70] TPAW Group.(2021). *P4^{16} Portable Switch Architecture (PSA)*. [Online]. Available: https://p4lang.github.io/p4-spec/docs/PSA.pdf
[71] W. Peterson and D. Brown, "Cyclic codes for error detection," *Proc. IRE*, vol. 49, no. 1, pp. 228–235, Jan. 1961.
[72] (2022). *Introduction to HQOS*. [Online]. Available: https://shorturl.at/wRYYN
[73] (2021). *Hierarchical Class of Service Overview*. [Online]. Available: https://shorturl.at/yql0C
[74] Z. Zhang et al., "VPIFO: Virtualized packet scheduler for programmable hierarchical scheduling in high-speed networks," in *Proc. ACM SIGCOMM Conf.*, Aug. 2024, pp. 983–999.
[75] Z. Jiang et al., "MegaScale: Scaling large language model training to more than 10,000 GPUs," in *Proc. NSDI*, 2024, pp. 745–760.
[76] J. Ros-Giralt et al., "Designing data center networks using bottleneck structures," in *Proc. ACM SIGCOMM Conf.*, Aug. 2021, pp. 319–348.
[77] J. Ros-Giralt et al., "A quantitative theory of bottleneck structures for data networks," 2022, *arXiv:2210.03534*.
[78] S. Hu et al., "Tagger: Practical PFC deadlock prevention in data center networks," in *Proc. CoNEXT*, 2017, pp. 889–902.
[79] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *Proc. ACM Conf. Special Interest Group Data Commun.*, Aug. 2015 , pp. 123–137.