Congestion Control for Datacenter Networks: A Control-Theoretic Approach

Danushka Menikkumbura, Parvin Taheri, Erico Vanini, Sonia Fahmy, *Fellow, IEEE*, Patrick Eugster, and Tom Edsall

***Abstract*—In this article, we present *RoCC*, a robust congestion control approach for datacenter networks based on RDMA. *RoCC* leverages switch queue size as an input to a PI controller, which computes the fair data rate of flows in the queue. The PI parameters are self-tuning to guarantee stability, rapid convergence, and fair and near-optimal throughput in a wide range of congestion scenarios. Our simulation and DPDK implementation results show that *RoCC* can achieve up to $7\times$ reduction in PFC frames generated under high load levels, compared to DCQCN. At the same time, *RoCC* can achieve $1.7-4.5\times$ and $1.4-3.9\times$ lower tail latency for long flows and $2.1-7\times$ and $3.5-8.2\times$ lower tail latency for short flows, compared to DCQCN and HPCC, respectively. We also find that *RoCC* does not require PFC. The functional components of *RoCC* can be efficiently implemented in P4 and FPGA-based switch hardware.**

***Index Terms*—Congestion control, datacenter, network programmability, RDMA.**

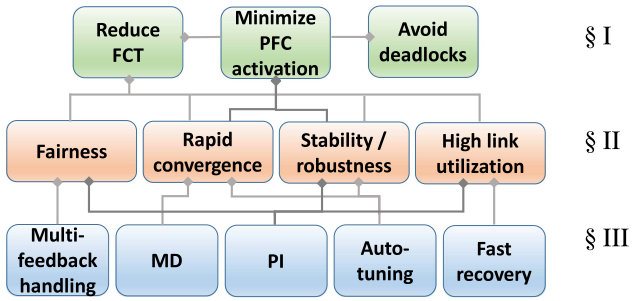


Fig. 1. Relationship of the components (Section III) of *RoCC* to its requirements (Section II) and high-level goals (Section I).

I. Introduction

CONGESTION control in packet-switched networks has a clear goal: reduce FCTs (flow completion times) by providing *low latency for small flows (mice)* and *high throughput for large flows (elephants)*. Typical datacenter networks have topologies with fixed distances (in contrast to the Internet) and fixed bisection bandwidth (in contrast to wireless networks), which may make congestion control there seem simple. It turns out to be quite the opposite, though, as evidenced by the spectrum of solutions that exploit different congestion signals [1], [2], leverage latest developments in network hardware [3], and revisit previous work with a new perspective [4].

a) Goals and Challenges: Datacenter applications have diverse traffic characteristics and require ultra-low latency and high throughput. Most datacenter network traffic has heavy-tailed flow size distribution [5], [6], [7], [8], [9]. At the same time, datacenter network hardware keeps improving in terms of processing power, speed, and capacity, requiring congestion control solutions to be more efficient to fully utilize these hardware enhancements.

TCP becoming a bottleneck in datacenter networks [1] has made operators switch to transports based on RDMA (remote direct memory access); kernel bypass transports such as RoCEv2 (RDMA over converged Ethernet v2) that reduce flow completion time (FCT) by orders of magnitude compared to traditional TCP/IP stacks. remote direct memory access (RDMA) requires losslessness, triggering the need for PFC (priority-based flow control) [10], which prevents packet drop by using back-pressure (at the traffic class level). Alas, priority-based flow control (PFC) has been observed to cause problems such as HoL (head-of-line) blocking, congestion spreading, and routing deadlocks [1], [2], [3], [4], [11]. Less aggressive flow control mechanisms [12] have been proposed to replace PFC. However, we believe that PFC should *only* be triggered to prevent buffer overrun, and we show that if congestion control is able to maintain stable queues on switches, then PFC activation is rare. Datacenter networks have failed to harness the full potential of RDMA due to inefficient congestion control [3], and the many RDMA congestion control solutions developed over the recent past, e.g., [1], [2], [3], [4], are indicators that congestion control for RDMA is a critical problem.

Fig. 1 summarizes high-level goals of congestion control we aim for and the technical requirements we pose to achieve these goals (detailed in Section II), and foreshadows the components of our solution and how these fulfill the requirements.

Manuscript received 16 April 2022; revised 22 December 2022; accepted 4 March 2023. Date of publication 27 March 2023; date of current version 4 April 2023. This work was supported in part by the National Science Foundation (NSF) under Grants CCR-1618923 and CNS-1717493, in part by the Swiss National Science Foundation (SNSF) under Grants 200021_192121 ("FORWARD") and 200021_197353 ("BASIS"), and in part by DFG center under Grant 1053 ("MAKI"). Recommended for acceptance by D. Medhi. *(Corresponding author: Danushka Menikkumbura.)*

Danushka Menikkumbura and Sonia Fahmy are with the Department of Computer Science, Purdue University, West Lafayette, IN 47907 USA (e-mail: dmenikku@purdue.edu; fahmy@cs.purdue.edu).

Parvin Taheri, Erico Vanini, and Tom Edsall are with the Datacenter Networking, Cisco Systems Inc, San Jose, CA 95134 USA (e-mail: pt873@nyu.edu; evanini@cisco.com; edsall@cisco.com).

Patrick Eugster is with the Faculty of Informatics, Universita della Svizzera Italiana (USI), 6904 Lugano, Switzerland (e-mail: eugstp@usi.ch).

Digital Object Identifier 10.1109/TPDS.2023.3259799

TABLE I
COMPARISON OF SELECTED CONGESTION CONTROL SOLUTIONS (*SOLUTION-SPECIFIC, CNP: CONGESTION NOTIFICATION PACKET)

Solution	Switch action	Source action	Destination action
DCTCP [7]	Mark ECN	Adjust congestion window based on ECN	Echo ECN
QCN [13]	Compute and send F_b* to source	Compute rate based on F_b	None
DCQCN [1]	Mark ECN	Compute rate based on CNP	Send CNP to source
TIMELY [2]	None	Send RTT probes and compute rate based on RTT	Echo RTT probes
HPCC [3]	Inject INT	Adjust sending window based on INT	Echo INT
PACC [16]	Send CNP to source	Compute rate based on CNP	None
BFC [17]	Pause traffic on back pressure	Pause traffic on back pressure and trigger back pressure	Trigger back pressure
RoCC	Compute and send rate to source	Use minimum rate received from switch(es)	None

b) State of the art: Congestion control solutions can be broadly categorized as (a) *source-driven* or (b) *switch-driven*, according to the entity (source or switch) playing the key role. With solutions of type (a), the source paces packets (rate or window adjustment) of individual flows, based on a congestion signal it receives from the network (switches and/or destination). With (b), the switch computes the pacing information (usually rate) and sends it to the source, which handles packet pacing.

Table I summarizes the most widely-known datacenter congestion control solutions. A very popular choice in production datacenter networks is DCQCN (datacenter QCN) [1]. DCQCN is a source-driven congestion control approach for RoCEv2, which adapts the CP (congestion point) algorithm of QCN (quantized congestion notification) [13], using the ECN (explicit congestion notification) field in IPv4 headers to notify destinations of congestion. The destination maintains per-flow state in order to relay congestion information back to the relevant source. While DCQCN is effective in reducing the number of PFC frames, its convergence is slow and it can be unstable [14], [15]. TIMELY [2] is another source-driven solution that uses delay (i.e., round-trip time) as the congestion signal, but it falls behind DCQCN in terms of stability and fairness [14]. Several enhancements, e.g., DCQCN+PI [14], DCQCN+ [15], and patched TIMELY [14], have been proposed, but they (too) fail to meet important properties such as stability and fairness, which affect FCT.

The recently proposed HPCC (high precision congestion control) [3] is a source-driven, window-based solution leveraging INT (in-band network telemetry) to gather link load information and adjust source-side transmission window sizes. HPCC outperforms DCQCN, but fails to meet fairness guarantees in scenarios — as we show (Section VI-A) — commonly observed in modern datacenter networks [1]. high precision congestion control (HPCC) also trades link bandwidth for shallow queues and further loses link bandwidth by carrying in-band network telemetry (INT) information.

PACC (proactive and accurate congestion feedback for RDMA congestion control) [16] is a switch-centric congestion control solution that modifies the congestion point (CP) algorithm of DCQCN to generate CNPs (congestion notification packets) and transmit them to the source (instead of performing explicit congestion notification (ECN) marking). proactive and accurate congestion feedback for RDMA congestion control (PACC) employs a PI controller. Based on the experiments in this paper, PACC is more stable and converges faster than DCQCN, but has the same fairness limitations. BFC (backpressure flow control) [17] is a recent solution that uses per-hop per-flow flow control. However, it incurs high implementation and control message overhead, and does not consider fairness and stability goals.

c) Path Forward: We posit that source-driven solutions cannot receive congestion signals (e.g., ECN in DCQCN [1], network delay in TIMELY [2], and INT in HPCC [3]) quickly enough and, as a result, different sources make conflicting decisions about the congestion level they experience in the network. We believe that we need a paradigm shift from host (source-driven) congestion control to core (switch-driven) congestion control in datacenter networks. Concerns with switch-driven solutions that the turnaround time of new features in switch ASICs (application specific integrated circuits) is high are being overturned by the increasing adoption of programmable switch application specific integrated circuits (ASICs) with P4 support [18] by leading switch manufacturers. Similarly, reservations that switch-driven congestion control can hinder line-rate packet processing are countered by recent work [19], [20], [21] showing that event processing (beyond packet arrival and departure events) using P4 does not sacrifice line-rate packet processing.

d) Contributions: We propose a new switch-driven congestion control solution for RDMA-based datacenter networks, *RoCC* (Robust Congestion Control), that: (i) computes a fair rate using a classic proportional integral (PI) controller [22], (ii) signals that fair rate to the sources via ICMP (internet control message protocol), and (iii) auto-tunes the control parameters to ensure stability and responsiveness. This paper extends our previous work [23] with emphasis on the hardware feasibility of the solution and its comparison to some recently-proposed solutions.

Our contributions can be summarized as follows:

1) After establishing important design requirements (Section II), we present the design of *RoCC* (Section III).
2) We analyze *RoCC*, and show how quantized auto-tuning allows trading off stability and rapid convergence under a variety of network conditions (Section V).
3) We evaluate *RoCC* via simulations and a DPDK implementation (Section VI) and compare it to DCQCN, TIMELY, HPCC, PACC, and BFC. Not only does *RoCC* achieve fairness and queue stability, but, compared to

DCQCN, HPCC, PACC, and BFC, it also reduces FCT for real datacenter workloads.

4) We explore the implementation feasibility of *RoCC* (Section IV) using P4 and FPGAs (field-programmable gate arrays), two common technologies used to implement modern datacenter switching devices.

Section VII summarizes related work, and Section VIII concludes the paper.

We will make our implementations used for experiments available to the community.

II. Solution Requirements

We design *RoCC* to satisfy *four* key requirements for effective congestion control in RDMA datacenter networks (Fig. 1).

*Fairness (*FAIR*)*: A set of flows on a congested link must equally share the link bandwidth if they offer equal loads on the link, or otherwise split the link bandwidth based on *max-min* fairness. A flow transmitting at a lower rate than the fair share of the link bandwidth should not be rate-limited. One could argue that short flows can be prioritized (over long flows) to minimize their FCTs, but congestion control is primarily responsible for fair bandwidth allocation across competing flows irrespective of flow size. We believe that prioritizing short flows over long flows should be done at a different level (e.g., packet scheduling, load balancing). Fairness has already been identified as an essential congestion control property by others [3], [16].

Fairness requires handling two special cases. (1) *Multiple bottlenecks:* Intuitively, a flow must effectively use the *minimum* fair rate it can attain through the bottleneck links along its path. That is, the effective rate a flow uses should be based on the *maximum* congestion it experiences along the bottleneck links it passes through and not their number. (2) *Asymmetric network topologies:* Datacenter network topologies can be asymmetric in terms of link bandwidth, switch heterogeneity, or the number of nodes connected to edge switches. Fair rate that flows attain should be agnostic to these asymmetries. The *multi-feedback handler* at the source and the *PI* controller at the switch in *RoCC* handle these cases (see Section III).

*High link utilization (*EFF*)*: Congestion control should not be performed at the expense of link under-utilization resulting in low throughput. A flow must always utilize the maximum possible (fair) rate it can attain — to achieve low FCT — and rapidly reduce the rate when its traffic contributes to potential queue overshoot to prevent PFC. *RoCC* has two key components ensuring optimal link utilization: the *self-riser* at the source rapidly increases the rate in absence of congestion feedback from the switch, and the *PI* controller at the switch guarantees max-min bandwidth allocation for competing flows on a congested link.

*Rapid convergence (*CONV*)*: For low latency and high throughput, it is important to react quickly to increasing and decreasing congestion levels. Rapid convergence helps maintain system stability and, as a result, reduces PFC activation. Switch-driven congestion control has the advantage of being able to disseminate rate updates at the onset of congestion increase (or decrease) at the switch. The *multiplicative decrease (MD)* and *auto-tuner* at the switch are the two key mechanisms of *RoCC* that achieve rapid convergence by aggressively, yet systematically, adjusting the rate.

*Stability/robustness (*STBL*):* Congestion control has to be stable regardless of the number of flows creating congestion. At the same time, the solution needs to be agile when responding to sudden changes in congestion level. Thus, it must *self-tune* to achieve its performance goals across a wide range of congestion scenarios. The *PI* controller and *auto-tuner* at the switch in *RoCC* work together to achieve this.

These four properties together make a flow attain its fair share along its path and reduce FCT. System stability and fast convergence minimize buffer overshoot, reducing PFC activation (PFC increases FCT and creates routing deadlocks).

In addition, it is important that the solution scales well in a datacenter network with an unpredictably large number of flows traversing switches. The amount of state information required to maintain on switches must be limited and the bandwidth demand for feedback messages negligible.

III. *RoCC* Design

We now discuss the different components of *RoCC* and how they achieve our requirements and goals (Fig. 1). At a high level, *RoCC* consists of two major components: (1) fair rate calculator at the switch, and (2) rate limiter at the source. *RoCC* carefully adapts ideas from AFD [24] (PI controller), QCN [13] (multi-bit feedback), PIE [25] (control parameter auto-tuning), and TCP (multiplicative decrease). Fig. 1 shows how each component of *RoCC* contributes to meeting each requirement. Sending a rate from the switch to the host (backward notification) is motivated by the fact that state-of-the-art solutions suffer from the inherent delay of end-to-end congestion signaling (forward notification). We believe that this is a key reason that current switch-assisted congestion control solutions [3], [26], [27] are suboptimal.

A. Definitions

An egress port with its associated queue is defined as the *CP (congestion point)*. The entity that handles traffic rate limiting at the source is defined as the *RP (reaction point)*. Each flow has its own *RL (rate limiter)* at the RP.

Table II defines the symbols used in this section. Δ^Q is the chunk size (resolution) for queue size and related parameters. Similarly, Δ^F is the resolution for rate and related parameters. The purpose of scaling down these parameters is explained in Section III-B. F is the current fair rate at the CP. F is bounded by $F_{\min}$ and $F_{\max}$, the minimum and maximum possible rates at the CP, respectively. Q_{cur} is the size of the queue at the time of calculation of F, and Q_{old} is the corresponding Q_{cur} for the previous value of F. Q_{ref} is a reference queue size, which is a system parameter. α and β are two system parameters whose purpose is explained below.

B. CP Algorithm

The CP periodically calculates the fair rate (FAIR) and sends it to certain sources using a special control message. Fig. 2 shows

TABLE II
SYMBOLS AND DEFINITIONS (*IN MULTIPLES OF Δ^F, †IN MULTIPLES OF Δ^Q, ‡IN MB/S)

Symbol	Definition
Δ^Q	Queue size resolution in Bytes
Δ^F	Rate resolution in Mb/s
Congestion point (CP)	
F	Current fair rate*
$F_{\min}$	Minimum fair rate*
$F_{\max}$	Maximum fair rate*
Q_{cur}	Current queue length†
Q_{old}	Q_{cur} at previous fair rate calculation†
Q_{ref}	Reference queue length†
$Q_{\max}$	Queue length threshold for MD†
Q_{mid}	Queue growth threshold for MD†
α, β	Current system control (PI) parameters
$\tilde{\alpha}$, $\tilde{\beta}$	Static values for α and β respectively
Reaction point (RP)	
cnp	Congestion notification packet (see § III-C)
R_{rcvd}	Received fair rate‡
R_{cur}	Current send rate‡
$R_{\max}$	Maximum send rate‡
CP_{rcvd}	CP that generated R_{rcvd}
CP_{cur}	CP that generated the last accepted R_{rcvd}
GETCP(cnp)	Get the origin (IP) of cnp
GETRATE(cnp)	Get the fair rate in cnp

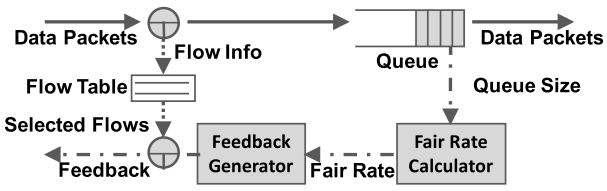


Fig. 2. Overview of *RoCC* design at the CP.

that *RoCC* has three main components at the CP: (1) the fair rate calculator that periodically reads the current queue size to calculate the fair rate and passes it on to (2) the feedback message generator that creates the control message encapsulating the fair rate and sends it to certain sources based on (3) the flow table that keeps track of the flows needing to receive the feedback.

The rate calculation algorithm is shown in Algorithm 1. The queue size and related parameters are scaled down by Δ^Q to reduce the number of bits required for storing Q_{old}. Similarly, fair rate and related parameters are scaled down by Δ^F, to reduce the number of bits required to represent the fair rate in the control message (see Section III-C). Scaling down these parameters is an implementation detail and does not affect the behavior of the algorithm.

The fair rate calculation consists of two main operations:

I. Multiplicative decrease (MD). If the queue length exceeds $Q_{\max}$ or the queue length *change* exceeds Q_{mid} and fair rate is high (i.e., $> \frac{F_{\max}}{8}$), the fair rate is set to $F_{\min}$ or $\frac{F}{2}$, respectively (Line 3 and Line 5). Sudden spikes in queue size can be caused by a new bandwidth-hungry stream or a burst of flows in which case a sharp rate cut is needed to reduce a potential buffer overrun causing PFC activation (CONV). This mode of immediate rate reduction is analogous to the exponential window decrease (i.e., multiplicative decrease) in TCP congestion control. Unlike traditional MD, *RoCC* imposes rapid rate reduction at two different levels (based on queue size and queue growth), further minimizing PFC activation. $Q_{\max}$, Q_{mid}, and Q_{ref} must be chosen such that $Q_{\max} > Q_{\text{mid}} > Q_{\text{ref}}$, to prevent system instability, as discussed below. Our experiments show that $\frac{F_{\max}}{8}$ is sufficiently high to trigger MD. However, this value can be reduced, as the algorithm assures that the rate rapidly converges to the correct value. Therefore, the parameters used in the MD component are not reliability-critical.

II. Proportional integral (PI). The controller that calculates the fair rate in *RoCC* is based on a classic PI controller as used in AFD [24], PIE [25], and QCN [13]. The fair rate is calculated based on *three* quantities derived from queue size: (i) current queue size (Q_{cur}), which signals presence of congestion, (ii) direction of queue change ($Q_{\text{cur}} - Q_{\text{old}}$), which signals congestion increase/decrease, and (iii) deviation of queue size from Q_{ref}, which signals system instability (Line 8). Parameters α and β determine the weight of the last two factors. The fair rate changes until the queue is stable at Q_{ref}. A stable queue indicates that its input rate matches its output rate, and the fair rate through its port has been determined. An important advantage of this controller is that it can find the fair rate without needing to know the output rate of the queue or the number of flows sharing the queue. The algorithm performs a boundary check on the calculated fair rate to make sure it stays within a preconfigured upper bound (Line 10) and lower bound (Line 12). After calculating the fair rate, Q_{old} is set to Q_{cur} (Line 13).

To maintain system stability (STBL) for all values of F while keeping the controller sufficiently agile with sudden queue changes, we design an auto-tuning mechanism for control parameters α and β (Line 15), based on the simple intuition that small adjustments are needed to reach a small target fair rate value (i.e., large number of competing flows) and conversely, larger adjustments are needed to reach a large target fair rate value (i.e., small number of competing flows) (CONV). Thus, the algorithm quantizes the possible fair rate range $[F_{\min}, F_{\max}]$ into six distinct regions, and maps each region to a different pair of values for α and β (as discussed in Section V).

RoCC uses base-2 numbers in multiplication and division operations, which are efficiently implemented using bit shift operations.

C. Feedback Message

The feedback message includes: (1) the fair rate value (in multiples of Δ^F) and (2) information (i.e., the packet headers) required to derive the identifier of the flow to which the rate applies. Using this information, the reaction point (RP) can correctly match the feedback message to the relevant rate limiter (RL). We use internet control message protocol (ICMP) for the CNP (congestion notification packet) and prioritize congestion notification packets (CNPs) to minimize queuing delay. This prioritization of feedback messages further reduces reaction delay of *RoCC* (CONV) compared to state-of-the-art solutions that employ end-to-end congestion notification (e.g., ECN in DCQCN [1], delay in TIMELY [2], and INT in HPCC [3]). ICMP

Algorithm 1: Fair Rate Computation at the CP.

1: **function** CALCULATE_FAIR_RATEQ_{cur}
2: **if** $Q_{\text{cur}} \geq Q_{\text{max}}$ AND $F > \frac{F_{\text{max}}}{8}$ **then**
3: $F \leftarrow F_{\text{min}}$
4: **else if** $(Q_{\text{cur}} - Qold) \geq Q_{\text{mid}}$ AND $F > \frac{F_{\text{max}}}{8}$ **then**
5: $F \leftarrow F \div 2$
6: **else**
7: $\alpha, \beta \leftarrow$ AUTO_TUNE()
8: $F \leftarrow F - \alpha \times (Q_{\text{cur}} - Q_{\text{ref}}) - \beta \times (Q_{\text{cur}} - Q_{\text{old}})$
9: **if** $F > F_{\text{max}}$ **then**
10: $F \leftarrow F_{\text{max}}$
11: **if** $F < F_{\text{min}}$ **then**
12: $F \leftarrow F_{\text{min}}$
13: $Q_{\text{old}} \leftarrow Q_{\text{cur}}$
14: **return** F
15: **function** AUTO_TUNE
16: $level \leftarrow 2$
17: **while** $F < \frac{F_{\text{max}}}{level}$ AND $level < 64$ **do**
18: $level \leftarrow level \times 2$
19: $ratio \leftarrow level \div 2$
20: $\alpha \leftarrow \tilde{\alpha} \div ratio$; $\beta \leftarrow \tilde{\beta} \div ratio$
21: **return** α, β

can be efficiently and conveniently prioritized over regular traffic on routers using its protocol number (0x01).

D. Flow Table

A flow table keeps track of the recipients of the feedback messages. *RoCC* has the flexibility of using different flow table implementations, such as:

1) Maintaining a table of the flows currently in the queue: This is our default flow table implementation and the table size is bounded by the queue size.
2) *RoCC* has a lower bound for fair rate, hence the number of concurrent flows on a link has an upper bound (i.e., F_{max} / F_{min}). This bounds the size of the table, and can be used in conjunction with a simple age-based flow eviction mechanism.
3) AFD-FT: This is the flow table implementation used in AFD [24], the first AQM mechanism that leveraged flow size and flow rate distributions to scale per-flow state.
4) ElephantTrap [28]: This identifies large (elephant) flows that cause persistent congestion by sampling packets. The probability of a flow being identified as an elephant depends on the sampling rate. A flow in the table is evicted based on a frequency counter (i.e., LFU).
5) BubbleCache [29]: This employs packet sampling to efficiently capture elephant flows at high speeds.

These different flow table implementations facilitate sending feedback messages to selected flows (e.g., elephants only) at the cost of lower stability margins. .

Since the PI controller changes the fair rate until the arrival rate matches the drain rate of the congested queue, the fair rate will stabilize at

$$F = \frac{C_l - BW_{\text{innocent}}}{N} \tag{1}$$

where C_l is the bandwidth of the congested link, BW_{innocent} is the total bandwidth used by the flows that do not contribute to congestion (innocent flows), and N is the number of flows contributing to congestion. Thus it suffices to track the flows that most contribute to congestion, hence queue buildup. Usually, long flows (e.g., disk replication traffic) or long-lasting bursts of short flows (e.g., gRPC calls to a popular service) contribute to congestion.

E. RP Algorithm

The RP employs an algorithm, triggered by each incoming CNP, to update the sending rate of the corresponding RL. The RP also uses a fast recovery mechanism to rapidly increase the sending rate of the RL, in the absence of CNPs, which implies absence of congestion (EFF).

Algorithm 2 shows the RP algorithm, which has two routines:

a) Process CNP: RoCC uses a simple yet effective approach for handling CNPs from CPs along a flow's path. The RP accepts a CNP if (i) it (CP_{rcvd}) was generated by the same CP that generated the last accepted CNP (CP_{cur}) for the RL or (ii) its fair rate (R_{rcvd}) is smaller than the current sending rate R_{cur} used by the RL (Line 4). This ensures that the RL always uses the fair bandwidth share that the flow can attain at the most congested CP on its path (FAIR). Upon accepting a new fair rate, the RL immediately updates its current sending rate to the new rate (Line 5). The algorithm also remembers CP_{rcvd} as most congested CP on the flow's path (Line 6).

b) Fast Recovery: An RL can stop receiving CNPs when the flow no longer contributes to any CP on its path. Since the RP may not receive all CNPs destined to it, the RL should automatically increase its rate R_{cur} after a certain period of not receiving CNPs (EFF). *RoCC* exponentially increases its rate based on a timer in this situation (Line 8). *RoCC* stops fast recovery upon accepting a CNP (Line 7). The sending rate is bounded by the maximum allowed rate R_{max}, usually the link bandwidth. If the rate reaches R_{max}, the RL is uninstalled, allowing the flow to transmit as without congestion. This fast recovery mechanism is simpler than that of DCQCN.

F. Rate Computation at the Host

RoCC does not require that the CP carry out the rate computation. Instead, the CP can send the values of Q_{cur}, Q_{ref}, Q_{mid}, Q_{max}, F, F_{min}, F_{max}, $\tilde{\alpha}$, and $\tilde{\beta}$, to the host and have it compute the rate. There are two simple approaches for sending these to the host, requiring modest modifications to the CNP: (1) CP provides all the values, (2) CP only provides Q_{cur} and F, and the host looks up the remainder of the values, which are specific to a given F, in a simple registry. This flexibility simplifies the *RoCC* implementation, especially on legacy switch ASICs that have limited arithmetic support (e.g., no floating-point operation support).

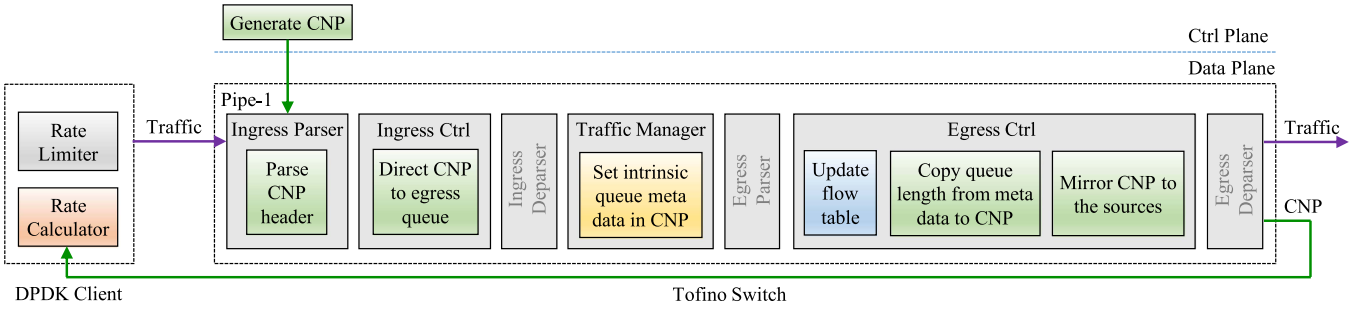


Fig. 3. *RoCC* switch implementation in P4.

Algorithm 2: Rate Limiting at the RP.

procedure PROCESS_CNPcnp
  $R_{\text{rcvd}} \leftarrow$ GETRATE(CNP) $\times \Delta^F$
  $CP_{\text{rcvd}} \leftarrow$ GETCP(CNP)
  if $R_{\text{rcvd}} \leq R_{\text{cur}}$ OR $CP_{\text{rcvd}} = CP_{\text{cur}}$ **then**
    $R_{\text{cur}} \leftarrow R_{\text{rcvd}}$
    $CP_{\text{cur}} \leftarrow CP_{\text{rcvd}}$
    RESET_TIMER()
procedure TIMER_EXPIRED
  if $R_{\text{cur}} > R_{\text{max}}$ **then**
    remove this rate limiter if its queue is empty
  else
    $R_{\text{cur}} \leftarrow R_{\text{cur}} \times 2$
    RESET_TIMER()

IV. IMPLEMENTATION

The feasibility of *RoCC* highly depends on its implementability on hardware, especially on the switch, which is resource-constrained and has strict data-path latency demands. Therefore, we investigate the feasibility of implementing the CP algorithm at the switch. We briefly discuss the resource demands for implementing it on a custom ASIC. We study P4, which is an emerging technology for implementing datacenter switches. Additionally, we study field-programmable gate array (FPGA) to assess the resource and latency demands of *RoCC* in switch hardware. These two factors – the lower the better – are key to the efficient implementation of *RoCC* (or any other application) on the switch without hindering its line-rate operation. It is important to note that switches are implemented using ASICs, and an FPGA-based implementation provides good approximations to latency and resource demands of the implementation on ASIC.

A. Basics

a) CP: The key components of the CP implementation include: (1) flow table, (2) periodic calculation of fair rate for egress queues (timer event handling), (3) associating computed fair rates with corresponding flows (flow table lookup), and (4) generating and transmitting CNPs to the flow sources.

b) RP: Host networking stacks support intercepting ICMP (CNP) messages as well as implementing the RP algorithm (e.g., DPDK, SmartNIC, and Linux raw sockets).

RoCC can be efficiently implemented on ASICs. A custom ASIC implementation of *RoCC* is estimated to require approximately 1.1 M gates, 1.9 Mb dual port SRAM, 1.2 Mb of SRAM, and 0.138 Mb of TCAM (totalling 3.2 Mbits of memory). This constitutes a negligible 0.7% of chip die area.

B. P4 Implementation

As data plane programmability becomes widespread, P4 [18] is becoming the de facto framework for programming switches, and major switch vendors already support P4.

Fig. 3 illustrates our *RoCC* implementation in P4. Our switch component (CP) is implemented in P4 using Intel Tofino that is based on TNA (tofino native architecture). Below, we walk through our switch implementation according to its execution path.

1) CNP generator: is implemented in the control plane. Its task is to send CNP packets onto the data plane every T seconds. We use the special network interface that Tofino exposes, to input the control packet to the data plane. Listing 1 defines the P4 header for CNP. The controller sends a CNP for each port specifying the port identifier as meta data.
2) Parser: extracts `rocc_h` header from an incoming CNP. We assume the data plane only receives CNPs through the control interface.
3) Ingress pipeline: directs CNPs to their respective egress queues by setting `ucast_egress_port` of `ingress_intrinsic_metadata_for_tm_t` to `port` of `rocc`. Only CNP has a valid `rocc_h` header.
4) Traffic manager: attaches certain intrinsic meta information including egress queue length (`deq_qdepth` of `egress_intrinsic_metadata_t`) to every passing packet. As a result, a CNP has its egress queue length when it reaches the egress pipeline.
5) Egress pipeline: handles two tasks: (i) maintain a flow table. Our flow table is implemented using a simple `Register` array in P4. It is similar to the flow table used in Turboflow [21], which is proven not to hinder line-rate traffic processing in the data plane. The flow table is updated for each data packet going through the egress pipeline; and (ii) set queue length and other parameters required for rate computation (see Section III-F) on CNP and "mirror" it to selected sources based on the flow table.

The open P4 v1model and proprietary Tofino TNA conceptually follow the same architecture, and modifying a program written for one architecture to work on the other is relatively straightforward. Unlike the v1model, Tofino TNA has an ingress deparser and an egress parser in addition to the ingress parser and egress deparser enabling more efficient packet mirroring and cloning, compared to the v1model. Additionally, Tofino TNA has an in-built packet generator that can be used to generate control packets in the data plane itself, and we plan to use this feature in the future to further enhance our P4 implementation eliminating the need for generating CNPs from the control plane.

Our client (RP) is implemennted using DPDK (data Plane development kit), and performs *two* main tasks: (I) intercept CNPs to compute fair rate, and (II) rate limit flows based on the computed rate. Pktgen [30] is an efficient traffic generation tool based on data Plane development kit (DPDK). We modify Pktgen to intercept CNP and change its data rate using the computed fair rate.

```
header rocc_h {
    bit<16> port;
    bit<16> qdepth;
    bit<8> counter;
}

struct headers {
    /* standard headers */
    ethernet_h  ethernet;
    ipv4_h      ipv4;
    icmp_h      icmp;
    /* RoCC header */
    rocc_h      rocc;
}
```

Listing 1. P4 Header Definition for CNP.

C. Implementation

We use Xilinx Vitis HLS 2020.2 [31] targeting Xilinx Virtex-7 XC7V2000 T FPGA device to design and perform a high-level synthesis of: (1) the CP algorithm, and (2) a flow table that supports flow identifier update and lookup, to understand the resource and latency demands of the two components. FPGA has two key resources: flip-flop (FF) and look-up table (LUT), and we measure the demand on these two resources in our implementation.

a) CP algorithm: Based on the Vitis synthesis report, the CP algorithm supports a maximum clock frequency of 370 MHz, and only requires 1% FF and 2% LUT demonstrating its feasibility on FPGA.

b) Flow table: In our implementation, the 5-tuple (flow identifier) of each frame traversing the egress port is streamed into the table module, which performs table updates in a *lazy* fashion. The flow table uses a circular array to store the flow identifiers, and round robin for retrieving them. Using other mechanisms for retrieving flow identifiers is outside the scope of this work. We synthesize a flow table of 1024 entries. Based on the Vitis synthesis report, the flow table supports a maximum clock frequency of 200.56 MHz. Both adding and retrieving a flow identifier take 5 ns with initiation interval 1. The flow table only requires 4% FF and 4% LUT on FPGA.

We expect our implementations to perform better on ASIC than on FPGA.

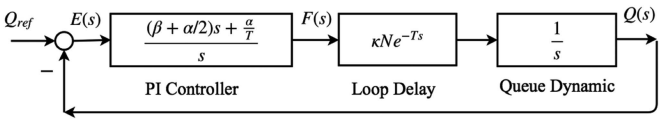


Fig. 4. The feedback loop of *RoCC* controller that includes PI, queue dynamic, and loop delay.

V. Stability Analysis

We analyze the stability of *RoCC* based on the control system it employs in its CP algorithm (i.e., PI). We first derive a mathematical model for the CP algorithm, and based on this, we use phase margin analysis to show that the automatic parameter tuning mechanism in *RoCC* ascertains system stability.

A. System Model

Assume that N flows are congesting a network link l with capacity C_l. Each flow is shaped using the same feedback rate sent by the CP. Therefore, the queue dynamic is

$$\frac{dQ(t)}{dt} = \frac{\Delta^F \times N \times F(t-T) - C_l}{\Delta^Q}, \tag{2}$$

where T is the update interval and $F(t)$ is the fair rate received from the CP at time t. For this analysis, we can safely ignore the MD rate reduction as long as we carefully choose thresholds not to interfere with the PI controller.

The PI controller calculates the fair rate based on the current queue length and the trend of change in queue length. Using bilinear transformation as in [25], we can convert the operation at Line 8 from Algorithm 1 into the continuous form

$$\frac{dF(t)}{dt} = -\frac{\alpha}{T}[Q(t) - Q_{\text{ref}}] - \left(\beta + \frac{\alpha}{2}\right)\frac{dQ(t)}{dt}. \tag{3}$$

After a Laplace transformation on (2) and (3) we get

$$Q(s) = \kappa N \frac{e^{-sT}}{s} F(s), \tag{4}$$

$$F(s) = \frac{(\beta + \alpha/2)s + \frac{\alpha}{T}}{s} E(s), \tag{5}$$

with $\kappa = \frac{\Delta^F}{\Delta^Q}$, and $E(s) = Q_{\text{ref}} - Q(s)$ the error signal. The open-loop transfer function of the whole system (see Fig. 4) is

$$G(s) = K \frac{(1 + \frac{s}{z_1})}{s^2} e^{-sT}, \tag{6}$$

where $z_1 = \alpha/((\beta + \alpha/2)T)$ and $K = \kappa N \alpha / T$.

Note that the gain of the open loop function, K, is in proportion to N, the number of flows sharing the link. This affects the stability of the closed-loop system.

B. Determining Control Parameters

The update interval T and reference queue depth Q_{ref} must be set carefully, as they impact system stability (STBL).

A typical guideline is to set T to 1–2 times the RTT (round trip time). The rationale behind making T larger than the RTT is to allow any change in the fair rate calculation to go into effect before the next update of fair rate.

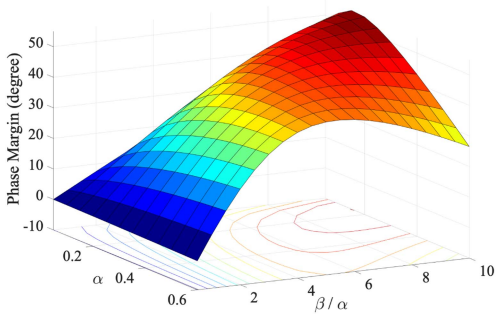


Fig. 5. Phase margin as a function of parameters α and β. Phase margin above 0 ensures a stable system.

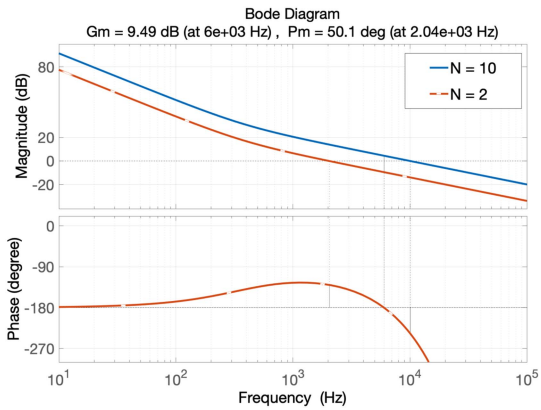


Fig. 6. Stability margin for two values of N. For $N > 2$, the system gain increases. Thus 0 dB gain occurs at higher frequency resulting in lower phase margin.

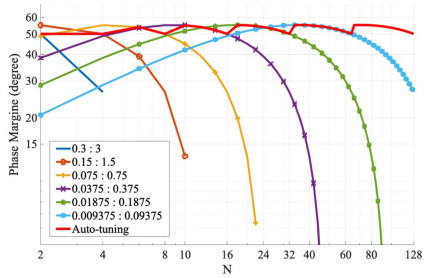
We chose Q_{ref} based on bandwidth-delay product, $T \times C_l$ where C_l is the egress link bandwidth. Our experiments show that Q_{ref} should be half of the bandwidth-delay product for lower latency.

For specific values of T and Q_{ref}, we use Bode diagram analysis to find values for α and β for the system to have sufficient phase margin. Fig. 5 shows phase margin for different values of α and β, with $T = 40\ \mu s$, $N = 2$. We need to set values of α and β producing a phase margin above 0, which guarantees system stability.

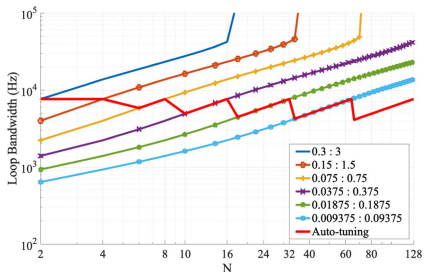
We now show how the number of flows N impacts the stability of the system. Since open-loop gain is in proportion to N, the larger the number of flows, the less stable our system would be for fixed α and β. Fig. 6 illustrates how setting $N = 10$ reduces the phase margin from 50 to -50, making the system completely unstable.

We can solve this problem by selecting conservative values for α and β to guarantee stability for $N \in [2, 128]$. E.g., setting $\alpha = 0.0093$ and $\beta = 0.0937$ ensures a phase margin above 20 degrees and stability for all values of N. However, when the number of flows sharing the link is small, it takes a long time for the PI controller to reach the stable fair rate.

Fig. 7(a) shows the phase margin as a function of N for various $\alpha : \beta$ value pairs. We start with $0.3 : 3$ and keep dividing each value by 2 to get the next pair. Fig. 7(b) plots loop bandwidth



(a) Phase margin as a function of N for 6 pairs of $\alpha : \beta$ values. Lower values of α and β provide a positive phase margin for all values of N.



(b) Loop frequency as a function of N for 6 pairs of $\alpha : \beta$ values. Lower values of α and β cause slower response for smaller values of N.

Fig. 7. Impact of the number of flows N with different values of α and β.

for each pair of values. A higher loop bandwidth yields a faster response time. As shown in Fig. 7(a), reducing α and β makes the system stable for a larger range of values of N, at the expense of slower convergence as Fig. 7(b) shows. Choosing higher values of α and β provides faster response time, but the system becomes unstable as N increases.

C. Auto-Tuning α and β

a) Rationale: With auto-tuning, *RoCC* can maintain stability and reduce response time for all values of N by adapting α and β. N can be inferred from the current fair rate, since we do not know the number of flows sharing the link at any given time. The fair rate and N are inversely proportional, since the bandwidth attained by flows contributing to congestion on a link follows (1). For large values of fair rate (small N), α and β should be large to ensure fast convergence without losing stability. In contrast, for small values of fair rate (large N), α and β should be small.

b) Discrete α and β values: Adjustments to α and β can be made in a continuous fashion. As Fig. 7(a) and (b) show, quantizing the fair rate range into 6 levels and choosing a different pair of values for α and β for each level is sufficient to maintain the same phase margin and response time for all values of N. PIE [25] follows the same approach and uses discrete control parameter value pairs for simplicity. The discrete adjustments can be easily implemented using a binary search-based lookup mechanism. Our experiments (Section VI) show that auto-tuning increases stability and reduces convergence time for a variety of workloads.

VI. Evaluation

We conduct three types of experiments to evaluate *RoCC*:

1) Micro-benchmarks and comparisons to the state of the art with respect to the four requirements in Section II using simulations.
2) Evaluation with DPDK and P4 to confirm the properties of *RoCC* on real systems and validate our simulations.
3) Larger scale evaluation using a simulation setup resembling a real datacenter network in terms of topology, number of nodes, and traffic patterns, to examine whether *RoCC* meets system and user goals (Section II).

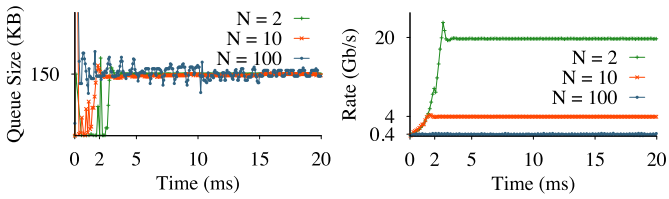
For simulations, we use OMNeT++ [32] to implement a prototype of *RoCC*. In our model, the fair rate has fixed point precision to mimic hardware implementation. Our datacenter model has been previously used in the literature [33], with simulation results corresponding to results obtained by running similar tests on real testbeds. We implement several state-of-the-art solutions, which do not have publicly available OMNeT++ implementations. We use their public code repositories for reference, and configure the solutions based on details given in respective papers. We use traffic workloads derived from publicly available datacenter traffic traces [7], [8], [9].

a) System parameters: All interconnections in our simulations are either 40 Gb/s or 100 Gb/s links. We chose PFC threshold values 500 KB and 800 KB for 40 Gb/s and 100 Gb/s links, respectively, based on [34]. NIC (RP) reaction delay for feedback messages is 15 μs. Δ^F is 10 Mb/s and Δ^Q is 600 B. T is set to 40 μs. $F_{\min}$ is 10, irrespective of link bandwidth. $F_{\max}$ is 4 K and 10 K for 40 Gb/s and 100 Gb/s links, respectively. Q_{ref}, Q_{mid}, and $Q_{\max}$ are 150 KB, 300 KB, and 360 KB, respectively, for 40 Gb/s links, and 300 KB, 600 KB, and 660 KB, respectively, for 100 Gb/s links. $\tilde{\alpha}$ and $\tilde{\beta}$ are 0.3 and 1.5, respectively, for 40 Gb/s links, whereas the values are 0.45 and 2.25, respectively, for 100 Gb/s links. We use the default flow table implementation (cf. Section III-D (1)).

A. Micro-Benchmarks

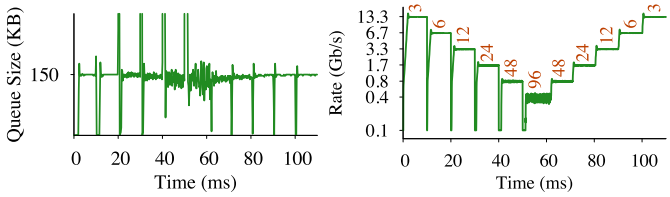
We use a topology with N source nodes connected to a single destination node through a switch. This setup has a single bottleneck link (from the switch to the destination node) of bandwidth B. Each source node has an RDMA application generating traffic based on the workloads mentioned earlier. The destination node has an application receiving traffic from all source nodes. *RoCC* is enabled on the egress switch interface towards the destination. Unless otherwise mentioned, we use this setup for all scenarios in this section.

a) Fairness (FAIR) and stability (STBL): The offered load at each source node is 90% of the link bandwidth, causing persistent congestion on the bottleneck link. We observe system stability and fair bandwidth allocation for $N = 2$, 10, and 100, and for two different values of B (40 Gb/s, 100 Gb/s). As Fig. 8 shows, the computed fair rate converges in ~ 2 ms for all values of N. Note that the fair rate converges faster with larger N, and the egress queue at the switch is stable at its reference value regardless of N. The stability of *RoCC* is governed by the PI controller, which uses queue size as input, hence queue stability indicates system stability. This observation is consistent with the auto-tuning discussion in Section V-B. Results for $B = 100$ Gb/s are consistent with those for $B = 40$ Gb/s.



(a) Queue stability. (b) Fair rate.

Fig. 8. Fairness and stability of *RoCC* as load increases.



(a) Queue stability. (b) Fair rate.

Fig. 9. Convergence of *RoCC*. The numbers in red are the flow counts during the intervals.

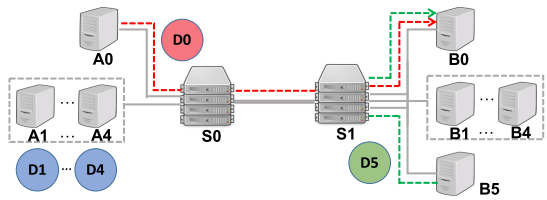
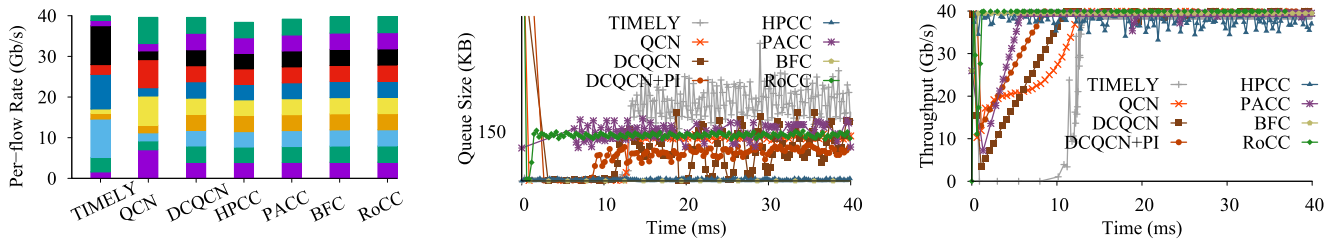


Fig. 10. Multi-bottleneck topology.

b) Convergence (CONV) and high link utilization (EFF): We exponentially increase (and reduce) the link load level by dynamically starting (and stopping) flows to investigate the system behavior when load fluctuates. We start with 3 flows (i.e., $N = 3$) and start new flows every 10 ms such that N doubles every time, until $N = 100$. After 10 ms, we begin to stop flows every 10 ms such that N halves every time until $N = 3$. We record the fair rate and egress queue size on the switch. As Fig. 9 shows, the fair rate decreases from 13.3 Gb/s to 400 Mb/s as N increases from 3 to 100. Similarly, the fair rate increases from 400 Mb/s back to 13.3 Gb/s as N decreases from 100 to 3 (EFF). As the load fluctuates, the queue size and fair rate always stabilize in less than 2 ms (CONV). When new flows start and create a traffic burst, the queue size suddenly increases causing the MD of *RoCC* to kick in and bring the rate down, draining the queue. Similarly, when flows stop, the traffic reduces, causing the queue to drain. The PI of *RoCC* ensures that the queue grows and rapidly stabilizes (CONV).

c) Comparison to existing solutions: We compare *RoCC* to the state of the art and state of the practice in datacenter networks, in terms of the expected congestion control requirements. We include QCN [13], DCQCN [1], DCQCN+PI [14], TIMELY [2], HPCC [3], PACC [16], and BFC [17] in our



(a) Fairness. (b) Queue stability. (c) Convergence.

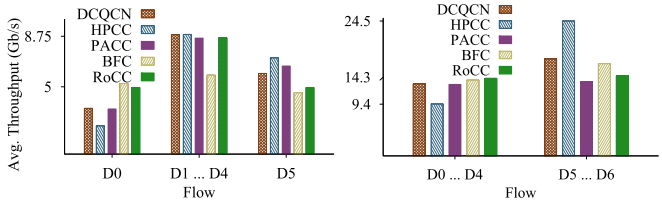
Fig. 11. Comparing *RoCC* with TIMELY, quantized congestion notification (QCN), DCQCN, HPCC, PACC, and BFC in terms of the requirements in Section II.

comparison. We configure the simulation setup with $N = 10$ and $B = 40$ Gb/s. We record the fair rate, egress queue size, and egress link utilization on the switch.

As Fig. 11(a) shows, the flows experience a significant deviation from the expected average fair rate of 4 Gb/s with TIMELY and QCN. In contrast, each flow attains the expected average fair rate with *RoCC* (FAIR). In this scenario, DCQCN, HPCC, PACC, and BFC are comparable to *RoCC* in terms of fairness, but the average per-flow rate attained is lower than the expected value with HPCC. This is a result of the link bandwidth headroom that HPCC reserves. When we rerun the same simulation with the link bandwidth headroom reduced from 5% (the recommended value [3]) to 1%, we observe that HPCC critically loses its stability and fairness. Therefore, HPCC relies on its recommended headroom for proper functionality and for maintaining a shallow queue at the switch ([3] sec. 3.3) to minimize PFC activation, which is one of its key design goals. In the next section, we further examine the fairness of DCQCN, HPCC, PACC, BFC, and *RoCC*.

Fig. 11(b) and (c) show the queue size and the bottleneck link utilization (EFF). *RoCC* maintains the queue size at the reference queue size (STBL). DCQCN and TIMELY fail to maintain a stable queue, fluctuating around ~ 100 KB and ~ 200 KB, respectively. There are *two* key observations here: (i) DCQCN's stability improves significantly when its ECN marking mechanism is modified to use a PI controller (DCQCN+PI [14]), and (ii) PACC, which modifies the CP algorithm of DCQCN to generate CNPs using a PI controller *and* transmit them directly to the source, is more stable and converges faster than DCQCN. These observations further justify the use of a PI controller and backward notifications in *RoCC*. DCQCN+PI and TIMELY achieve high link utilization – DCQCN+PI with a fairly stable queue, and TIMELY with an unstable yet non-empty queue. HPCC by design underutilizes links to reserve bandwidth headroom, hence our results in this case are consistent with its expected behavior. The stability of the rate attained by each flow closely follows that of link utilization.

d) Multiple bottlenecks: A flow in a datacenter network may encounter multiple CPs on its path. Intuitively, the flow in this case must attain the fair bandwidth corresponding to the most congested CP on its path (FAIR). We examine the effectiveness of *RoCC* and the state of the art in handling multiple CPs. We use the topology in Fig. 10, which has 6 source nodes (A0...A4, B5) and 5 destination nodes (B0...B4) connected to

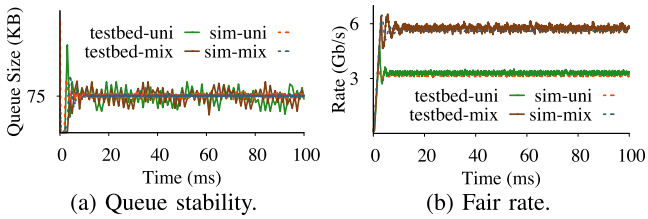


(a) Multi-bottleneck. (b) Asymmetric topology.

Fig. 12. Fairness of DCQCN, HPCC, PACC, BFC, and *RoCC*.

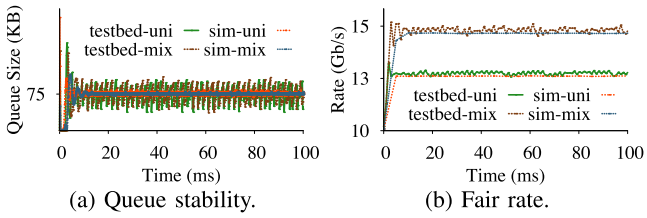
switches S0 and S1. Each access link is 10 Gb/s and the link between the switches is 40 Gb/s. Ai transmits data flow Di to Bi ($i = 1, 2, 3, 4$). A0 and B5 transmit data flows D0 and D5, respectively, to B0. D0 traverses two CPs, S0 and S1. We compare *RoCC* to DCQCN [1], HPCC [3], PACC [16], and BFC [17] in terms of flow-level bandwidth allocation. As Fig. 12(a) shows, flows {D0, D5} and {D1,...,D4} attain their fair bandwidth shares of 5 Gb/s and 8.75 Gb/s, respectively, with *RoCC*. In contrast, D0 attains 30% less throughput than expected with DCQCN and, as a result, the remaining flows utilize more bandwidth than they should. HPCC exhibits a similar behavior where flow D0 has around 50% less throughput than expected. The fairness of PACC is very similar to that of DCQCN in this scenario. Distinctively, {D1,...,D4} attain about 45% less bandwidth than expected with BFC while D0 and D5 utilize almost the expected bandwidth with it. We conclude that *RoCC* is best at handling feedback messages received from multiple CPs (FAIR).

e) Asymmetric topology: As datacenter equipment is upgraded over time, their topologies become asymmetrical. We use a network topology with asymmetric links to compare *RoCC* to DCQCN [1], HPCC [3], PACC [16], and BFC [17] in terms of flow-level bandwidth allocation. 2 switches (S0 and S1) are connected to a third switch (S2) using 100 Gb/s links. A destination node (B0) is connected to S2 using a 100 Gb/s link. 5 source nodes (A0...A4) are connected to S0 using a 40 Gb/s links, and 2 source nodes (A5 and A6) are connected to S1 using 100 Gb/s links. Nodes A0...A6 each transmit a data flow (D0...D6, respectively), destined to B0. A0...A4 and A5...A6 should get the same total bandwidth (40 Gb/s $\times$ 5 and 100 Gb/s $\times$ 2, respectively) through S0 and S1, respectively. We run the experiment at 90% load to record average throughput attained by D0...D4 and D5...D6.



(a) Queue stability. (b) Fair rate.

Fig. 13. Testbed results versus simulation results.



(a) Queue stability. (b) Fair rate.

Fig. 14. P4 results versus simulation results.

We make an interesting observation. As the bottleneck link in this topology (S2 $\rightarrow$ B0) is shared among the 7 flows, each flow should obtain a fair bandwidth share of 14.29 Gb/s. Fig. 12(b) shows that, with *RoCC*, each flow attains the intended bandwidth. In contrast, HPCC allocates more bandwidth to the flows originating from nodes connected using higher bandwidth links and, as a result, D5 and D6 equally share most of the bandwidth on the bottleneck link and attain ~ 24.5 Gb/s bandwidth each. The remaining 5 flows equally share the remaining bandwidth on the bottleneck link causing each to only obtain ~ 9.40 Gb/s bandwidth. DCQCN is better than HPCC in terms of fairness in this scenario where each flow attains bandwidth close to the fair share value. Both PACC and BFC are better than DCQCN with respect to fairness in this case, and PACC closely follows *RoCC*.

f) Key takeaways: From these experiments, we make the following key observations: (i) *RoCC* is fair, efficient, stable, and converges rapidly. It is effective in handling feedback from multiple CPs and works well with asymmetric network topologies; (ii) *RoCC* can outperform the state of the art in terms of fairness, stability, and convergence.

B. Evaluation With DPDK Implementation

We implement *RoCC* using the popular DPDK [35] kernel bypass stack to validate our simulation results. The switch implementation uses three logical cores for handling data reception, packet switching, and data transmission and congestion control respectively. The source implementation uses two logical cores, one for data reception and the other for data transmission and rate limiting. We use the reserved ICMP type 253 for feedback messages.

We deploy our DPDK implementation on a network setup on CloudLab [36] that has 3 source nodes connected to a destination node through a switch using 10 Gb/s links. Each node in this topology is a Dell Poweredge R430 machine with two 2.4 GHz 64-bit 8-Core Xeon E5-2630v3 processors, 8 GT/s, 20 MB cache, 64 GB 2133 MT/s DDR4 RAM, and 2 Intel X710 10 Gb/s NICs. With this configuration, our switch is capable of working as a 10 GbE 4-port switch. We use an iPerf [37] UDP client at each source node and three iPerf UDP servers at the destination, each receiving traffic from one client. We set Q_{ref}, Q_{mid}, and Q_{max} to 75 KB, 150 KB, and 210 KB, respectively. We set T to 100 μs to match the propagation delays in this environment. We run two different test scenarios and record fair rate and switch egress queue size. We record the same observations for the corresponding simulations for comparing with our testbed results.

In the first scenario, we configure each client to generate traffic load equal to the link bandwidth (10 Gb/s). Fig. 14(a) shows that the queue size stabilizes at 75 KB for both the testbed (testbed-uni) and simulation (sim-uni). In Fig. 14(b), the fair rate for the testbed and simulation both stabilize at 3 Gb/s.

In the second scenario, we configure the sending rate of the 3 clients to 10 Gb/s, 3 Gb/s, and 1 Gb/s, respectively. Fig. 14(a) shows that Q_{cur} stabilizes at 75 KB for both the testbed (testbed-mix) and simulation (sim-mix) results. Fig. 14(b) shows that the flows attain the max-min fair value of 6 Gb/s in both testbed and simulation experiments.

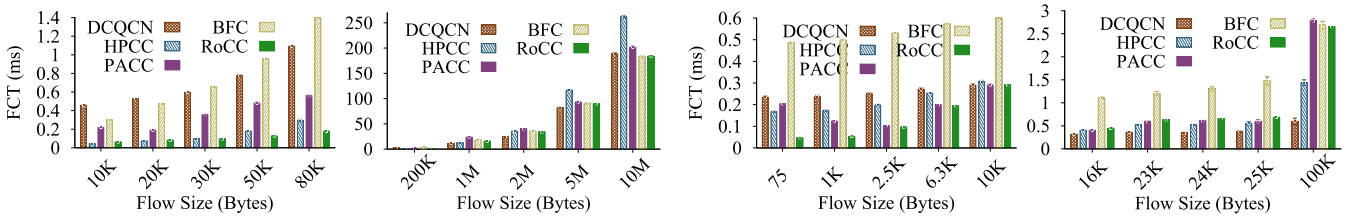
C. Evaluation With P4 Implementation

We deploy our P4 switch implementation on a Wedge 100BF-32X switch that has a programmable Intel Tofino ASIC. The switch is connected to a node that has a D1517 1.60 GHz 8-core CPU and 8 G RAM, using Chelsio T62100-SO-CR unified wire ethernet controllers. We run our traffic generators and receivers on this node. We configure the links to run at 40 Gb/s. With this configuration, our switch is capable of working as a 40 Gb/s 4-port switch. We setup Pktgen on the host machine to send UDP traffic through 3 ports, and receive traffic through the fourth. Q_{ref}, Q_{mid}, and Q_{max} are 75 KB, 150 KB, and 210 KB, respectively. T is 20 μs to match the propagation delays in this environment. Δ^Q in Tofino is 80 bytes. We run two different test scenarios and record fair rate and switch egress queue size. We record the same observations for the corresponding simulations for comparing with our P4 testbed results (same as in Section VI-B).

In the first scenario, we configure each sender to transmit data at link speed (40 Gb/s). Fig. 14(a) shows that the queue size stabilizes at 75 KB for both the testbed (testbed-uni) and simulation (sim-uni). In Fig. 14(b), the fair rate for the testbed and simulation both stabilize at $\sim$ 13 Gb/s.

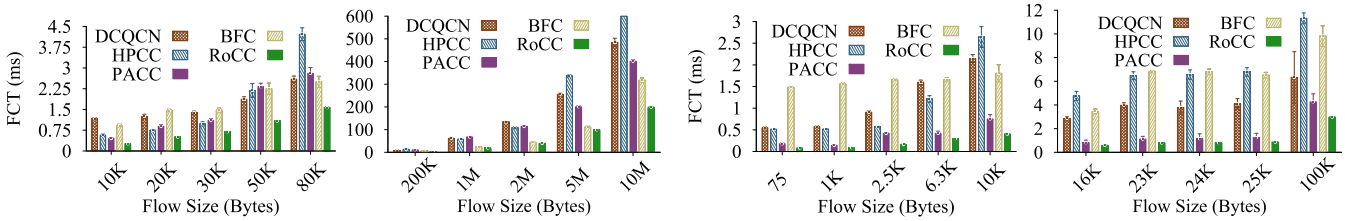
In the second scenario, we configure the sending rate of the 3 clients to 40 Gb/s, 30 Gb/s, and 10 Gb/s, respectively. Fig. 14(a) shows that Q_{cur} stabilizes at 75 KB for both the testbed (testbed-mix) and simulation (sim-mix) results. Fig. 14(b) shows that the flows attain the max-min fair value of 15 Gb/s in both testbed and simulation experiments.

It is important that *RoCC* be validated under real-life constraints in datacenter networks, i.e., with a real protocol stack, latency introduced by different layers, and NIC transmission delays, which can all adversely affect its behavior. From the



(a) *WebSearch* traffic. (b) *FB_Hadoop* traffic.

Fig. 15. Average FCT of DCQCN, HPCC, PACC, BFC, and *RoCC* (70% average load).



(a) *WebSearch* traffic. (b) *FB_Hadoop* traffic.

Fig. 16. 99th percentile FCT of DCQCN, HPCC, PACC, BFC, and *RoCC* (70% average load).

DPDK-based and P4-based evaluation, we conclude that *RoCC* would behave as expected under these constraints, and the results of the simulations are representative.

D. Large-Scale Simulations

We use large-scale simulations to evaluate *RoCC* and compare it with DCQCN [1], HPCC [3], PACC [16], and BFC [17], in terms of (1) FCT and (2) PFC activation. We use a two-level fat-tree [38] topology with 3 core switches and 3 edge switches. Each edge switch is connected to each core switch using 2 100 Gb/s links (i.e., 200 Gb/s effective bandwidth). Each edge switch has 30 nodes connected to it using 40 Gb/s links (i.e., 2:1 oversubscription). We implement ECMP on the edge switches to equally distribute the load across the links. Each node behind the first two edge switches transmits traffic to every node behind the third edge switch. As a result, the maximum incast levels are 150, 300, and 60 on ingress edge switches, core switches, and egress edge switches, respectively. This setup is sufficiently large to represent a production datacenter network in terms of bisection bandwidth, incast congestion level, and number of CPs. We use traffic loads derived from two publicly available datacenter traffic distributions consisting of throughput-sensitive large flows [7], [8] (*WebSearch* traffic) and latency-sensitive small flows [8], [9] (*FB_Hadoop* traffic). We run our simulations using 50% and 70% average link load levels. Besides FCT and number of PFC activations at CPs, we record flow-level rate at sources and buffer usage on CPs to rationalize our observations on FCT and PFC activation. We repeat each experiment 5 times, each on a machine with a fresh simulation environment setup. The FCTs, PFC counts, and queue sizes we present are the average values of the 5 sets of results, with 95% confidence intervals.

a) FCT: Figs. 15 and 16 respectively show the average and 99th percentile FCT of DCQCN, HPCC, PACC, BFC, and *RoCC* for *WebSearch* traffic and *FB_Hadoop* traffic at 70% average load. We chose the flow sizes (bins) based on the flow size distributions of *WebSearch* traffic and *FB_Hadoop* traffic. Based on the 99th percentile FCT, *RoCC* clearly outperforms DCQCN, HPCC, PACC, and BFC for all the flow sizes. Therefore, the results confirm that *RoCC* has lower tail latency than DCQCN, HPCC, PACC, and BFC. Especially, *RoCC* experiences very low tail latency even for large flows (i.e., elephants) where HPCC clearly fails. This is the behavior expected of HPCC [3] as a result of headroom bandwidth it loses and the INT information it piggybacks on data frames. These two overheads of HPCC reduce effective throughput for large flows, hence increasing tail latency. HPCC shows a different behavior for large flows with *FB_Hadoop* traffic. In this case, the FCT of HPCC increases significantly and, in particular, the 99th percentile FCT is an order of magnitude higher than that of DCQCN and PACC. One important observation here is that FCTs (both average and 99th percentile) of BFC with *FB_Hadoop* traffic – mostly has small flows – are significantly higher than those of other solutions. We believe this to be due to excessive back pressure imposed by BFC (due to small back pressure thresholds) that more severely impacts short flows than long flows. *RoCC* has very low FCTs compared to DCQCN, HPCC, PACC, and BFC resulting in much lower tail latency than that of DCQCN, HPCC, PACC, and BFC. At 50% load, the results are consistent with those at 70% load.

To understand the FCTs of the three solutions, we study flow-level rate allocation in the three solutions. Any given data flow in our setup traverses *four* links from its source to destination. The bandwidth of these links are 40 Gb/s, 100 Gb/s, 100 Gb/s, and 40 Gb/s with maximum concurrent flows of 30, 150, 300, and 60, respectively. Based on these values, the maximum per-flow bandwidth a flow can attain is $\sim$ 333 Mb/s (i.e., 100 Gb/s $\div$ 300). We use the flow-level rate values we recorded for *FB_Hadoop* traffic under 70% load, which mostly consists of short flows and as a result, the average number of concurrent flows on each bottleneck link is close to the corresponding numbers we

TABLE III
FLOW-LEVEL AVERAGE RATE ALLOCATION OF DCQCN, HPCC, AND *RoCC* WITH *FB_HADOOP* TRAFFIC (70% AVERAGE LOAD). THE IDEAL AVERAGE RATE IN THIS CASE IS ∼ 333 MB/S

Solution	Average rate (Mb/s)	Standard deviation (Mb/s)
DCQCN	378.86	2635.63
HPCC	211.02	1459.04
PACC	301.33	382.18
BFC	208.76	1341.67
RoCC	335.86	232.52

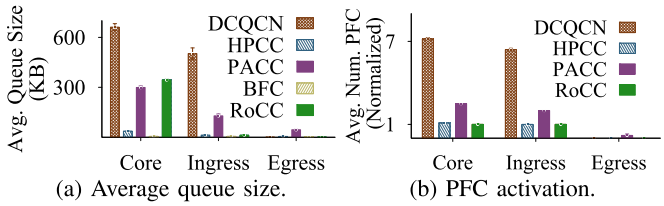


Fig. 17. Queue size and PFC activation of DCQCN, HPCC, PACC, BFC and *RoCC* with *WebSearch* traffic (70% average load).

mentioned above, and the bottleneck link utilization is close to link bandwidth. Table III shows the average per-flow rate and their variances for the three solutions. The average rate for *RoCC* closely matches the ideal value with low variance. In contrast, the average rate values for DCQCN, HPCC, and BFC deviate from the ideal value with very high variance. However, PACC closely follows *RoCC* in terms of average per-flow rate. Based on this analysis, it is clear that the fairness (FAIR), stability (STBL), and convergence (CONV) of *RoCC* allow a flow to constantly attain the optimal bandwidth along its path from source to destination, resulting in lower tail latency than that of DCQCN and HPCC, regardless of flow size. In essence, our simulation setup does not prioritize flows, and the fairness of *RoCC* ensures that flows of the same size exhibit low variance in their FCT (in other words, almost equal average and 99th percentile FCT).

b) Shallow Versus Stable Queues: HPCC is based on the idea that shallow queues reduce congestion signal delay, and hence convergence delay. To examine this, we analyze the average queue size at different CPs for the five solutions. Fig. 17(a) shows the average queue size of DCQCN, HPCC, PACC, BFC, and *RoCC* at potential CPs: core switches, ingress edge switches, and egress edge switch, for *WebSearch* traffic. DCQCN clearly experiences congestion at two different CPs (core and ingress edge), yielding poor performance. In contrast, HPCC experiences congestion only at a single CP (core) with a very shallow queue (mild congestion). Similarly to HPCC, *RoCC* experiences congestion at a single CP (core) with an average queue size close to its reference (Q_{ref}) of 300 KB. Though HPCC maintains a shallow queue at the CP, it has higher overall FCTs than *RoCC*. Therefore, we argue that maintaining a stable queue is more effective than maintaining a shallow queue at the expense of link underutilization. PACC maintains shallower queues than DCQCN. We believe this to be due to the use of a PI controller that improves queue stability, and backward notifications that reduce convergence time. Due to small back pressure thresholds in BFC, it maintains very shallow queues. The queue sizes for *FB_Hadoop* traffic is consistent with those for *WebSearch* traffic.

c) PFC activation: Fig. 17(b) shows the normalized average numbers of PFC activations at different CPs for DCQCN, HPCC, PACC, and *RoCC* at 70% average load, for *WebSearch* traffic (the number of PFC activations is for a segment of experiment duration, and we divide it into 50 segments). DCQCN suffers from high levels of PFC activation, whereas HPCC, PACC, and *RoCC* do not. This observation agrees with the queue size observation (Fig. 17(c)), where DCQCN has deep queues, whereas HPCC has shallow queues. In contrast, *RoCC* maintains a stable queue at the CP. Furthermore, PACC has significantly shallower and more stable queues than DCQCN does, and that justifies the notable reduction in PFC activation in PACC compared to DCQCN.

The PFC activation for *FB_Hadoop* traffic is consistent with that for *WebSearch* traffic. We do not include BFC in this scenario as it eliminates the need for PFC by employing its own link-level back pressure mechanism.

d) Key takeaways: From these experiments, we can make the following key observations: (i) *RoCC* is more fair (FAIR) than DCQCN, HPCC, PACC, and BFC. For *WebSearch* traffic, the tail FCTs of *RoCC* are $1.7-4.5\times$, $1.4-3.9\times$, $1.6-3.6\times$, and $1.1-3.5\times$ lower than those of DCQCN, HPCC, PACC, and BFC, respectively. For *FB_Hadoop* traffic, they are $2.1-7\times$, $3.5-8.2\times$, $1.4-2.6\times$, and $3.3-19.1\times$ lower than those of DCQCN, HPCC, PACC, and BFC, respectively, and (ii) *RoCC* is more stable (STBL) than DCQCN, HPCC, PACC, and BFC, and as a result, *RoCC* causes up to $7\times$ and $2\times$ reduction in PFC activation, compared to DCQCN and PACC, respectively.

VII. RELATED WORK

Although congestion control research started as far back as the 1980 s, several new proposals for the Internet (e.g., [39], [40], [41], [42], [43]) and datacenters (e.g., [1], [2], [3], [4], [7], [15], [44], [45]) have emerged over the past few years. Table I and Section I summarized the most widely-known datacenter solutions.

With sender-driven congestion control solutions individual senders determine their sending rates or windows based on congestion signals they receive. DCQCN [1] is one such widely deployed solution. TIMELY [2] is another production-grade solution that uses RTT as congestion signal. We have shown that they do not meet convergence and fairness goals. HPCC [3] uses INT supported by modern network switches to gather link load information and uses it to adjust sending window sizes at sources. HPCC is more stable than other datacenter solutions, but we saw that it becomes notably unstable at high load levels and is unfair when multiple bottlenecks exist or the network topology is asymmetric. In addition, HPCC causes link underutilization due to the bandwidth headroom it retains and the INT transmission overhead it incurs.

Switch-driven solutions have the advantage of precisely measuring congestion, and directly sending critical congestion control information to sources, using special control messages that can be prioritized so that sources can quickly react. However,

most existing solutions underutilize this advantage by relying on end-to-end congestion feedback that incurs delays. QCN [13] measures the extent of congestion at the switch, and conveys this to the source using multiple bits (as opposed to a single bit in the case of ECN). QCN is limited to layer 2. XCP [26] achieves efficiency (link utilization) and fair bandwidth allocation on the switch, by making adjustments to window size information in packet headers. The window adjustments are relayed by the receiver, causing feedback delay. XCP requires substantial modifications to end systems, switches, and packet headers. RCP [27] requires the switch to calculate a fair-share rate per link and have each data packet carry the minimum fair-share rate along its path from the source to destination and back to the source. As a result, RCP suffers from rate message propagation delay, just like XCP. TFC [46] uses a token-based bandwidth allocation mechanism at the switch, based on the number of active flows at each time interval. It is difficult to measure the quantities TFC uses in its computation, especially with bursty datacenter traffic. PACC [16] has higher stability and faster convergence than DCQCN through using a PI controller and generating CNPs. However, it suffers from the same fairness issues that DCQCN does. BFC [17] uses per-hop per-flow flow control, but incurs high implementation and control message overhead and increases FCTs, especially for small flows. Floodgate [47] is a flow control solution that aims to mitigate incast congestion, and *supplements* congestion control by reducing queue occupancy, hence queuing delay, on the switch. Overall, existing switch-based solutions do not satisfy the requirements of datacenter networks, and fail to realize the full potential of operating at the CP where it is possible to compute and provide the source with the fair rate instead of congestion information. In contrast, *RoCC* employs a closed-loop control system at the switch, enabling rapid convergence to the fair rate. The rate value is conveyed to the source using a special ICMP message that can be prioritized. *RoCC* only sends feedback to those flows that cause congestion.

VIII. Conclusion

Programmable switch architectures with P4 support becoming more widespread has motivated us to explore switch-driven congestion control in datacenter networks. We have proposed *RoCC*, a new switch-driven congestion control solution for RDMA. *RoCC* employs a closed-loop control system that uses the queue size as input to compute a fair rate through the egress port, maintaining a stable queue. *RoCC* is fair and efficient, yields low tail latency, and reduces PFC activation, even when flows traverse multiple bottlenecks or the topology becomes asymmetric over time due to changes that are inevitable as datacenter networks evolve. *RoCC* also allows datacenter networks to be run at higher load levels than the state of the art. Our plans for future work include additional experiments to compare *RoCC* with a wider variety of congestion control approaches, with emphasis on QoS, where class-level fairness is essential. The *RoCC* code repository is available at [48].

References

[1] Y. Zhu et al., "Congestion control for large-scale RDMA deployments," in *Proc. ACM Conf. Special Int. Group Data Commun.*, 2015, pp. 523–536.

[2] R. Mittal et al., "TIMELY: RTT-based congestion control for the datacenter," in *Proc. ACM Conf. Special Int. Group Data Commun.*, 2015, pp. 537–550.

[3] Y. Li et al., "HPCC: High precision congestion control," in *Proc. ACM Conf. Special Int. Group Data Commun.*, 2019, pp. 44–58.

[4] R. Mittal et al., "Revisiting network support for RDMA," in *Proc. ACM Conf. Special Int. Group Data Commun.*, 2018, pp. 313–326.

[5] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. Internet Meas. Conf.*, 2010, pp. 267–280.

[6] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of datacenter traffic: Measurements & analysis," in *Proc. Internet Meas. Conf.*, 2009, pp. 202–208.

[7] M. Alizadeh et al., "Data Center TCP (DCTCP)," in *Proc. ACM Conf. Special Int. Group Data Commun.*, 2010, pp. 63–74.

[8] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout, "Homa: A receiver-driven low-latency transport protocol using network priorities," in *Proc. ACM Conf. Special Int. Group Data Commun.*, 2018, pp. 221–235.

[9] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (Datacenter) network," in *Proc. ACM Conf. Special Int. Group Data Commun.*, 2015, pp. 123–137.

[10] I. D. Standard, "802.1Qbb - Priority-based flow control," 2011. [Online]. Available: https://1.ieee802.org/dcb/802-1qbb/

[11] S. Hu et al., "Deadlocks in datacenter networks: Why do they form, and how to avoid them," in *Proc. ACM Workshop Hot Topics Netw.*, 2016, pp. 92–98.

[12] K. Qian, W. Cheng, T. Zhang, and F. Ren, "Gentle flow control: Avoiding deadlock in lossless networks," in *Proc. ACM Conf. Special Int. Group Data Commun.*, 2019, pp. 75–89.

[13] M. Alizadeh et al., "Data center transport mechanisms: Congestion control theory and IEEE standardization," in *Proc. Annu. Allerton Conf. Commun., Control, Comput.*, 2008, pp. 1270–1277.

[14] Y. Zhu, M. Ghobadi, V. Misra, and J. Padhye, "ECN or delay: Lessons learnt from analysis of DCQCN and TIMELY," in *Proc. Int. Conf. Emerg. Netw. Experiments Technol.*, 2016, pp. 313–327.

[15] Y. Gao, Y. Yang, T. Chen, J. Zheng, B. Mao, and G. Chen, "Taming large-scale incast congestion in RDMA over ethernet networks," in *Proc. IEEE 26th Int. Conf. Netw. Protoc.*, 2018, pp. 110–120.

[16] X. Zhong, J. Zhang, Y. Zhang, Z. Guan, and Z. Wan, "PACC: Proactive and accurate congestion feedback for RDMA congestion control," in *Proc. IEEE Conf. Comput. Commun.*, 2022, pp. 2228–2237.

[17] P. Goyal, P. Shah, K. Zhao, G. Nikolaidis, M. Alizadeh, and T. E. Anderson, "Backpressure flow control," in *Proc. 19th USENIX Symp. Netw. Syst. Des. Implementation*, Renton, WA, 2022, pp. 779–805.

[18] P. Bosshart et al., "P4: Programming protocol-independent packet processors," in *Proc. ACM Conf. Special Int. Group Data Commun.*, 2014, pp. 87–95.

[19] S. Ibanez, G. Antichi, G. Brebner, and N. McKeown, "Event-driven packet processing," in *Proc. ACM Workshop Hot Topics Netw.*, 2019, pp. 133–140.

[20] R. Joshi, B. Leong, and M. C. Chan, "TimerTasks: Towards time-driven execution in programmable dataplanes," in *Proc. ACM SIGCOMM Conf. Posters Demos*, 2019, pp. 69–71.

[21] J. Sonchack, A. J. Aviv, E. Keller, and J. M. Smith, "Turboflow: Information rich flow record generation on commodity switches," in *Proc. Eur. Conf. Comput. Syst.*, 2018, pp. 1–16.

[22] G. Franklin, D. Powell, and A. Emami-Naeini, *Feedback Control of Dynamic Systems*, Englewood Cliffs, NJ, USA: Prentice Hall, 1995.

[23] P. Taheri, D. Menikkumbura, E. Vanini, S. Fahmy, P. Eugster, and T. Edsall, "ROCC: Robust congestion control for rdma," in *Proc. 16th Int. Conf. Emerg. Netw. Experiments Technol.*, 2020, pp. 17–30.

[24] R. Pan, L. Breslau, B. Prabhakar, and S. Shenker, "Approximate fairness through differential dropping," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 33, no. 2, pp. 23–39, 2003.

[25] R. Pan et al., "Pie: A lightweight control scheme to address the bufferbloat problem," in *Proc. IEEE Int. Conf. High Perform. Switching Routing*, 2013, pp. 148–155.

[26] D. Katabi, M. Handley, and C. Rohrs, "Congestion control for high bandwidth-delay product networks," in *Proc. ACM Conf. Special Int. Group Data Commun.*, 2002, pp. 89–102.

[27] C.-H. Tai, J. Zhu, and N. Dukkipati, "Making large scale deployment of RCP practical for real networks," in *Proc. IEEE Int. Conf. Comput. Commun.*, 2008, pp. 2180–2188.

[28] Y. Lu, M. Wang, B. Prabhakar, and F. Bonomi, "ElephantTrap: A low cost device for identifying large flows," in *Proc. IEEE Symp. High- Perform. Interconnects*, 2007, pp. 99–108.
[29] J. Ros-Giralt, A. Commike, S. Maji, and M. Veeraraghavan, "High speed elephant flow detection under partial information," in *Proc. IEEE Int. Symp. Netw., Comput. Commun.*, 2018, pp. 1–7.
[30] "Pktgen," 2021. [Online]. Available: https://pktgen-dpdk.readthedocs.io/en/latest/index.html
[31] "Xilinx VitisHLS," 2020. [Online]. Available: https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/vitis_hls_process.html
[32] "OMNeT discrete event simulator," 2018. [Online]. Available: https://omnetpp.org/
[33] E. Vanini, R. Pan, M. Alizadeh, P. Taheri, and T. Edsall, "Let it flow: Resilient asymmetric load balancing with flowlet switching," in *Proc. USENIX Conf. Netw. Syst. Des. Implementation*, 2017, pp. 407–420.
[34] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz, "DeTail: Reducing the flow completion time tail in datacenter networks," in *Proc. ACM Conf. Special Int. Group Data Commun.*, 2012, pp. 139–150.
[35] "DPDK Intel," 2019. [Online]. Available: http://dpdk.org/
[36] D. Duplyakin et al., "The design and operation of CloudLab," in *Proc. USENIX Annu. Tech. Conf.*, 2019, pp. 1–14.
[37] "iPerf," 2019. [Online]. Available: http://iperf.fr/
[38] C. E. Leiserson, "Fat-trees: Universal networks for hardware-efficient supercomputing," *IEEE Trans. Comput.*, vol. 34, no. 10, pp. 892–901, Oct. 1985.
[39] V. Arun and H. Balakrishnan, "Copa: Practical delay-based congestion control for the internet," in *Proc. USENIX Conf. Networked Syst. Des. Implementation*, 2018, pp. 329–342.
[40] P. Goyal, M. Alizadeh, and H. Balakrishnan, "Rethinking congestion control for cellular networks," in *Proc. ACM Workshop Hot Topics Netw.*, 2017, pp. 29–35.
[41] K. Winstein and H. Balakrishnan, "TCP ex machina: Computer-generated congestion control," in *Proc. ACM Conf. Special Int. Group Data Commun.*, 2013, pp. 123–134.
[42] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "BBR: Congestion-based congestion control," *ACM Queue*, vol. 14, no. 5, pp. 20–53, 2016.
[43] M. Dong, Q. Li, M. Schapira, D. Zarchy, and P. Brighten Godfrey, "PCC: Re-architecting congestion control for consistent high performance," in *Proc. USENIX Conf. Netw. Syst. Des. Implementation*, 2015, pp. 395–408.
[44] M. Alizadeh, A. Kabbani, T. Edsall, and B. Prabhakar, "Less is more: Trading a little bandwidth for ultra-low latency in the data center," in *Proc. USENIX Conf. Netw. Syst. Des. Implementation*, 2012, pp. 253–266.
[45] S. Huang, D. Dong, and W. Bai, "Congestion control in high-speed lossless data center networks: A survey," *Future Gener. Comput. Syst.*, vol. 89, pp. 360–374, 2018.
[46] J. Zhang, F. Ren, R. Shu, and P. Cheng, "TFC: Token flow control in data center networks," in *Proc. Eur. Conf. Comput. Syst.*, 2016, pp. 1–14.
[47] K. Liu et al., "Floodgate: Taming incast in datacenter networks," in *Proc. 17th Int. Conf. Emerg. Netw. Experiments Technol.*, 2021, pp. 30–44.
[48] "RoCC Repository," 2020. [Online]. Available: https://github.com/danushkam/rocc

Danushka Menikkumbura received the BSc degree in computer science and engineering from the University of Moratuwa, Sri Lanka. He is currently working toward the PhD degree in computer science with Purdue University, advised by professor Sonia Fahmy and Professor Patrick Eugster. His research focuses on improving the efficiency and robustness of datacenter networks, particularly using the programmability of modern datacenter network hardware. For more information he can be contacted at mailto:dmenikku@purdue.edu.

Parvin Taheri received the bachelor's degree in electrical engineering from the University of Tehran, and the master's degree from the Tandon school of NYU with the focus on computer networks. She joined an R&D team, Cisco and worked on multiple projects which resulted in 5 inventions on congestion control algorithms for data center switches. She is currently a part of the Meta AI team. For more information she can be contacted at mailto:pt873@nyu.edu.

Erico Vanini received the MS degree in computer science from EPFL, Switzerland, in 2015. He has been working for Cisco Systems since 2014, initially in the Insieme CTO office, and in 2019 he switched to a more customer facing role. His areas of interest are, load balancing, traffic optimization and automation. In these fields he has few patents and peer-reviewed papers. For more information he can be contacted at mailto:evanini@cisco.com.

Sonia Fahmy (Fellow, IEEE) received the PhD degree from the Ohio State University, in 1999, She is a professor of computer science with Purdue University. Her research interests lie in the design and evaluation of network architectures and protocols. She is a recipient of an NSF CAREER award (2003). For more information she can be contacted at mailto:fahmy@purdue.edu.

Patrick Eugster received the MS and PhD degrees from EPFL, in 1998 and 2001, respectively. He is a professor of computer science with the Università della Svizzera Italiana (USI), and an adjunct faculty member with Purdue University. His research interest lies in networked distributed systems and programming languages, and in particular in the intersection of the two. He is a recipient of an NSF CAREER award (2007) and an ERC Consolidator award (2012). For more information he can be contacted at mailto:eugstp@usi.ch.

Tom Edsall received the bachelor's and master's degrees in electrical engineering from Stanford University, where he enjoys guest lecturing from time to time. He is a Cisco fellow and Emeritus advisor, and was formerly the CTO of the Data Center Business Group, Cisco Systems. He was a co-founder of Insieme Networks, acquired by Cisco in November 2013. He also served as general manager of the Cisco Data Center Business Unit responsible for the MDS and Nexus 7000 product lines, and served as CTO and co-founder of Andiamo Systems, another Cisco spin-in. Tom is particularly passionate about innovation in the network and holds more than 100 patents. For more information he can be contacted at mailto:edsall@cisco.com.