

FNCC: Fast Notification Congestion Control in Data Center Networks

Jing Xu

Institute of Computing Technology,
Chinese Academy of Sciences
University of Chinese Academy of
Sciences
Beijing, China
xujing@ncic.ac.cn

Zhan Wang

Institute of Computing Technology,
Chinese Academy of Sciences
Beijing, China
wangzhan@ncic.ac.cn

Fan Yang

Institute of Computing Technology,
Chinese Academy of Sciences
Beijing, China
yangfan@ncic.ac.cn

Ning Kang

University of Chinese Academy of
Sciences
Beijing, China
kangning18z@ict.ac.cn

Zhenlong Ma

University of Chinese Academy of
Sciences
Beijing, China
mazhenlong@ncic.ac.cn

Guojun Yuan

Institute of Computing Technology,
Chinese Academy of Sciences
Beijing, China
yuanguojun@ncic.ac.cn

Guangming Tan

Institute of Computing Technology,
Chinese Academy of Sciences
Beijing, China
tgm@ncic.ac.cn

Ninghui Sun

Institute of Computing Technology,
Chinese Academy of Sciences
Beijing, China
snh@ict.ac.cn

ABSTRACT

Congestion control plays a pivotal role in large-scale data centers, facilitating ultra-low latency, high bandwidth, and optimal utilization. Even with the deployment of data center congestion control mechanisms such as DCQCN and HPCC, these algorithms often respond to congestion sluggishly. This sluggishness is primarily due to the slow notification of congestion. It takes almost one round-trip time (RTT) for the congestion information to reach the sender. In this paper, we introduce the Fast Notification Congestion Control (FNCC) mechanism, which achieves sub-RTT notification. FNCC leverages the acknowledgment packet (ACK) from the return path to carry in-network telemetry (INT) information of the request path, offering the sender more timely and accurate INT. To further accelerate the responsiveness of last-hop congestion control, we propose that the receiver notifies the sender of the number of concurrent congested flows, which can be used to adjust the congested flows to a fair rate quickly. Our experimental results demonstrate that FNCC reduces flow completion time by 27.4% and 88.9% compared to HPCC and DCQCN, respectively. Moreover, FNCC triggers minimal pause frames and maintains high utilization even at 400Gbps.

KEYWORDS

Congestion Control, HPCC, RDMA, Data Center, PFC

ACM Reference Format:

Jing Xu, Zhan Wang, Fan Yang, Ning Kang, Zhenlong Ma, Guojun Yuan, Guangming Tan, and Ninghui Sun. 2024. FNCC: Fast Notification Congestion Control in Data Center Networks. In *Proceedings of arXiv Preprint (arXiv 2024)*. ACM, New York, NY, USA, 12 pages. <https://doi.org/XXXXXXX.XXXXXXX>

1 INTRODUCTION

As an evolving infrastructure, data centers cater to multiple tenants, concurrently running diverse applications [13] like web search [19], Hadoop [20], and machine learning [14]. The workloads of these applications comprise both throughput-sensitive large flows and latency-sensitive small flows simultaneously [28, 30], demanding ultra-high throughput and ultra-low latency, respectively. The substantial requirements imposed by these applications exert significant pressure on the data center networks to meet these diverse needs.

Consequently, RoCEv2 (RDMA over Ethernet version 2) has gained widespread adoption among data center designers [11] due to its kernel bypass and hardware offloading features, significantly improving network performance compared to traditional TCP/IP. However, network designers still face numerous challenges, with the Congestion Control (CC) mechanism being a notable concern [8, 10, 16–18, 31].

Especially over the past decades, the link speed has grown rapidly from 10, 50, 100, 200, and up to 400Gb/s [1]. Once congestion happens, if the control mechanism cannot respond timely,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

arXiv 2024, Online, 2024

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/18/06...\$15.00
<https://doi.org/XXXXXXX.XXXXXXX>

the offending traffic will aggressively occupy the limited buffer resources [10, 12, 23] and seriously affects the performance of latency-sensitive flows. Moreover, priority-based Flow Control (PFC) [2] is commonly used to prevent link-level packet loss when congestion control fails. PFC allows ethernet switches to avoid buffer overflows by forcing the direct upstream switch or host to pause data transmission. However, the pauses can trigger PFC deadlocks and PFC storms [12, 27], causing partial network entity paralysis.

Therefore, a fast congestion control mechanism is urgently needed to reduce the network queue length and avoid PFC triggering. An ideally fast congestion control solution includes ***accurate congestion detection, fast congestion notification, and accurate rate adjustment***.

HPCC (High Precision Congestion Control) [17] is now a state-of-the-art congestion control scheme in RoCEv2 networks compared with DCQCN [31], Timely [18], Swift [16], etc. The key idea behind HPCC is to leverage the precise link load information from the in-network telemetry (INT) information to compute an accurate flow rate. HPCC senders can quickly ramp up flow rates for high utilization or ramp down flow rates for congestion avoidance. Moreover, HPCC utilizes INT information to compute the actual in-flight data volume, enabling accurate congestion detection.

Unfortunately, HPCC is sluggish in notifying the sender when congestion happens. The INT information cannot be sent directly to the sender when congestion occurs in the intermediate switch. HPCC can only send the data packet with INT to the receiver first, and then the ACK packet passes INT back to the sender. It takes the sender at least one round-trip time (RTT) to slow down when congestion occurs and another RTT to resume acceleration when congestion subsides.

Building upon the aforementioned observations, this paper introduces a Fast Notification Congestion Control (FNCC) mechanism, an extension of HPCC, tailored for swiftly expanding data center networks. FNCC uses the ACK packet of the return path to carry the INT information of the request path, providing more timely and accurate INT for the sender than HPCC. The term “request path” refers to the route for the application’s data packet transmission, while the term “return path” pertains to the route for ACK packet transmission. Additionally, to further accelerate the speed of last-hop congestion control, the primary source of congestion [21], we propose that the receiver informs the sender of the number of concurrent congested flows. This information empowers the sender to calculate the target rate directly, facilitating the fast adjustment of flow rates to converge at the desired rate, bypassing a step-by-step approach.

Our contributions are summarized as follows:

- We uncover and delve into the delayed notification issue in existing congestion control mechanisms.
- We introduce the design of FNCC, a fast notification congestion control mechanism, and provide a thorough analysis of its design principles and detailed algorithm.
- We assess FNCC through simulations, comparing it with DCQCN and HPCC. FNCC demonstrates advantages such as lower queue size, fewer pause frames, higher utilization, and improved fairness. Moreover, in comparison to DCQCN

and HPCC, FNCC reduces Flow Completion Time (FCT) for actual data center workloads.

2 MOTIVATION

2.1 Rapid growing of link speed

The 2023 anniversary edition of the Ethernet Alliance Ethernet Roadmap [1] shows that the link speed has increased rapidly over the past decades. The link speed has quickly risen to 400 Gbps. 800Gbps and 1.6Tbps link speeds are now being developed.

Nevertheless, a critical challenge persists in this escalating era of faster link speeds. Figure 1a reveals a notable discrepancy between the switch buffer size and its corresponding capacity, as observed in the hardware trends of NVIDIA’s top data center switches [5]. The diminishing ratio points to a concern: the switch’s buffer scaling is not proportionately keeping pace with its increasing capacity. This mismatch poses a potential hurdle as it restricts the time in which the switch can absorb bursts of traffic, thereby amplifying the difficulty in managing end-to-end congestion control as these buffers will be quickly filled.

2.2 Longer queuing delay

Congestion control algorithms like DCQCN [4, 5] and HPCC [17] have been extensively researched and deployed in data centers. Although both congestion control algorithms have been proven to improve network congestion significantly, they still face the issue of sluggish response to congestion.

As depicted in Figure 2a, using HPCC as a reference, the mechanism of DCQCN closely resembles that of HPCC. If congestion arises at switch1 at time t, the congestion notification is not directly relayed to the sender. Instead, it persists in being forwarded to the receiver. Subsequently, the receiver generates an ACK packet and sends it back. This entire process is time-consuming. Any delayed response to congestion could lead to an escalation in congestion levels, resulting in a progressively deepening queue length. Figure 1b~d illustrates the fluctuations in queue length at the congestion point. In this test scenario, two elephant flows are transmitted to the same switch port at the line rate. Results show that the slow response of HPCC and DCQCN resulted in deeper queue lengths across different link rates. Refer to Section 5.1 for a detailed experimental environment setup.

2.3 Higher risk to trigger PFC pauses

PFC (Priority-based Flow Control) is the most widely used flow control method in data centers to avoid packet loss. In instances where congestion arises within the downstream device’s lossless queue, it promptly sends a pause frame to the upstream device to halt the flow. However, it is preferable to minimize the number of pause frames triggered by PFC, as they can affect flow fairness [31], lead to broadcast storms, and result in deadlocks [11].

Nevertheless, the long queues caused by slow congestion control will trigger PFC more frequently. Figure 3 illustrates the count of pause frames at the congestion point. The results indicate a higher number of pause frames generated by HPCC and DCQCN algorithms at both 200Gbps and 400Gbps rates. The experimental environment setup is the same as Section 2.2.

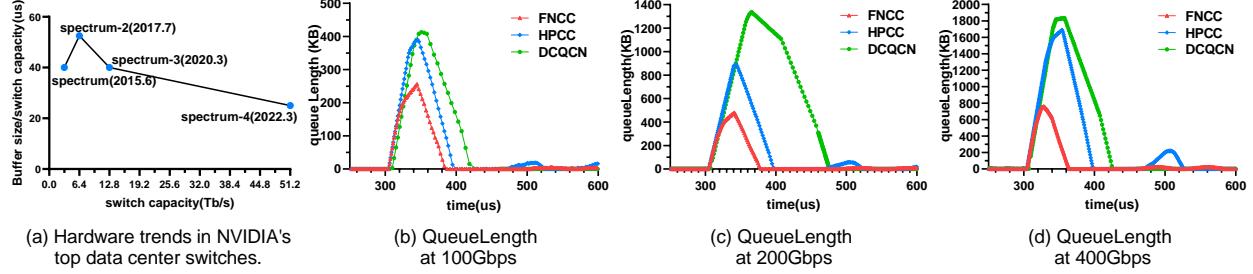


Figure 1: (a), Hardware trends in NVIDIA’s top data center switches. Switch capacity and link speeds have grown rapidly, but buffer sizes can not keep up with switch capacity growth. (b)~(d), Deeper queue lengths are observed across different link rates when applying HPCC and DCQCN than FNCC.

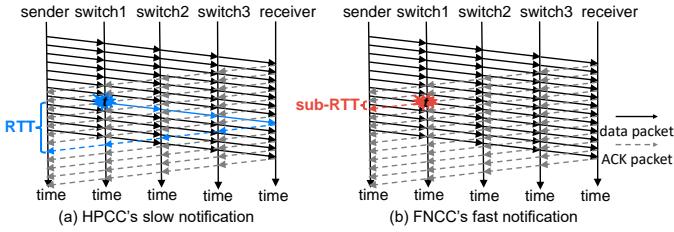


Figure 2: Notification scheme of HPCC and FNCC. Assuming congestion arises at switch1 at time t, the HPCC’s congestion information is not directly relayed to the sender.

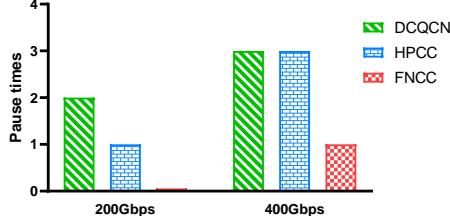


Figure 3: The count of pause frames at the congestion point. The results indicate a higher number of pause frames generated by HPCC and DCQCN algorithms than FNCC at both 200Gbps and 400Gbps rates.

2.4 Most flows are short

Studies by [10, 19] analyze the flow size distribution observed in the Facebook Hadoop cluster and webSearch workload. As the link speed increases, a growing percentage of traffic is swiftly processed relative to the Round-Trip Time (RTT). Moreover, it is projected that within the coming decade, a significant portion of Facebook Hadoop traffic might be completed within a single round trip.

These short-duration flows often evade control by RTT-based congestion mechanisms due to their smaller size compared to the Bandwidth Delay Product (BDP). Additionally, these flows are highly sensitive to queuing performance. Even a single erroneous or delayed congestion control decision can result in tens of microseconds of tail queuing.

2.5 Summary

Based on the above discussion and analysis, it becomes evident that data centers are in dire need of a fast congestion control scheme.

Such a scheme is imperative to curtail the rapid filling of buffers, mitigate longer queue lengths, reduce the occurrence of pause frames, and ensure optimal performance for latency-sensitive flows.

3 FNCC DESIGN

In this section, we propose a Fast Notification Congestion Control (FNCC) mechanism, which builds upon the foundation of HPCC. While HPCC boasts accurate congestion detection and rate regulation, it suffers from sluggish notification. FNCC addresses this issue by incorporating the following core designs:

- FNCC leverages the ACK packet of the return path to convey the INT information of the request path, which offers the sender more timely and precise INT information.
- The concurrent congested flows are informed to the sender so that it can quickly adjust the congested flow rate to a fair one when last-hop congestion occurs.

Concretely, Figure 4 illustrates the framework of FNCC. Unlike HPCC, in FNCC, the switch is only responsible for inserting INT information into the ACK packet. To further speed up the last-hop congestion control, FNCC’s receiver notifies the sender of the number of concurrent congested flows. FNCC’s sender can quickly regulate the flow to a fair rate.

3.1 Design Basis

The proposal of the FNCC is based on the following observations.

Observation 1. The ACK packets, traversing the congestion point, reach the sender faster.

In the scenario depicted in Figure 2a, suppose switch1 experiences congestion at the moment t , HPCC does not promptly relay the existing INT (i.e., queue information) back to the sender. Its notification process consumes almost one RTT, rendering it notably slow.

Upon observing that ACK packets navigating through the congestion point reach the sender faster (as illustrated in Figure 2b), FNCC capitalizes on this observation. Suppose switch1 experiences congestion at the moment t , FNCC directly embeds INT into ACK packets on the return path, enabling fast notification of congestion situations to the source in less than an RTT.

Observation 2. The application’s data packet and its ACK packet’s forwarding path can be identical.

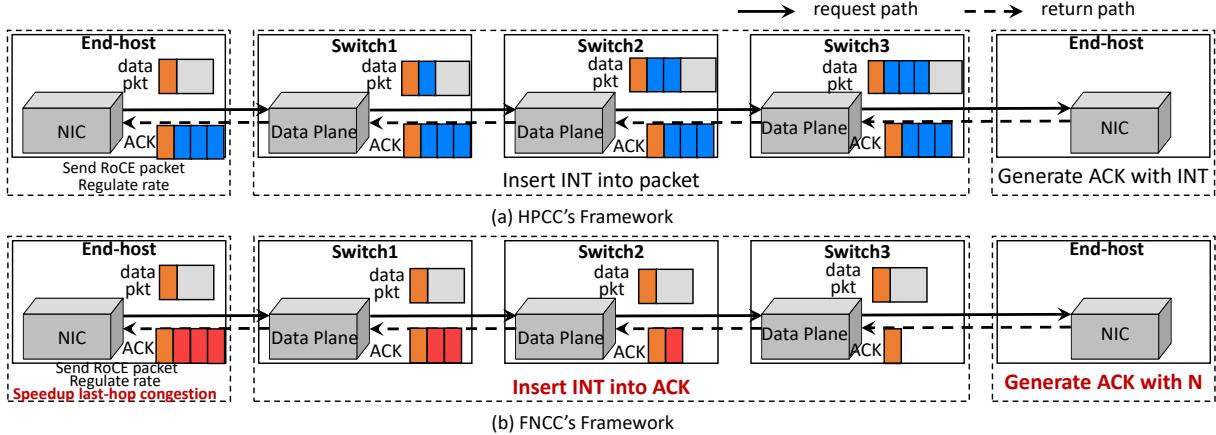


Figure 4: The framework of HPCC and FNCC design. HPCC needs to add In-Network Telemetry (INT) after each data packet. The target end-host generates ACK containing all INTs and sends them back to the sender. In FNCC, the switch only adds INT to the ACK. The ACK can reach the sender faster and can provide more timely INT information. To further speed up the last-hop congestion control, FNCC’s receiver notifies the sender of the number of concurrent congested flows (N). FNCC’s sender can quickly regulate the flow rate to a fair rate.

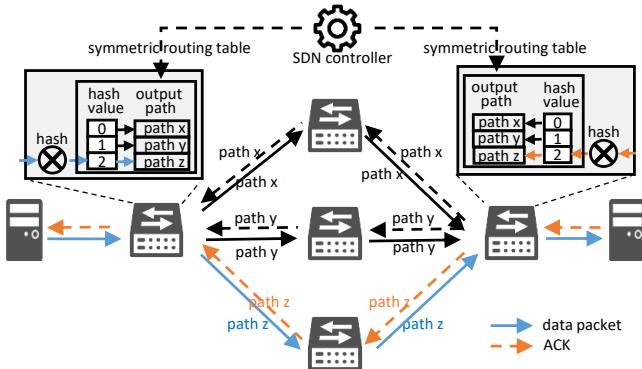


Figure 5: Symmetric route table. When the SDN controller builds the routing table, it should ensure that the data packet forwarding paths are in the same order as the ACK packet forwarding paths. Since the data packet and its corresponding ACK packet share the same five-tuple values, they will possess identical hash values. With a symmetric routing table in place, these packets will select the same path.

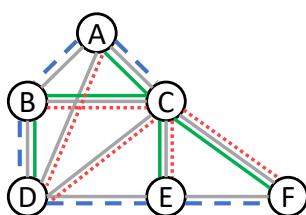


Figure 6: Three example spanning trees in a small network. The links are shown in gray, while the spanning trees are highlighted in color. Each spanning tree exhibits unique paths between any nodes in the network.

There are various topologies with path diversity, such as Fat-Tree, Jellyfish [22], Dragonfly [15], and so on. In these topologies, packets and ACK packets may choose different paths during multipath selection. However, there are several methods to address this issue.

One method is establishing a symmetric routing table where the data packet forwarding paths are in the same order as the ACK packet forwarding paths, as illustrated in Figure 5. The Equal-Cost Multipath (ECMP) protocol is commonly utilized for load balancing among paths with identical costs. ECMP employs a hash function that operates on the five-tuple {source IP, destination IP, source port, destination port, protocol type}, using operations such as addition, XOR, etc., to determine the path followed by the packet. Since the data packet and its ACK packet share the same five-tuple values, they will possess identical hash values. With a symmetric routing table in place, these packets will select the same path.

Another method is constructing multiple spanning trees, each of which has a unique path between any two nodes in the network, as shown in Figure 6. This approach draws inspiration from TCP-Bolt [24], which is designed to prevent routing paths from forming loops and causing deadlocks. In our research, we employ this method to ensure that data packet and ACK packet’s path are identical.

Additionally, there are some topologies that inherently lack path diversity, where the communication path between any two nodes is unique, such as standard butterfly topology and star topology. In these topologies, the data packet and its ACK packet’s forwarding path are always the same.

Observation 3. Application traffic, such as RC RDMA Write, may induce congestion along the request path, while their corresponding ACK packets do not contribute to congestion along the return path.

Application traffic, like reliable connected RDMA Write, often carries a substantial payload, potentially reaching up to two gigabytes, rendering it particularly susceptible to network congestion. However, their ACK packets are small, containing only a few dozen bytes of data, which are used to ensure that application traffic is

Algorithm 1 CP behavior

```

1: // Port's input engine
2: if Received ACK then
3:   ack.input_port_num ← current port num;
4: end if
5: Forward to Ingress pipeline;
6: // Port's output engine
7: if Received ACK then
8:   INT_Info ← All_INT_Table[ack.input_port_num];
9:   Insert INT_Info to ACK packet;
10: end if
11: Forward to Egress pipeline;

```

received reliably and do not serve to transmit application data. Therefore, the ACK packet does not cause network congestion in the opposite direction. As a result, congestion control is mainly for the congestion in the direction of application data packets transmission.

Building upon this observation, FNCC is designed to insert the INT of the output port of the application data packets. For instance, if the data packets arrive at the switch with input port i and output port j , then its ACK's input port is j and output port is i , FNCC will insert INT of the output queue of switch port j into the ACK packet.

Observation 4. *When last-hop congestion occurs, all flows eventually converge to a rate determined by dividing the bandwidth of the congested port by the number of flows. Moreover, it is easy for the receiver to know the number of concurrent flows.*

Equation 1 illustrates the variation of queue length $q(t)$ over time at the bottleneck, where $W_i(t)$ represents the sending window size of flow i , RTT denotes the round-trip time, $W_i(t)/RTT$ represents the input rate of flow i , and $Bandwidth$ stands for the output bandwidth at the bottleneck. As the bottleneck eventually stabilizes, Equation 1 reaches zero. At this point, the sum of the input rates of all flows equals the output bandwidth, and all flows achieve equal rates, as illustrated in Equation 2.

$$\frac{dq(t)}{dt} = \sum_i^N \frac{W_i(t)}{RTT} - Bandwidth \quad (1)$$

$$\sum_i^N \frac{W_i(t)}{RTT} = Bandwidth \quad (2)$$

$$W_i(t) = \frac{Bandwidth * RTT}{N} \quad (3)$$

This insight encourages us to regulate the sending window size $W_i(t)$ more precisely at the sender, provided we are aware of the number of concurrent flows N , as illustrated in Equation 3. Fortunately, the transport layer at the receiver possesses information about the number of concurrencies, i.e., RDMA QP connections.

3.2 Algorithm

The FNCC algorithm comprises three main components: the switch (Congestion Point, CP), the sender (Reaction Point, RP), and the receiver (ACK Generation Point). In the following sections, we will delve into each of them in detail.

3.2.1 CP Algorithm at switch. HPCC relies on data packets and ACK to share information, where INT information is inserted into data packets and then forwarded by ACKs to the sender. In FNCC, it is inserted only in the ACKs. In particular, the ACK carries INT information for the request path.

Algorithm 1 outlines the behavior of the switch. When the input engine of the switch port receives an ACK packet, it records the $ack.input_port_number$ as the current port number (Line 3) and forwards it to the ingress pipeline (Line 5). If the incoming packet is a regular data packet, it is directly forwarded to the ingress pipeline. The ingress pipeline then executes operations such as protocol checking, traffic classification, and routing.

As the ACK packet reaches the output engine, the All_INT_Table is looked up with the $ack.input_port_number$ as the index. All_INT_Table encapsulates the INT information for all ports, and INT information contains the exact specific details as in HPCC, refer to Figure 7. This process retrieves the INT_Info value for the request path (Line 8), which is then inserted into the ACK packet (Line 9) before forwarding it to the egress pipeline. In the case of a regular data packet, it is directly forwarded to the egress pipeline (Line 11). Subsequently, the egress pipeline performs operations such as flow control and quality of service (QoS) scheduling.

Note that there is no mention of multiple priority queues per port for clarity of description.

3.2.2 RP Algorithm at sender. The FNCC's Reaction Point (RP) algorithm mirrors HPCC, employing a window-based scheme to regulate the number of in-flight bytes. Similarly, FNCC's congestion detection is also grounded in monitoring the in-flight bytes. The subsequent sections will offer a comprehensive overview of FNCC's congestion detection, rate regulation, and approach to addressing last-hop congestion scenarios.

Congestion detection:

$$U_j = \frac{I_j}{Bandwidth_j * RTT} \quad (4)$$

The congestion coefficient, U_j , indicates the level of congestion. When U_j exceeds a threshold value, η (where η is nearly one, e.g., 0.95), it signifies that link j is congested. The calculation of U_j is based on Equation 4, where I_j represents all in-flight bytes for link j and is the sum of queue length and bytes sent between two ACKs, $Bandwidth_j$ is the bandwidth of path j , and RTT is the round-trip time between the sender and receiver.

Rate regulation:

Both HPCC and FNCC perform rate adjustments based on a window W . As depicted in Equation 5, the variation in W is contingent on the congestion coefficient U_j of the most congested link. η is a constant close to 1, e.g., 95%, and W_{AI} represents an additive increase (AI) component designed to ensure fairness, which is kept very small.

$$W = \frac{W}{\max_j(U_j)/\eta} + W_{AI} \quad (5)$$

To achieve a fast reaction without overreaction, both HPCC and FNCC integrate the per-ACK and per-RTT strategies. The key idea is to introduce a reference window size W^c , a runtime state updated per RTT. The sender cautiously updates its window size solely upon receiving the ACK of the first packet sent with the current W^c . At

Algorithm 2 RP's last-hop congestion speedup behavior

```

1:  $U_{\max} \leftarrow 0;$ 
2:  $\text{hop} \leftarrow 0;$ 
3: function HOP_DETECTION()
4:   for  $j = 0$  to  $nHop$  do
5:     if  $U_j > U_{\max}$  then
6:        $U_{\max} \leftarrow U_j;$ 
7:      $\text{hop} \leftarrow j;$ 
8:   end function
9: function UPDATERWc(ack)
10:  HOP_DETECTION();
11:  if  $\text{hop} = \text{last hop}$  &  $U_{\max} > \alpha$  then
12:     $W^c \leftarrow \frac{B \times \text{RTT} \times \beta}{\text{ack.N}};$ 
13:   $U_{\max} \leftarrow 0;$ 
14:   $\text{hop} \leftarrow 0;$ 
15: end function

```

this juncture, the sender updates W^c to the current window size W . Within the round-trip time, the sender updates the window size based on per-ACK information to adjust W , see Equation 6.

$$W = \frac{W^c}{\max_j(U_j)/\eta} + W_{AI} \quad (6)$$

Fast response to last-hop congestion:

Inspired by our previously mentioned Observation 4, we propose a **last-hop** congestion speedup (LHCS) algorithm. When congestion is detected in the last hop, the sending window of these flows is directly set to the final convergence value, offering a more agile response to the last congestion.

In Algorithm 2, Lines 1 to 7 are dedicated to determining the location of congestion occurrence and assessing the congestion level U_{\max} . Congestion is identified when U_{\max} is greater than α (Line 11). To prevent excessive sensitivity to the network state, the value of α is set slightly larger than one (e.g., 1.05). In the case of congestion occurring in the last hop, the final convergence value is assigned to W^c (Line 12), where B represents last-hop bandwidth, the value of β is slightly smaller than one (e.g., 0.9) for draining the congested queue, N is the number of concurrent flows. Note that the N value is carried in the ACK packet. We will discuss it next.

3.2.3 ACK Generation at receiver. The ACK is generated by the receiver in FNCC. To help the sender converge to the target rate as soon as possible, the receiver directly informs the sender of the number of concurrent congested flows, specifically by writing the number of RDMA QP connections into the ACK packet. We define N as the number of concurrent flows occupying 16 bits and supporting a maximum of 64k connections, which is sufficient to meet the requirements of the data center [29]. As the ACK traverses the switch, the switch inserts INT information after the packet header of the ACK, as depicted in Figure 4b. The complete ACK format is detailed in Figure 7.

It is worth noting that our design supports a cumulative ACK scheme (one cumulative ACK for every m consecutively received packets) at the receiver.

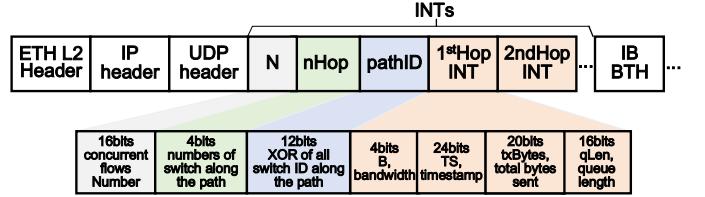


Figure 7: The ACK format of FNCC.

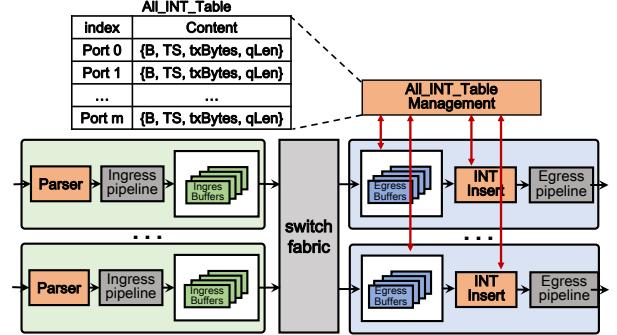


Figure 8: FNCC switch architecture.

4 IMPLEMENTATION

In this section, we explore the feasibility of deploying the CP algorithm at the switch while implementing the RP algorithm and ACK generation at the host.

4.1 CP algorithm at switch

We give the FNCC switch architecture, as shown in Figure 8. The key components of the CP implementation include:

Parser module recognizes regular data traffic and ACK packets. Once an ACK packet is recognized, record the input port number into the metadata of the ACK packet. As mentioned above, the input port number of the ACK in the return path is the output port number of application traffic in the request path.

All_INT_Table Management module is a critical system component tasked with maintaining the INT information of all output ports throughout the switch. To allow efficient lookup of the INT associated with an ACK packet, the INT is stored in a table, which is an array indexed by port number, ranging from 0 to the number of ports on the switch minus 1. This module will update *All_INT_Table* periodically.

INT_Insert module is responsible for recognizing regular traffic and ACK packets and inserting INT information into the ACK packet. The INT information is obtained by using the metadata *ack.input_port_num* to look up the *All_INT_Table* table. Subsequently, the processed ACK is forwarded to the Egress pipeline.

4.2 RP algorithm and ACK generation at host

Compared to HPCC, FNCC's RP algorithm at the sender host incorporates congestion hop detection and W^c update functions, with low computational costs and without requiring additional buffer resources. Meanwhile, at the receiver host, FNCC only inserts the

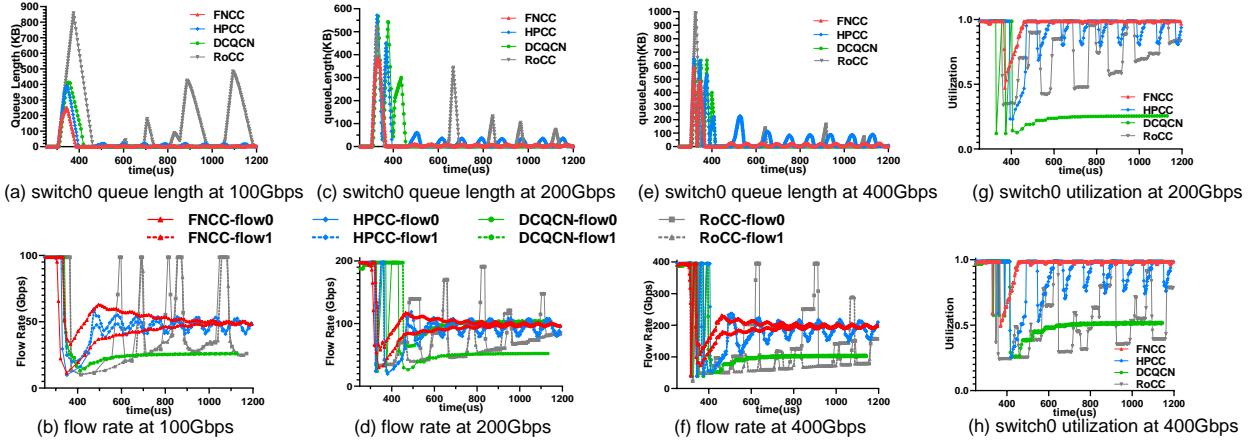


Figure 9: (a)~(b), Queue length's variation at congestion point and flow rate's variation at senders at 100 Gbps. FNCC is the first to detect congestion and slow down. The queue length at FNCC's congestion point is the lowest, and the two elephant flows of FNCC are the first to converge to a fair rate. (c)~(f), Queue length's variation at congestion point and flow rate's variation at senders at 200Gbps and 400Gbps. FNCC is also the fastest to react. (g)~(h) FNCC can maintain the highest utilization.

number of concurrent flows, N , into the ACK packet, also incurring minimal computational overhead.

4.3 Hardware feasibility

FNCC requires a memory size proportional to the number of physical output queues in the switch to store the *All_INT_Table* table. Each entry in the table is 64 bits width {4 bits B, 24 bits TS, 20 bits txBytes, 16 bits qLen}. Assuming a 64-port switch, the size of the INT table is 4k bits, which is small compared to the switch's shared buffer size (3.2MB, 6.4MB, 12.8MB, 51.2MB, etc.).

Implementing FNCC in switches is both simple and cost-effective. It only requires additional resources to implement its module's design, which is abundantly available in today's commodity switches [3, 6, 26]. The process of looking up the INT table and inserting INT into ACK requires only $O(1)$ time. With an increasing number of switches supporting programmable and open data planes, we anticipate that the *All_INT_Table* table lookup and insertion procedures can be executed at the data plane at a line rate.

FNCC's implementation in hosts is also straightforward and cost-effective. At the sender host, FNCC incorporates congestion hop detection and W^c update functions. At the receiver host, FNCC only inserts the number of concurrent flows. Those introduce minimal computational and storage overhead.

5 PERFORMANCE EVALUATION

In this section, we evaluate the performance of FNCC through the INET model suite on the OMNeT++ [7] simulator. We use different traffic patterns and topologies to assess the effectiveness of FNCC, utilizing traffic workloads derived from publicly available data center traffic traces.

All links in all topologies are running at the 100Gbps rate with $1.5\mu s$ propagation delay. To eliminate the impact of RoCEv2 QoS, packets from all sources are transferred on the same service level (SL) or virtual lane (VL). The Maximum Transmission Unit (MTU) is

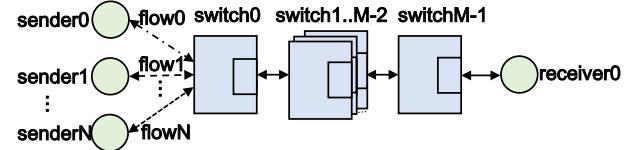


Figure 10: Typical dumbbell topology.

set to 1518 bytes. We implement several state-of-the-art solutions, including DCQCN, HPCC, and RoCC. DCQCN has been implemented in Mellanox commercial network adapters and switches [4, 5]. HPCC is already employed in Alibaba's predictable network [17]. RoCC is a typical switch-driven congestion control proposed by Cisco. At the switch, RoCC uses the proportional-integral (PI) controller to compute the fair rate. DCQCN and RoCC's parameters are assigned to the default values recommended in research [25, 31].

We conduct five types of experiments to evaluate FNCC:

- Response speed on micro-benchmarks
- Robustness against higher line rates
- Fairness over multiple flows
- Gains across various congestion scenarios
- Large-scale simulation resembling a real data center

5.1 Response speed on micro-benchmarks

Typical scenario. As shown in Figure 10, we build a typical dumbbell topology network. The experiment includes three switches ($M = 3$) and two senders ($N = 2$). The first elephant flow, flow0, starts at time 0 at 100Gbps. And a new elephant flow, flow1, joins at 300us at 100Gbps. To prevent packet loss, we enable PFC and set the PFC threshold to 500KB. We monitor the rate changes of the two elephant flows at the sender to visualize the congestion response speed. Moreover, we also observe the queue length of the congestion port of switch1.

FNCC has a faster response to congestion. Figure 9b illustrates the behavior of RoCC, DCQCN, HPCC, and FNCC. FNCC is the first to slow down at $300\mu s$, followed by HPCC at $330\mu s$, DCQCN at $346\mu s$, and RoCC at $370\mu s$. It is demonstrated that our scheme of utilizing the ACK packet carrying INT information is not only theoretically faster but also empirically.

FNCC has faster convergence to fair rates. When congestion is relieved at the switch, from Figure 9b, the two elephant flows applying FNCC have faster and smoother convergence to fair rates. However, the two flows applying HPCC converge to a fair rate jitterily. When using DCQCN, the two flows are slow to recover. And RoCC is hard to converge at the microsecond level.

FNCC has the shallowest queue depth. From Figure 9a, due to the advantage of fast notification in FNCC, it can quickly reduce the sender's rate and minimize the queue length peak at congestion points.

5.2 Robustness against higher line rates

One of the goals of the FNCC is to remain robust in the face of increasing data center line rates. We repeat the simulation in Section 5.1, increasing the link capacity from 100Gbps to 200Gbps and 400Gbps. It is difficult for the switch to maintain a small queue.

Low queue length. As shown in Figure 9c and Figure 9e, FNCC always maintains the shortest queue length at the congestion point compared with RoCC, DCQCN and HPCC. FNCC is always the first to slow down and converge to a fair rate, as shown in Figure 9d, 9f. This proves that the fast notification scheme of FNCC is robust even at high rates.

High utilization. From Figure 9g and Figure 9h, FNCC consistently maintains the highest utilization. Once congestion is relieved, FNCC promptly updates senders with the latest queue length at the congestion point, enabling them to promptly increase speed to avoid underutilized links.

Few pause frame. We measured the number of pause frames generated at the congestion point using FNCC, HPCC, and DCQCN at 200Gbps and 400Gbps. From Figure 3, it can be seen that the number of pause frames generated when applying the FNCC algorithm is the lowest at both 200Gbps and 400Gbps rates.

5.3 Fairness over multiple flows

To verify the fairness of FNCC, we adopt a dumbbell topology as shown in Figure 10. The experiment includes four senders ($N = 4$). They send long-lived flows to receiver0. Every 100 milliseconds, a new sender initiates a new flow and subsequently exits in sequence. Figure 13e demonstrates the fair throughput of these four flows. The experimental results prove that FNCC can achieve good fairness even in a short time scale.

5.4 Gains across various congestion scenarios

We extended the experiment with congestion occurring in the middle hop and the last hop, both with a link rate of 100Gbps. The experimental topologies are shown in Figure 11.

5.4.1 Theoretical Analysis. In Figure 12, HPCC inserts INTs at switches along the request path at moments t1, t2, and t3. The sender receives an ACK message with INT and initiates rate adjustment at the moment t8. On the other hand, FNCC inserts INTs at the switch

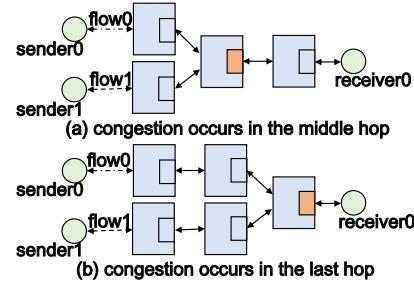


Figure 11: Topology: Congestion occurs in different locations.

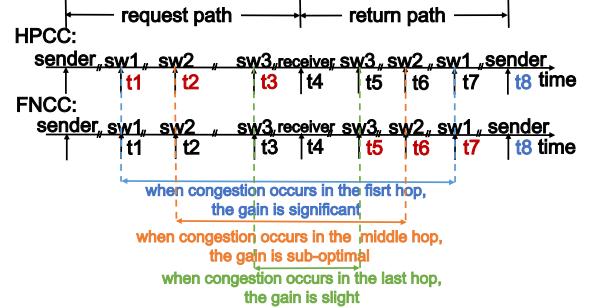


Figure 12: Theoretical model.

along the return path at moments t5, t6, and t7. The INT values inserted at t5, t6, and t7 are more timely, allowing congestion or under-utilization to be captured earlier and more promptly when the sender performs rate adjustment at t8.

However, the rate adjustment is determined by the INT of the most congested link, as discussed above. In the scenario where congestion occurs on sw1 (switch1), FNCC exhibits an even more pronounced advantage over HPCC due to the significant time difference between t7 and t1. If congestion arises on sw2 (switch2), the time difference between t6 and t2 is relatively large, resulting in FNCC having a sub-optimal advantage over HPCC. In the case of congestion on sw3 (switch3), where the time difference between t5 and t3 is not very prominent, the FNCC advantage over HPCC is marginal. However, for the last-hop congestion scenario, we propose the “last-hop congestion speedup algorithm” in Section 3.2.2. The sender can quickly make precise rate adjustments.

5.4.2 Experimental results. According to Figure 13, while maintaining a higher link utilization, FNCC reduces queue depth by 37.5% in first-hop congestion, 29.5% in middle-hop congestion, 8.4% in last-hop congestion without enabling Last-Hop Congestion Speedup (FNCC without LHCS), and 38.5% in last-hop congestion with enabling Last-Hop Congestion Speedup (FNCC with LHCS). We also compared the variations of flow rate when using HPCC, FNCC without LHCS, and FNCC with LHCS. The experimental results are shown in Figure 13d. At $300\mu s$, the addition of flow1 causes the last-hop switch to become congested, and the FNCC with last-hop speedup can be quickly adjusted to a fair rate $\ast \beta$. The value of β is slightly smaller than one (e.g., 0.9) for draining the congested queue and maintaining high utilization. After $450\mu s$, the last-hop congestion disappears, and the rate of the two flows is no longer constrained by the last-hop speedup Algorithm 2, but is controlled by Equation 6 and gradually adjusted to a fair rate. To facilitate

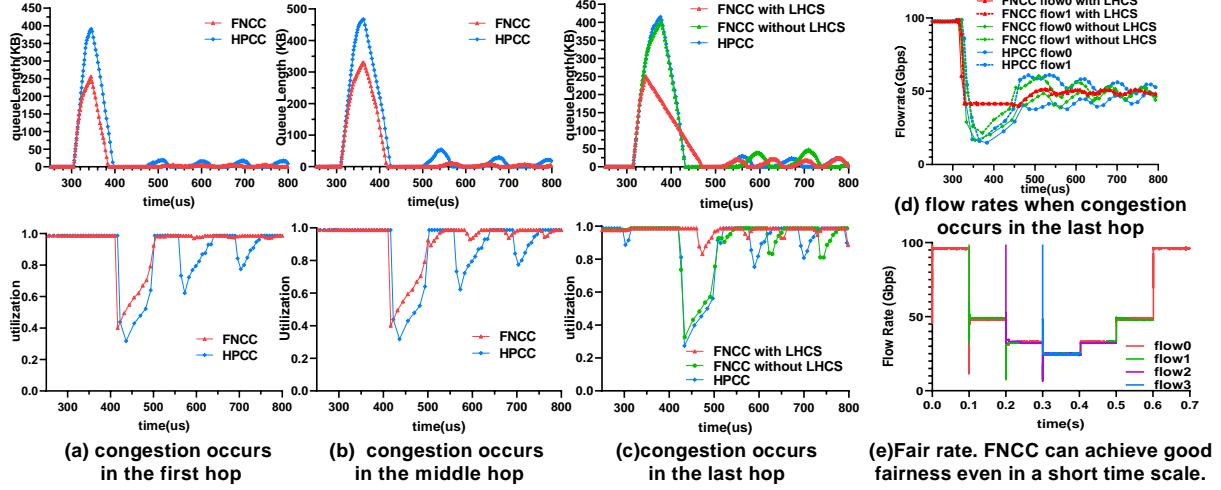


Figure 13: (a)~(c), FNCC maintains lower queue depths while guaranteeing higher link utilization. FNCC reduces queue depth by 37.5% in the first-hop congestion, 29.5% in the middle-hop congestion, 8.4% in the last-hop congestion without enabling Last-Hop Congestion Speedup (LHCS), and 38.5% in the last-hop congestion with LHCS. (d) FNCC with LHCS can sensitively detect the last-hop congestion and quickly adjust to a fair rate $\ast \beta$. The value of β is slightly smaller than one (e.g., 0.9) for draining the congested queue. (e), FNCC can achieve good fairness.

readability, we have omitted the comparison with DCQCN and RoCC. From the results shown in Figure 9, it is evident that HPCC outperforms DCQCN and RoCC.

5.5 Large-scale simulations

We use large-scale simulations to evaluate FNCC and compare it with DCQCN [31] and HPCC [17] regarding FCT slowdown. “FCT slowdown” means a flow’s actual FCT normalized by its ideal FCT when the network only has this flow. Due to the poor performance of RoCC at the microsecond level and the fact that it has not been used in practice, we overlooked the comparison with it. We use a three-level fat-tree topology ($k = 8$) with 128 servers. Each server has a single 100Gbps NIC connected to a single top-of-rack (ToR) switch. The capacity of each link between core and aggregation switches, aggregation switches and ToR switches are all 100Gbps (1:1 oversubscription). All links have a 1.5 μ s propagation delay. We implement ECMP on the ToR and aggregation switches to distribute the load across the links equally. We use traffic loads derived from two publicly available data center traffic distributions consisting of throughput-sensitive large flows (WebSearch traffic) and latency-sensitive small flows (FB_Hadoop traffic) [19]. We run our simulations using 50% average link load levels and repeat each experiment 5 times. The FCTs slowdown we present are the average values of the five sets of results.

Figure 14 and Figure 15 respectively show the average, medium, 95th percentile, and 99th percentile FCT slowdown for WebSearch traffic and FB_Hadoop traffic at 50% average load.

FCT slowdown. Based on the average percentile FCT, as shown in Figure 14a and Figure 15a, FNCC outperforms DCQCN and HPCC for all the flow sizes.

FNCC is beneficial to long flows. Our fast notification mechanism not only swiftly relieves congestion but also prevents low

link utilization. Once congestion is alleviated, FNCC can quickly increase speed to avoid under-utilization of links, which is especially beneficial for long flows. As depicted in Figure 14a and Figure 14b, for flows larger than 1 MB, FNCC can reduce congestion by up to 12.4% compared to HPCC and 42.8% compared to DCQCN.

FNCC is also beneficial to short flows. Since FNCC has a fast notification mechanism and keeps the shallowest queues, it is beneficial to short flows mostly. HPCC and DCQCN schemes endure slow notification, leading to slow congestion relief, so they cannot keep the latency low. Figure 15c shows that for the flows shorter than 100KB, FNCC achieves a much lower FCT slowdown than other schemes at the 95th percentile. Specifically, FNCC has a 27.4% and 88.9% reduction in FCT slowdown than HPCC and DCQCN, respectively.

6 RELATED WORK

Congestion control is an enduring topic, and here we briefly introduce some closely related work.

Switch-driven notification. RoCC [25] and BFC [10] are two typical mechanisms whose notification entity is located at the switch. RoCC leverages a proportional-integral controller to compute the fair rate of active flows. BFC achieves accurate per-hop per-flow flow control. However, RoCC relies on at least nine parameters to tune, and requires millisecond-level delays to converge, see Figure 9. BFC requires large buffer resources, as it necessitates setting up separate queues for each flow, which poses a challenging implementation in switches.

Receiver-driven notification. To prevent the last-hop congestion, certain proposals advocate relocating the notification entity to the receiver side. ExpressPass [9] avoids congestion by directly dispatching credits to senders. However, managing distinct timers on RDMA NICs to orchestrate credit pacing for each flow poses

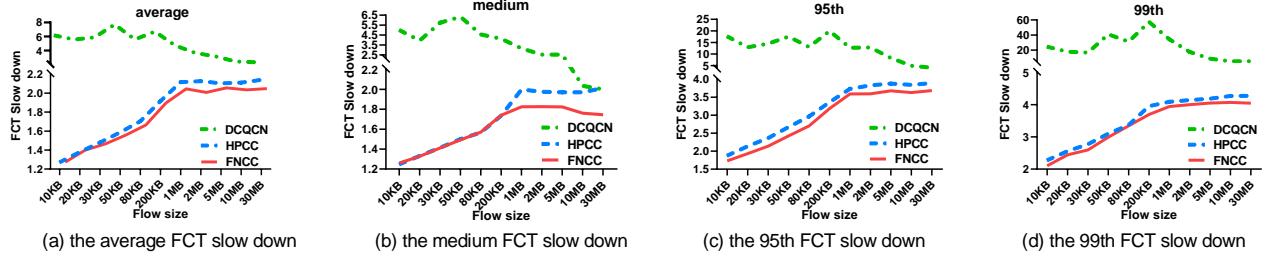


Figure 14: The average, medium, 95th, and 99th FCT slowdown of DCQCN, HPCC, and FNCC with WebSearch (50% avg. load). FNCC always has the lowest tail latency. Especially for flows larger than 1 MB, FNCC has a 12.4% and 42.8% reduction in medium FCT slowdown than HPCC and DCQCN, respectively.

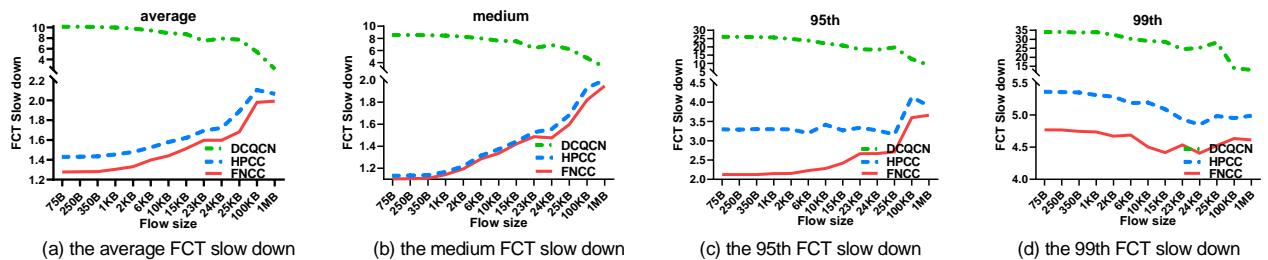


Figure 15: The average, medium, 95th, and 99th FCT slowdown of DCQCN, HPCC and FNCC with Hadoop (50% avg. load). For flows shorter than 100KB, FNCC has a 27.4% and 88.9% reduction in 95th FCT slowdown than HPCC and DCQCN, respectively.

challenges. Homa [19] dynamically schedules packets using priority queues. Nevertheless, due to the likelihood of packet reordering with the packet spraying scheme, more robust support is needed in RDMA networks.

End-to-end notification. End-to-end based solutions like DCQCN [31], Timely [18], Swift [16], and HPCC [17], etc., rely on rate regulation through congestion feedback mechanisms like ECN, RTT, and INT. However, a shared drawback among these approaches is their delayed reaction to congestion. We leverage HPCC's benefits of accurate flow rate updates. To address its slow congestion notification, we introduce FNCC, offering more timely and precise INT to the sender.

7 CONCLUSION

We discuss the limits of existing congestion control methods and propose FNCC, a novel fast notification congestion control mechanism. Compared to HPCC, FNCC can provide a more timely and accurate INT for the sender by utilizing the ACK packet of the return path to carry INT information of the request path. Additionally, FNCC further accelerates the speed of last-hop congestion control, by promptly informing the sender of the number of concurrent congested flows and adjusting the congested flow to a fair rate. Experimental results show that FNCC not only achieves lower queue size, fewer pause frames, and higher utilization even at 400Gbps, but also reduces FCT for real data center workloads compared with DCQCN and HPCC.

REFERENCES

- [1] [n. d.]. 2023 Ethernet Roadmap - Ethernet Alliance. <https://ethernetalliance.org/technology/ethernet-roadmap/>
- [2] [n. d.]. 802.1QBB – Priority-based Flow Control |. <https://1.ieee802.org/dcb/802-1qb/>
- [3] [n. d.]. NetFPGA. <https://netfpga.org/NetFPGA-SUME.html>
- [4] [n. d.]. NVIDIA ConnectX NICs. <https://www.nvidia.com/en-us/networking/ethernet-adapters/>
- [5] [n. d.]. NVIDIA Spectrum Ethernet Platform. <https://www.nvidia.com/en-us/networking/products/ethernet/>
- [6] [n. d.]. Trident3-X7 / BCM56870 Series. <https://www.broadcom.com/products/ethernet-connectivity/switching/stratixgs/bcm56870-series>
- [7] 2023. OMNET++ Discrete Event Simulator. <https://omnetpp.org/>
- [8] Serhat Arslan, Yuliang Li, Gautam Kumar, and Nandita Dukkipati. 2023. Bolt: {Sub-RTT} Congestion Control for {Ultra-Low} Latency. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 219–236.
- [9] Inho Cho, Dongsu Han, and Keon Jang. 2016. ExpressPass: End-to-end credit-based congestion control for datacenters. *arXiv preprint arXiv:1610.04688* (2016).
- [10] Prateesh Goyal, Preety Shah, Naveen Kr Sharma, Mohammad Alizadeh, and Thomas E Anderson. 2019. Backpressure flow control. In *Proceedings of the 2019 Workshop on Buffer Sizing*. 1–3.
- [11] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. 2016. RDMA over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 202–215.
- [12] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. 2016. RDMA over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 202–215.
- [13] Torsten Hoefer, Ariel Hendel, and Duncan Roweth. 2022. The convergence of hyperscale data center and high-performance computing networks. *Computer* 55, 7 (2022), 29–37.
- [14] Yimin Jiang, Yibo Zhu, Chang Lan, Bairén Yi, Yong Cui, and Chuanxiong Guo. 2020. A unified architecture for accelerating distributed {DNN} training in heterogeneous {GPU/CPU} clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 463–479.
- [15] John Kim, William J Dally, Steve Scott, and Dennis Abts. 2008. Technology-driven, highly-scalable dragonfly topology. *ACM SIGARCH Computer Architecture News* 36, 3 (2008), 77–88.
- [16] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan MG Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld,

- Michael Ryan, et al. 2020. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 514–528.
- [17] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. 2019. HPCC: High precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*. 44–58.
- [18] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blehm, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zets. 2015. TIMELY: RTT-based congestion control for the datacenter. *ACM SIGCOMM Computer Communication Review* 45, 4 (2015), 537–550.
- [19] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. 2018. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 221–235.
- [20] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. 2015. Inside the social network’s (datacenter) network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. 123–137.
- [21] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. 2015. Jupiter rising: A decade of clos topologies and centralized control in google’s datacenter network. *ACM SIGCOMM computer communication review* 45, 4 (2015), 183–197.
- [22] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P Brighten Godfrey. 2012. Jellyfish: Networking data centers randomly. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. 225–238.
- [23] Brent Stephens, Alan L Cox, Ankit Singla, John Carter, Colin Dixon, and Wesley Felter. 2014. Practical DCB for improved data center networks. In *IEEE INFOCOM 2014-IEEE Conference on Computer Communications*. IEEE, 1824–1832.
- [24] Brent Stephens, Alan L Cox, Ankit Singla, John Carter, Colin Dixon, and Wesley Felter. 2014. Practical DCB for improved data center networks. In *IEEE INFOCOM 2014-IEEE Conference on Computer Communications*. IEEE, 1824–1832.
- [25] Parvin Taheri, Danushka Menikumbura, Erico Vanini, Sonia Fahmy, Patrick Eugster, and Tom Edsall. 2020. RoCC: robust congestion control for RDMA. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*. 17–30.
- [26] Han Wang, Robert Soulé, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. 2017. P4fpga: A rapid prototyping framework for p4. In *Proceedings of the Symposium on SDN Research*. 122–135.
- [27] Xinyu Crystal Wu and TS Eugene Ng. 2022. Detecting and Resolving PFC Deadlocks with ITSY Entirely in the Data Plane. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*. IEEE, 1928–1937.
- [28] Siyu Yan, Xiaoliang Wang, Xiaolong Zheng, Yinben Xia, Derui Liu, and Weishan Deng. 2021. ACC: Automatic ECN tuning for high-speed datacenter networks. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 384–397.
- [29] Minlan Yu, Albert Greenberg, Dave Maltz, Jennifer Rexford, Lihua Yuan, Srikanth Kandula, and Changhoon Kim. 2011. Profiling network performance for multi-tier data center applications. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*.
- [30] Yiwen Zhang, Yue Tan, Brent Stephens, and Mosharaf Chowdhury. 2022. Justitia: Software {Multi-Tenancy} in Hardware {Kernel-Bypass} Networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 1307–1326.
- [31] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion control for large-scale RDMA deployments. *ACM SIGCOMM Computer Communication Review* 45, 4 (2015), 523–536.

A THE OVERALL PROCESS OF CONGESTION CONTROL AT THE SENDER SIDE.

Algorithm 3 outlines the complete procedure of Congestion Control on the sender side for a single flow. Upon receiving each new ACK message, the procedure NewACK is triggered at Line 41. At Line 42, the variable “lastUpdateSeq” is utilized to store the sequence number of the first packet sent with a new W^c . For a new synchronization between W^c and W , the sequence number in the incoming ACK must be larger than “lastUpdateSeq” (as indicated in Line 33–34 and Line 37–38). The sender calculates a new window size, W , at Line 43 or Line 46.

Function MeasureInFlight calculates the normalized in-flight bytes at Line 8. Initially, it calculates the transmission rate ($txRate$) for each link based on the current and previously accumulated transmitted bytes ($txBytes$) and the corresponding timestamp (ts) (Line 7). Additionally, it employs the minimum of the current and previous queue length to eliminate noise in $qlen$ (Line 8). The loop from Lines 10 to 13 selects the maximum value of U_i . Instead of directly utilizing U_i , an Exponentially Weighted Moving Average (EWMA) is applied at Line 13 to filter out noise arising from timer inaccuracies and transient queues.

Function Hop_Detection is dedicated to determining the location of congestion occurrence and assessing the congestion level (Line 17–20). Note that if congestion is detected in the last hop, the sending window of W^c is directly set to the final convergence value by Function UpdateWc (Line 25).

Function ComputeWind, in addition to containing Function UpdateWc to speed up last-hop congestion control, also integrates multiplicative increase/decrease (MI/MD) and additive increase (AI) to strike a balance between responsiveness and fairness. When a sender determines the need to increase the window size, it initially employs AI for a maximum of maxStage attempts with a step size of W_{AI} (Line 36). If there is still room for increase after maxStage AI attempts or if the normalized in-flight bytes exceed η , the window size is promptly adjusted upward or downward (Line 31–32).

Algorithm 3 Overall algorithm at the sender

```

1:  $U[5] \leftarrow 0;$ 
2:  $U_{\max} \leftarrow 0;$ 
3:  $\text{hop} \leftarrow 0;$ 
4: function MEASUREINFIGHT(ack)
5:    $u \leftarrow 0;$ 
6:   for each link  $i$  on the path do
7:      $\text{txRate} \leftarrow \frac{\text{ack.L}[i].\text{txBytes} - L[i].\text{txBytes}}{\text{ack.L}[i].\text{ts} - L[i].\text{ts}};$ 
8:      $u' \leftarrow \frac{\min(\text{ack.L}[i].\text{qlen}, L[i].\text{qlen})}{\text{ack.L}[i].B.T} + \frac{\text{txRate}}{\text{ack.L}[i].B};$ 
9:      $U_i \leftarrow u';$ 
10:    if  $u' > u$  then
11:       $u \leftarrow u'; \tau \leftarrow \text{ack.L}[i].\text{ts} - L[i].\text{ts};$ 
12:       $\tau \leftarrow \min(\tau, T);$ 
13:       $U \leftarrow (1 - \frac{\tau}{T}) \cdot U + \frac{\tau}{T} \cdot u;$ 
14:    return  $U$ 
15: end function
16: function HOP_DETECTION()
17:   for  $j = 0$  to  $nHop$  do
18:     if  $U_j > U_{\max}$  then
19:        $U_{\max} \leftarrow U_j;$ 
20:        $\text{hop} \leftarrow j;$ 
21:   end function
22:   function UPDATERWC(ack)
23:     HOP_DETECTION();
24:     if hop = last hop &  $U_{\max} > \alpha$  then
25:        $W^c \leftarrow \frac{\text{ack.L}[0].B \times \text{RTT} \times \beta}{\text{ack.N}},$ 
26:        $U_{\max} \leftarrow 0;$ 
27:        $\text{hop} \leftarrow 0;$ 
28:   end function
29:   function COMPUTEWIND( $U$ , updateWc, ack)
30:     UPDATERWC(ack);
31:     if  $U \geq \eta$  or incStage  $\geq \text{maxStage}$  then
32:        $W \leftarrow \frac{W^c}{U/\eta} + W_{\text{AI}};$ 
33:       if updateWc then
34:         incStage  $\leftarrow 0; W^c \leftarrow W;$ 
35:       else
36:          $W \leftarrow W^c + W_{\text{AI}};$ 
37:         if updateWc then
38:           incStage  $\leftarrow \text{incStage} + 1; W^c \leftarrow W;$ 
39:       return  $W$ 
40:   end function
41:   procedure NEWACK(ack)
42:     if ack.seq > lastUpdateSeq then
43:        $W \leftarrow \text{ComputeWind}(\text{MeasureInFlight}(ack), \text{True}, ack);$ 
44:       lastUpdateSeq = snd_nxt;
45:     else
46:        $W \leftarrow \text{ComputeWind}(\text{MeasureInFlight}(ack), \text{False}, ack);$ 
47:        $R \leftarrow \frac{W}{T}; L \leftarrow \text{ack.L}$ 
48:   end procedure

```
