RoCC: Robust Congestion Control for RDMA

Parvin Taheri
Cisco Systems

Danushka Menikkumbura
Purdue University

Erico Vanini
Cisco Systems

Sonia Fahmy
Purdue University

Patrick Eugster
Università della Svizzera italiana
TU Darmstadt
Purdue University

Tom Edsall
Cisco Systems

Abstract

In this paper, 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, signaling it to the flow sources. 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× reduction in PFC frames generated under high average load levels, compared to DCQCN. At the same time, *RoCC* can achieve up to 8× lower tail latency, compared to DCQCN and HPCC. We also find that *RoCC* does not require PFC. The functional components of *RoCC* are implementable in P4-based and fixed-function switch ASICs.

CCS Concepts

• **Networks** → **Transport protocols**; **Data center networks**.

Keywords

Datacenter; Congestion Control; RDMA; Network Programmability

ACM Reference Format:
Parvin Taheri, Danushka Menikkumbura, Erico Vanini, Sonia Fahmy, Patrick Eugster, and Tom Edsall. 2020. *RoCC*: Robust Congestion Control for RDMA. In *The 16th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '20), December 1–4, 2020, Barcelona, Spain.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3386367.3431316

1 Introduction

Congestion control in packet-switched networks has a clear goal: to reduce flow completion time (FCT) for users by providing *low latency for small flows (mice)* and *high throughput for large flows (elephants)*. Typical datacenter networks have simple topologies with fixed distances (in contrast to the Internet) and fixed bisection bandwidth (in contrast to wireless networks), which may make congestion control in datacenter networks seem simple. It turns out to be quite the opposite, though, as evidenced by the spectrum of solutions that exploit different congestion signals [26, 44], leverage latest developments in network hardware [24], and revisit previous work with a new perspective [27].

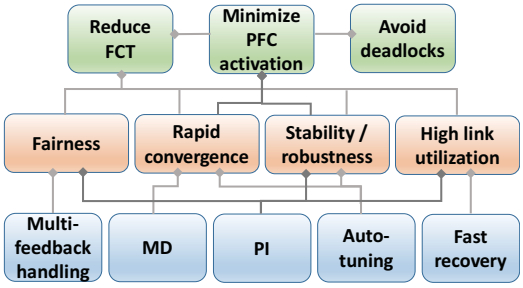


Figure 1: Relationship of the components (§3) of *RoCC* to its requirements (§2) and high-level goals (§1).

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 [2, 5, 21, 28, 35]. 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 [44] has made operators switch to transports based on remote direct memory access (RDMA); kernel bypass transports such as RDMA over converged Ethernet v2 (RoCEv2) reduce FCT by orders of magnitude compared to traditional TCP/IP stacks. RDMA requires losslessness, triggering the need for priority-based flow control (PFC) [37], which prevents packet drop by using back-pressure (at the traffic class level). Alas, PFC has been observed to cause problems such as head-of-line (HoL) blocking, congestion spreading, and routing deadlocks [15, 24, 26, 27, 44]. Less aggressive flow control mechanisms [32] 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 [24], and the many RDMA congestion control solutions developed over the recent past, e.g., [24, 26, 27, 44], 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 §2), and foreshadows the components of our solution and how these fulfill the requirements.

CoNEXT '20, December 1–4, 2020, Barcelona, Spain

ACM ISBN 978-1-4503-7948-9/20/12…$15.00
https://doi.org/10.1145/3386367.3431316

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 gets from the network (switches and/or receiver). With (b), the switch computes the pacing information (usually rate) and sends it to the source.

Table 1 summarizes the most widely-known datacenter congestion control solutions. A very popular choice in production datacenter networks is datacenter QCN (DCQCN) [44]. DCQCN is a source-driven congestion control approach for RoCEv2, which adapts the congestion point (CP) algorithm of quantized congestion notification (QCN), using the explicit congestion notification (ECN) 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 [13, 45]. TIMELY [26] is another source-driven solution that uses delay as the congestion signal, but it falls behind DCQCN in terms of stability and fairness [45]. Several enhancements, e.g., DCQCN+proportional integral (PI) [45], DCQCN+ [13], and patched TIMELY [45], have been proposed, but they (too) fail to meet important properties such as stability and fairness, which affect FCT.

The recently proposed high precision congestion control (HPCC) [24] is a source-driven, window-based solution leveraging in-band network telemetry (INT) 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 (Fig. 6.1) — commonly observed in modern datacenter networks [44]. HPCC also trades link bandwidth for shallow queues and further loses link bandwidth by carrying INT information.

Path forward. We posit that source-driven solutions cannot receive congestion signals (e.g., ECN in DCQCN [44], network delay in [26], and INT in HPCC [24]) 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 application specific integrated circuit (ASIC) is high are being overturned by the increasing adoption of programmable switch ASICs with P4 support [6] by leading switch manufacturers. Similarly, reservations that switch-driven congestion control can hinder line-rate packet processing are countered by recent work [17, 20, 36] showing that event processing (beyond packet arrival and departure events) using P4 does not sacrifice line-rate packet processing.

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 PI controller [12], (ii) signals that fair rate to the sources via Internet control message protocol (ICMP), and (iii) auto-tunes the control parameters to ensure stability and responsiveness.

Our contributions can be summarized as follows:

(1) After establishing important design requirements (§ 2) we present the design of *RoCC* (§3).
(2) We analyze *RoCC*, and show how quantized auto-tuning allows trading off stability and rapid convergence under a variety of network conditions (§5).
(3) We evaluate *RoCC* via simulations and a DPDK implementation (§6) and compare it to DCQCN, TIMELY, and HPCC. Not only does *RoCC* achieve fairness and queue stability, but, compared to DCQCN and HPCC, it also reduces FCT for real datacenter workloads.

§7 summarizes related work, and §8 concludes the paper. App. A includes additional evaluation results.

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

2 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 [24].

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. I.e., 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 (core and edge), switch heterogeneity (vendor, configuration), or the number of nodes connected to edge switches (load). Fair rates 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 §3).

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

Table 1: Comparison of selected congestion control solutions.
(*solution-specific, CNP: congestion notification packet)

Solution	Switch action	Source action	Destination action
DCTCP [2]	Mark ECN	Adjust congestion window based on ECN	Echo ECN
QCN [1]	Compute and send F_b* to source	Compute rate based on F_b	None
DCQCN [44]	Mark ECN	Compute rate based on CNP	Send CNP to source
TIMELY [26]	None	Send RTT probes and compute rate based on RTT	Echo RTT probes
HPCC [24]	Inject INT	Adjust sending window based on INT	Echo INT
RoCC	Compute and send rate to source	Use minimum rate received from switch(es)	None

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.

3 *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 [30] (PI controller), QCN [1] (multi-bit feedback), PIE [31] (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).

3.1 Definitions

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

Table 2 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 §3.2. 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.

3.2 CP Algorithm

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

Table 2: 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 §3.3)
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*

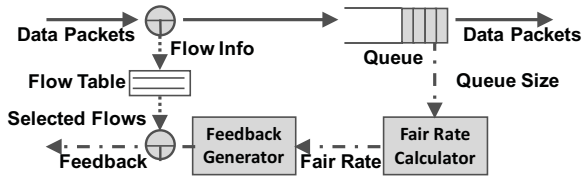


Figure 2: Overview of *RoCC* design at the CP.

§3.3). 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 short 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 $Q_{\max} > Q_{\mathrm{mid}} > Q_{\mathrm{ref}}$, 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 [30], PIE [31], and QCN [1]. 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 ($Q_{\mathrm{cur}} - Q_{\mathrm{old}}$), 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 §5).

Algorithm 1 Fair rate computation at the CP

```
1:  function CALCULATE_FAIR_RATE(Q_cur)
2:      if Q_cur ≥ Q_max AND F > F_max/8 then
3:          F ← F_min
4:      else if (Q_cur − Qold) ≥ Q_mid AND F > F_max/8 then
5:          F ← F ÷ 2
6:      else
7:          α, β ← AUTO_TUNE()
8:          F ← F − α × (Q_cur − Q_ref) − β × (Q_cur − Q_old)
9:      if F > F_max then
10:         F ← F_max
11:     if F < F_min then
12:         F ← F_min
13:     Q_old ← Q_cur
14:     return F
15: function AUTO_TUNE()
16:     level ← 2
17:     while F < F_max/level AND level < 64 do
18:         level ← level × 2
19:     ratio ← level ÷ 2
20:     α ← α̃ ÷ ratio; β ← β̃ ÷ ratio
21:     return α, β
```

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

3.3 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 RP can correctly match the feedback message to the relevant RL. We use ICMP for the congestion notification packet (CNP) and prioritize 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 [44], delay in TIMELY [26], and INT in HPCC [24]).

3.4 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 [30], the first AQM mechanism that leveraged flow size and flow rate distributions to scale per-flow state.
(4) ElephantTrap [25]: 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 [34]: 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:

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

where C_l is the bandwidth of the congested link, BW_{mice} is the total bandwidth used by the flows that do not contribute to congestion (innocent/mice flows), and N is the number of flows contributing to congestion. Thus it suffices to track the flows that most contribute to congestion and queue buildup.

3.5 RP Algorithm

The RP employs an event-driven 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).

Alg. 2 shows the RP algorithm which has two routines:

(1) 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).

(2) 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.

3.6 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).

Algorithm 2 Rate limiting at the RP

```
1:  procedure PROCESS_CNP(cnp)
2:      R_rcvd ← GETRATE(cnp) × Δ^F
3:      CP_rcvd ← GETCP(cnp)
4:      if R_rcvd ≤ R_cur OR CP_rcvd = CP_cur then
5:          R_cur ← R_rcvd
6:          CP_cur ← CP_rcvd
7:          RESET_TIMER()
8:  procedure TIMER_EXPIRED()
9:      if R_cur > R_max then
10:         remove this rate limiter if its queue is empty
11:     else
12:         R_cur ← R_cur × 2
13:         RESET_TIMER()
```

4 Implementation

In this section, we investigate the feasibility of implementing the CP algorithm at the switch and the RP algorithm at the host.

4.1 Basics

CP. The key components of the CP implementation include: (1) flow table (hash generation and table update), (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.

RoCC can be implemented on custom ASICs. Based on our discussions with switch ASIC designers, a proprietary fixed-function ASIC implementation of *RoCC* requires 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.

As data plane programmability becomes more widespread, P4 [6] is becoming the de facto framework for programming switch ASICs, and major datacenter switch hardware vendors already support P4 programmability in their ASICs [7].

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

4.2 P4 Implementation

Fig. 3 illustrates a potential switch implementation of *RoCC* (rate computation is at the host). We use "v1model"[1] as the data plane runtime architecture. Runtime models (e.g., Portable Switch Architecture (PSA), Tofino Native Architecture (TNA)) closely follow the v1model (with additional features), hence our implementation is adaptable to other P4 runtime models. Below, we walk through our 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 standard approach for implementing timer events in P4 that uses the "packet out" channel (using P4Runtime API) to input the control packet to the data plane through its "CPU port." List. 1 defines the P4 header for CNP. We send a CNP packet for each egress port and specify port id as meta data

[1] `struct standard_metadata_t` defines runtime meta data in "v1model."

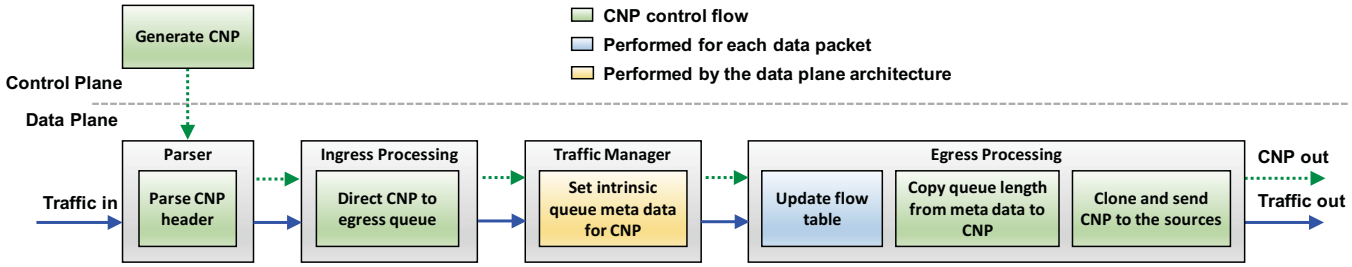


Figure 3: *RoCC* switch implementation in P4.

when we call the P4Runtime API. Other runtime architectures may have more efficient ways of implementing timer events. For instance, Tofino can periodically generate control packets within the data plane (i.e., ingress packet generator), hence our solution can be implemented entirely in the Tofino data plane.

(2) Parser: extracts `packetout_t` header from an incoming CNP. We assume the data plane only receives CNPs from the control plane (i.e., through the "CPU port"). Therefore, a CNP can be identified from its ingress port (`ingress_port` of `standard_metadata_t`).

(3) Ingress processing: directs CNPs to their respective egress queues. Only CNPs have a valid `packetout_t` header (i.e., `isValid()` on `packetout` returns `true`). A CNP is directed to its intended egress queue by setting `egress_spec` of `standard_metadata_t` to `packetout.egress_port`.

(4) Traffic manager: attaches certain intrinsic meta information including egress queue length (`deq_qdepth` of `standard_metadata_t`) to every passing packet. As a result, a CNP has its egress queue length when it reaches the egress pipeline.

(5) Egress processing: handles two tasks: (i) maintain a flow table. Our flow table is implemented using a simple `Register` array in P4 and it is very similar to the flow table used in Turboflow [36], 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 §3.6) on CNP and "mirror" it to selected sources based on the flow table.

```
@controller_header("packet_out")
header packetout_t {
    bit<8> egress_port;
}
struct headers {
    packetout_t packetout;
    /* standard headers */
    ethernet_t  ethernet;
    ipv4_t      ipv4;
    icmp_t      icmp;
}
```

Listing 1: P4 header definition for CNP.

5 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.

5.1 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 [31], we can convert the operation at Line 8 from Alg. 1 into the continuous form

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

After a Laplace transformation on Eq. 2 and Eq. 3 we get

$$Q(s) = \kappa N \frac{e^{-sT}}{s} F(s), \quad (4) \qquad F(s) = \frac{(\beta + \alpha/2)s + \frac{\alpha}{T}}{s} E(s), \quad (5)$$

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.

5.2 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 round trip time (RTT). 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.

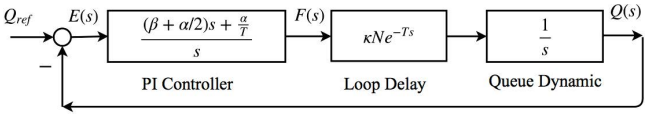


Figure 4: The feedback loop of *RoCC* controller that includes PI, queue dynamic, and loop delay.

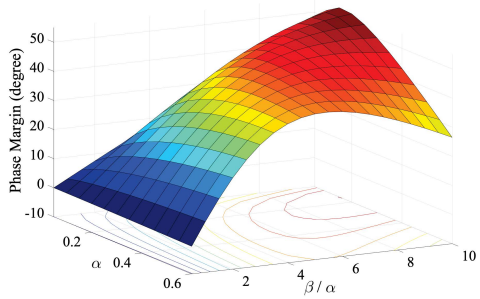


Figure 5: Phase margin as a function of parameters α and β. Phase margin above 0 ensures a stable system.

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

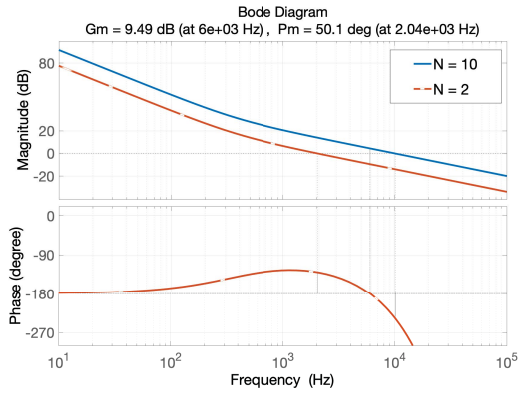


Figure 6: Stability margin for two values of N. For $N > 2$, the system gain increases. Thus 0 dB gain occurs at higher frequency resulting in lower phase margin.

sharing the link is small, it takes a long time for the PI controller to reach the stable fair rate.

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

5.3 Auto-tuning α and β

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 Eq. 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.

Discrete α and β values: Adjustments to α and β can be made in a continuous fashion. As Fig. 7a and Fig. 7b 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 [31] 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 (§ 6) show that auto-tuning increases stability and reduces convergence time for a variety of workloads.

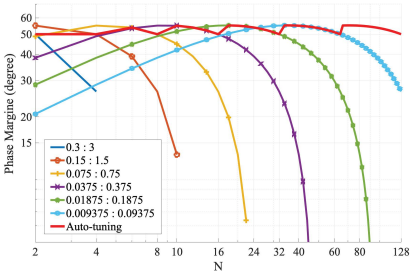
6 Evaluation

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

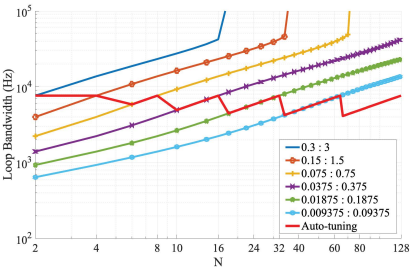
(1) Micro-benchmarks and comparisons to the state of the art with respect to the four properties in § 2 using simulations.
(2) Evaluation with DPDK to confirm the properties of *RoCC* on a real network and validate our simulations.
(3) Larger scale evaluation using a simulation setup resembling a real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

For simulations, we use the INET model suite [18] on OMNeT++ [29] event simulator to implement a prototype of *RoCC*. In our simulation model, the fair rate has fixed point precision to mimic hardware implementation. Our datacenter model has been previously used in the literature [40], 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 [2, 28, 35].

System parameters. All interconnections in our simulations are either 40 Gb/s or 100 Gb/s links with 1.5 μs delay. We chose PFC threshold values 500 KB and 800 KB for 40 Gb/s and 100 Gb/s links, respectively, based on [42]. NIC (i.e., RP) reaction delay for feedback



(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.

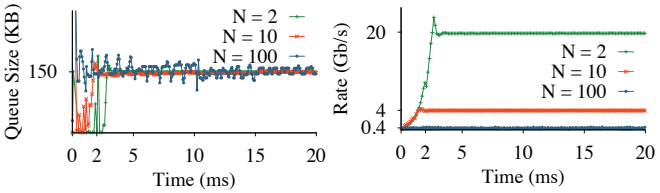
Figure 7: Impact of the number of flows N with different values of α and β.

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 4000 and 10000 for 40 Gb/s and 100 Gb/s links, respectively. We set Q_{ref}, Q_{mid}, and $Q_{\max}$ to 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. §3.4 (1)).

6.1 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.

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



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

Figure 8: Fairness and stability of *RoCC* as load increases.

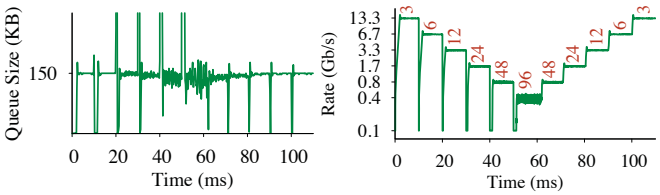
is consistent with the auto-tuning discussion in §5.2. Results for B = 100 Gb/s are consistent with those for B = 40 Gb/s.

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 (CONV). 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. When new flows start and create a traffic burst in Fig. 9a, 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.

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 properties. We include QCN [1], DCQCN [44], DCQCN+PI [45], TIMELY [26], and HPCC [24] in our 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. 11a shows, the flows experience a significant deviation from the expected average fair rate of 4 Gb/s with TIMELY. In contrast, each flow attains the expected average fair rate with *RoCC* (FAIR). In this scenario, DCQCN and HPCC 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. In the next section, we further examine the fairness of DCQCN, HPCC, and *RoCC*.

Fig. 11b and Fig. 11c 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 for DCQCN and TIMELY, respectively. A key observation here is that DCQCN's stability improves significantly when its ECN marking mechanism is modified to use a feedback loop based on a PI controller (DCQCN+PI [45]). This observation further justifies the use of a PI controller 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



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

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

case are consistent with its expected behavior. The stability of the rate attained by each flow closely follows that of link utilization.

Multiple bottlenecks. A flow in a datacenter network may encounter multiple CPs on its path from the source to destination [44]. Intuitively, a flow that traverses multiple CPs must attain the fair bandwidth corresponding to the most congested CP on the path of the flow (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 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 [44] and HPCC [24] in terms of flow-level bandwidth allocation. As Fig. 12a 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. We conclude that *RoCC* is best at handling feedback messages received from multiple CPs (FAIR).

Asymmetric topology. As equipment of datacenters is gradually upgraded over time, their topologies become less symmetrical. We thus use a simple network topology with asymmetric links to compare *RoCC* to DCQCN [44] and HPCC [24] in terms of average 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 (i.e., D0…D6, respectively), destined to B0. A0…A4 and A5…A6 should get the same total bandwidth (40 Gb/s × 5 and 100 Gb/s × 2, respectively) through S0 and S1, respectively (2:1 oversubscription). We run the experiment at 90% load to record average throughput attained by D0…D4 and D5…D6.

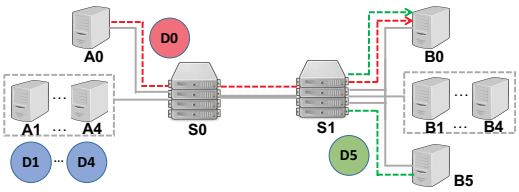


Figure 10: Multi-bottleneck topology.

We make an interesting observation. As the bottleneck link in this topology (S2 → B0) is shared among the 7 flows, each flow should obtain a fair bandwidth share of 14.29 Gb/s. Fig. 12b 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.

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.

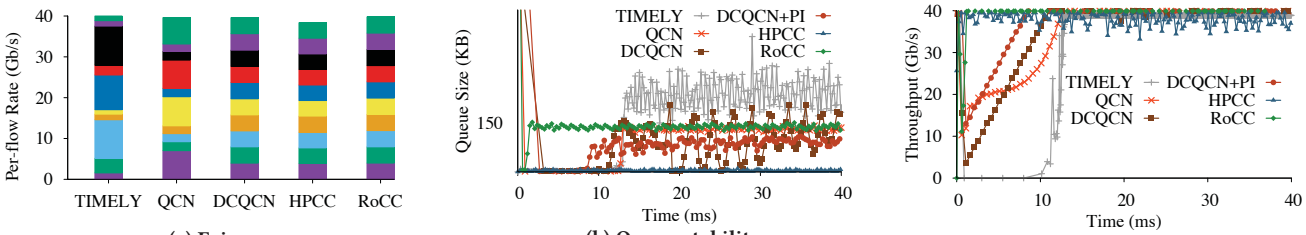
6.2 DPDK Evaluation

We implement *RoCC* using the popular DPDK [10] 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 [11] 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 GbE NICs. With this configuration, our switch is capable of working as a 10 GbE 4-port switch. We use an iPerf [19] 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. 13a shows that the queue size stabilizes at 75 KB for both the testbed (testbed-uni) and simulation (sim-uni). In Fig. 13b, 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. 13a shows that Q_{cur} stabilizes at 75 KB for both the testbed (testbed-mix) and simulation (sim-mix) results. Fig. 13b shows that the flows attain the max-min fair value of 6 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 DPDK



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

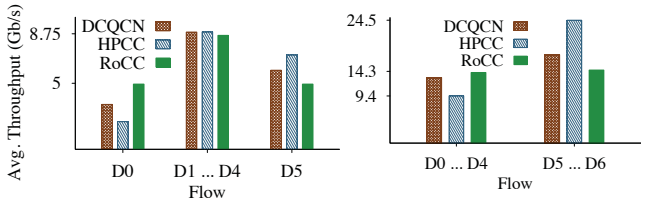
Figure 11: Comparing *RoCC* with TIMELY, QCN, DCQCN, and HPCC in terms of fairness, stability, and convergence.

evaluation, we conclude that *RoCC* would behave as expected under these constraints, and the results of the simulations are valid.

6.3 Large-Scale Simulations

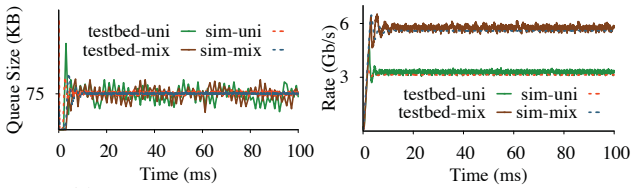
We use large-scale simulations to evaluate *RoCC* and compare it with DCQCN [44] and HPCC [24], in terms of (1) FCT and (2) PFC activation. We use a two-level fat-tree [23] topology with 3 core switches and 3 edge switches. Each edge switch is connected to each core switch using 2 100 GbE links (i.e., 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 [2, 28] (*WebSearch* traffic) and latency-sensitive small flows [28, 35] (*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.

FCT. Fig. 14, Fig. 15, and Fig. 16 respectively show the average, 90^{th} percentile, and 99^{th} percentile FCT of DCQCN, HPCC, and *RoCC* for *WebSearch* traffic and *FB_Hadoop* traffic at 70% average load. We chose the flow sizes (i.e., bins) based on the heavy-tailed flow size distributions of *WebSearch* traffic and *FB_Hadoop* traffic. Based on the 99^{th} percentile FCT, *RoCC* clearly outperforms



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

Figure 12: Fairness of DCQCN, HPCC, and *RoCC*.

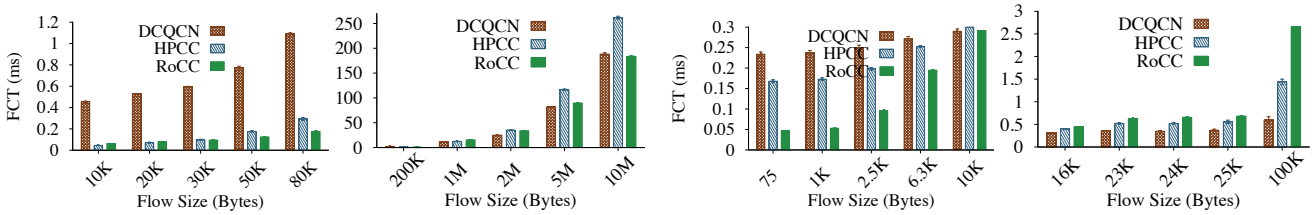


(a) Queue stability. (b) Fair rate.

Figure 13: Testbed results vs. simulation results.

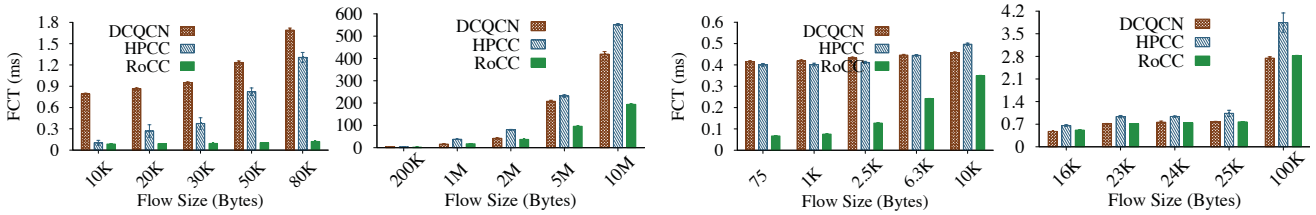
DCQCN and HPCC for all the flow sizes. Therefore, the results confirm that *RoCC* has lower tail latency than DCQCN and HPCC. Especially, *RoCC* experiences very low tail latency even for large flows (i.e., elephants) where HPCC clearly fails. This is the behavior expected of HPCC [24] as a result of 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 99^{th} percentile FCT is an order of magnitude higher than that of DCQCN. *RoCC* has very low FCTs compared to DCQCN and HPCC resulting in much lower tail latency than that of DCQCN and HPCC. 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 ~ 333 Mb/s (i.e., 100 Gb/s ÷ 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 mentioned above, and the bottleneck link utilization is close to link bandwidth. Table 3 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 and HPCC deviate from the ideal value with very high variance. 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



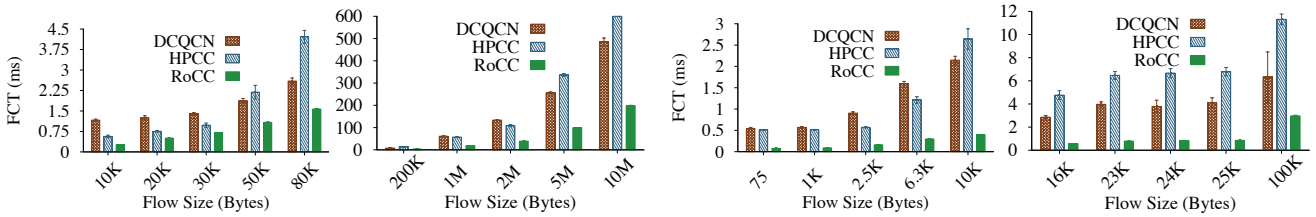
(a) *WebSearch* traffic. **(b) *FB_Hadoop* traffic.**

Figure 14: Average FCT of DCQCN, HPCC, and *RoCC* (70% average load).



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

Figure 15: 90^{th} percentile FCT of DCQCN, HPCC, and *RoCC* (70% average load).



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

Figure 16: 99^{th} percentile FCT of DCQCN, HPCC, and *RoCC* (70% average load).

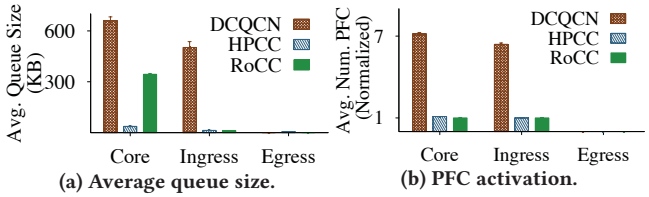
same size exhibit low variance in their FCT (in other words, almost equal average, 90^{th} percentile, and 99^{th} percentile FCT).

Shallow vs. 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 three solutions. Fig. 17a shows the average queue size of DCQCN, HPCC, 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. The queue sizes for *FB_Hadoop* traffic is consistent with those for *WebSearch* traffic.

PFC activation. Fig. 17b shows the normalized average numbers of PFC activations at different CPs for DCQCN, HPCC, 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 and *RoCC* do not. This observation agrees with the queue size observation (Fig. 17a), where DCQCN has deep queues at the CPs, whereas HPCC has shallow queues. In contrast, *RoCC* maintains a stable queue at the CP. The PFC activation for *FB_Hadoop* traffic is consistent with that for *WebSearch* traffic.

Table 3: 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
RoCC	335.86	232.52



(a) Average queue size. **(b) PFC activation.**

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

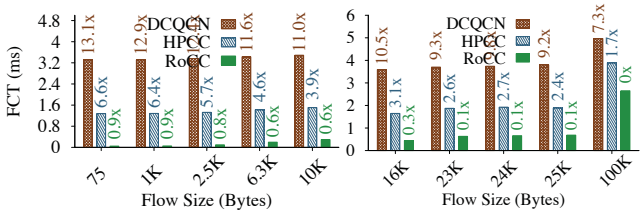


Figure 18: Average FCT of DCQCN, HPCC, and *RoCC* with PFC disabled and unlimited buffer (*FB_Hadoop* traffic at 70% average load). The numbers in respective colors show the fold increase in FCT, w.r.t. the case when PFC is enabled with limited buffer (Fig. 16).

Unlimited buffer. We now examine the behavior of DCQCN, HPCC, and *RoCC* with unlimited buffers on switches. We use the same network setup as before, with PFC disabled and relatively large amount of buffer space on the switches (i.e., no packet drop) with *FB_Hadoop* traffic. We record FCT and average buffer usage at the CPs for the three solutions. Datacenter networks do not operate under these conditions in practice, but we investigate the extent of each solution's buffer demand in order to estimate the amount of buffer space a switch needs for the solution to function (1) without activating PFC, and (2) without dropping packets and retransmissions. We also examine the impact on FCT in this situation. Fig. 18 shows the FCTs of the solutions at 70% load. The FCT of DCQCN increases up to a factor of ~ 13 and that of HPCC increases up to a factor of ~ 7. In contrast, the FCTs of *RoCC* remains close to the FCTs when PFC is enabled with limited buffer (see Fig. 14). *RoCC* maintains its average buffer usage around the reference value (Q_{ref}) of 300 KB, whereas DCQCN and HPCC on average use ~ 80× and ~ 20× more buffer space, respectively, than *RoCC* does. This demonstrates *RoCC*'s ability to operate without PFC.

Key takeaways. From these experiments, we can make the following key observations: (i) *RoCC* is more fair (FAIR) than DCQCN and HPCC. For *WebSearch* traffic, the tail FCTs of *RoCC* are up to ~ 4× and ~ 3× lower than those of DCQCN and HPCC, respectively. For *FB_Hadoop* traffic, they are up to ~ 7× and ~ 8× lower than those of DCQCN and HPCC, respectively., and (ii) *RoCC* is more stable (STBL) than DCQCN and HPCC, and as a result *RoCC* causes up to ~ 7× reduction in PFC activation, compared to DCQCN.

7 Related Work

Although congestion control research started as far back as the 1980s, several new proposals for the Internet (e.g., [4, 8, 9, 14, 41]) and datacenters (e.g., [2, 3, 13, 16, 24, 26, 27, 38, 44]) have emerged over the past few years. Table 1 and §1 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 [44] is one such widely deployed solution. TIMELY [26] is another production-grade solution that uses RTT as congestion signal. We have shown that they do not meet convergence and fairness goals. HPCC [24] 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. QCN [1] 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 [22] 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 [39] 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 [43] 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. 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.

8 Conclusions

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 egress 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 topology 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 [33].

Acknowledgments

We thank our shepherd and anonymous reviewers for their feedback that helped improve our paper. This work has been supported in part by NSF grants CCR-1618923 and CNS-1717493, by SNSF grant 200021_192121 ("FORWARD"), and DFG center 1053 ("MAKI").

References

[1] Mohammad Alizadeh, Berk Atikoglu, Abdul Kabbani, Ashvin Lakshmikantha, Rong Pan, Balaji Prabhakar, and Mick Seaman. 2008. Data Center Transport Mechanisms: Congestion Control Theory and IEEE Standardization. In *Annual Allerton Conference on Communication, Control, and Computing*.

[2] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data Center TCP (DCTCP). In *ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*. https://doi.org/10.1145/1851182.1851192

[3] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, and Balaji Prabhakar. 2012. Less is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center. In *USENIX Conference on Networked Systems Design and Implementation (NSDI)*.

[4] Venkat Arun and Hari Balakrishnan. 2018. Copa: Practical Delay-Based Congestion Control for the Internet. In *USENIX Conference on Networked Systems Design and Implementation (NSDI)*. https://doi.org/10.1145/3232755.3232783

[5] Theophilus Benson, Aditya Akella, and David A Maltz. 2010. Network Traffic Characteristics of Data Centers in the Wild. In *Internet Measurement Conference (IMC)*. https://doi.org/10.1145/1879141.1879175

[6] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming Protocol-independent Packet Processors. *ACM Conference on Special Interest Group on Data Communication (SIGCOMM)* 44, 3 (2014). https://doi.org/10.1145/2656877.2656890

[7] Mihai Budiu and Chris Dodd. 2017. The P416 Programming Language. *ACM SIGOPS Operating Systems Review* 51, 1 (2017). https://doi.org/10.1145/3139645.3139648

[8] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2016. BBR: Congestion-Based Congestion Control. *ACM Queue* 14, 5 (2016). https://doi.org/10.1145/3012426.3022184

[9] Mo Dong, Qingxi Li, Michael Schapira, Doron Zarchy, and P Brighten Godfrey. 2015. PCC: Re-architecting Congestion Control for Consistent High Performance. In *USENIX Conference on Networked Systems Design and Implementation (NSDI)*. https://doi.org/10.5555/2789770.2789798

[10] dpdk 2019. Intel DPDK. http://dpdk.org/.

[11] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *USENIX Annual Technical Conference (ATC)*.

[12] Gene Franklin, David Powell, and Abbas Emami-Naeini. 1995. *Feedback Control of Dynamic Systems*.

[13] Yixiao Gao, Yuchen Yang, Tian Chen, Jiaqi Zheng, Bing Mao, and Guihai Chen. 2018. Taming Large-scale Incast Congestion in RDMA over Ethernet Networks. In *IEEE International Conference on Network Protocols (ICNP)*. https://doi.org/10.1109/ICNP.2018.00021

[14] Prateesh Goyal, Mohammad Alizadeh, and Hari Balakrishnan. 2017. Rethinking Congestion Control for Cellular Networks. In *ACM Workshop on Hot Topics in Networks (HotNets)*. https://doi.org/10.1145/3152434.3152437

[15] Shuihai Hu, Yibo Zhu, Peng Cheng, Chuanxiong Guo, Kun Tan, Jitendra Padhye, and Kai Chen. 2016. Deadlocks in datacenter networks: Why do they form, and how to avoid them. In *ACM Workshop on Hot Topics in Networks (HotNets)*. https://doi.org/10.1145/3005745.3005760

[16] Shan Huang, Dezun Dong, and Wei Bai. 2018. Congestion Control in High-speed Lossless Data Center Networks: A Survey. *Future Generation Computer Systems* 89 (2018). https://doi.org/10.1016/j.future.2018.06.036

[17] Stephen Ibanez, Gianni Antichi, Gordon Brebner, and Nick McKeown. 2019. Event-Driven Packet Processing. In *ACM Workshop on Hot Topics in Networks (HotNets)* (Princeton, NJ, USA) *(HotNets '19)*. Association for Computing Machinery, 8 pages. https://doi.org/10.1145/3365609.3365848

[18] INET 2018. INET Framework. https://inet.omnetpp.org/.

[19] iperf 2019. iPerf. http://iperf.fr/.

[20] Raj Joshi, Ben Leong, and Mun Choon Chan. 2019. TimerTasks: Towards Time-driven Execution in Programmable Dataplanes. In *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos*.

[21] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. 2009. The Nature of Datacenter Traffic: Measurements & Analysis. In *Internet Measurement Conference (IMC)*. https://doi.org/10.1145/1644893.1644918

[22] Dina Katabi, Mark Handley, and Charlie Rohrs. 2002. Congestion Control for High Bandwidth-Delay Product Networks. In *ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*. https://doi.org/10.1145/633025.633035

[23] Charles E Leiserson. 1985. Fat-trees: Universal Networks for Hardware-efficient Supercomputing. *IEEE Trans. Comput.* 100, 10 (1985).

[24] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. 2019. HPCC: High Precision Congestion Control. In *ACM Conference on Special Interest Group on Data Communication (SIGCOMM)* (Beijing, China). ACM, 15 pages. https://doi.org/10.1145/3341302.3342085

[25] Yi Lu, Mei Wang, Balaji Prabhakar, and Flavio Bonomi. 2007. ElephantTrap: A low cost device for identifying large flows. In *IEEE Symposium on High-Performance Interconnects (HOTI)*. https://doi.org/10.1109/HOTI.2007.13

[26] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. 2015. TIMELY: RTT-based Congestion Control for the Datacenter. *ACM Conference on Special Interest Group on Data Communication (SIGCOMM)* 45, 4 (2015). https://doi.org/10.1145/2785956.2787510

[27] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. 2018. Revisiting Network Support for RDMA. In *ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*, Vol. 18. https://doi.org/10.1145/3230543.3230557

[28] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. 2018. Homa: A receiver-driven low-latency transport protocol using network priorities. In *ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*. https://doi.org/10.1145/3230543.3230564

[29] OMNeT++ 2018. OMNeT++ Discrete Event Simulator. https://omnetpp.org/.

[30] Rong Pan, Lee Breslau, Balaji Prabhakar, and Scott Shenker. 2003. Approximate Fairness through Differential Dropping. *ACM SIGCOMM Computer Communication Review* 33 (2003). Issue 2. https://doi.org/10.1145/956981.956985

[31] Rong Pan, Preethi Natarajan, Chiara Piglione, Mythili Suryanarayana Prabhu, Vijay Subramanian, Fred Baker, and Bill VerSteeg. 2013. PIE: A Lightweight Control Scheme to Address the Bufferbloat Problem. In *IEEE International Conference on High Performance Switching and Routing (HPSR)*. IEEE. https://doi.org/10.1109/HPSR.2013.6602305

[32] Kun Qian, Wenxue Cheng, Tong Zhang, and Fengyuan Ren. 2019. Gentle Flow Control: Avoiding Deadlock in Lossless Networks. In *ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*. https://doi.org/10.1145/3341302.3342065

[33] RoCC 2020. RoCC Repository. https://github.com/danushkam/rocc.

[34] Jordi Ros-Giralt, Alan Commike, Sourav Maji, and Malathi Veeraraghavan. 2018. High Speed Elephant Flow Detection Under Partial Information. In *IEEE International Symposium on Networks, Computers and Communications (ISNCC)*. IEEE. https://doi.org/10.1109/ISNCC.2018.8530979

[35] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. 2015. Inside the Social Network's (Datacenter) Network. In *ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*. https://doi.org/10.1145/2785956.2787472

[36] John Sonchack, Adam J Aviv, Eric Keller, and Jonathan M Smith. 2018. Turboflow: Information Rich Flow Record Generation on Commodity Switches. In *European Conference on Computer Systems (EuroSys)*. https://doi.org/10.1145/3190508.3190558

[37] IEEE Draft Standard. 2008. 802.1Qbb - Priority-based Flow Control.

[38] Brent Stephens, Alan L Cox, Ankit Singla, John Carter, Colin Dixon, and Wesley Felter. 2014. Practical DCB for Improved Data Center Networks. In *IEEE International Conference on Computer Communications (INFOCOM)*. https://doi.org/10.1109/INFOCOM.2014.6848121

[39] C-H Tai, Jiang Zhu, and Nandita Dukkipati. 2008. Making Large Scale Deployment of RCP Practical for Real Networks. In *IEEE International Conference on Computer Communications (INFOCOM)*. IEEE. https://doi.org/10.1109/INFOCOM.2008.285

[40] Erico Vanini, Rong Pan, Mohammad Alizadeh, Parvin Taheri, and Tom Edsall. 2017. Let It Flow: Resilient Asymmetric Load Balancing with Flowlet Switching.. In *USENIX Conference on Networked Systems Design and Implementation (NSDI)*.

[41] Keith Winstein and Hari Balakrishnan. 2013. TCP ex Machina: Computer-Generated Congestion Control. In *ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*. https://doi.org/10.1145/2486001.2486020

[42] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy Katz. 2012. DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks. In *ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*. https://doi.org/10.1145/2342356.2342390

[43] Jiao Zhang, Fengyuan Ren, Ran Shu, and Peng Cheng. 2016. TFC: Token Flow Control in Data Center Networks. In *European Conference on Computer Systems (EuroSys)*. https://doi.org/10.1145/2901318.2901336

[44] 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 Conference on Special Interest Group on Data Communication (SIGCOMM)* 45, 4 (2015). https://doi.org/10.1145/2829988.2787484

[45] Yibo Zhu, Monia Ghobadi, Vishal Misra, and Jitendra Padhye. 2016. ECN or Delay: Lessons Learnt from Analysis of DCQCN and TIMELY. In *International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*. https://doi.org/10.1145/2999572.2999593

A Additional Evaluation Results

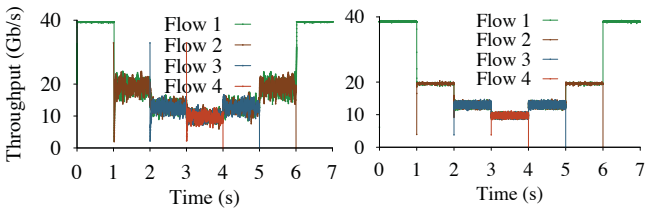
In this appendix, we include additional experimental results that are omitted from the paper for brevity.

A.1 Verification of reference solution implementations

We verify our implementations of DCQCN and HPCC in terms of their stability, convergence, and fairness, which are the key congestion control properties that impact FCT and PFC activation. We use the topology of §6.1. We increase (and decrease) the link load level by dynamically starting (and stopping) flows. We start with 1 flow (i.e., $N = 1$) and start a new flow every 1 s until $N = 4$. Then, after 1 s, we begin to stop a flow every 1 s until $N = 1$. We record the instantaneous rate attained by each flow. As Fig. 19a shows, the per-flow rate of DCQCN decreases from 40 Gb/s to 10 Gb/s as N increases from 1 to 4, and it increases from 10 Gb/s to 40 Gb/s as N decreases from 4 to 1. As Fig. 19b shows, the per-flow rate of HPCC follows that of DCQCN. HPCC is more stable, fair, and converges faster than DCQCN. Fig. 9h and Fig. 9g of [24] show results consistent with ours (Fig. 19a and Fig. 19b respectively) for the same experiment. Therefore, we can ascertain that our implementations of DCQCN and HPCC have the expected behavior.

A.2 Lossy network

We extend the scenario of §6.3 to examine the behavior of *RoCC* with limited buffer space on the switch. We implement *go-back-N* as the loss recovery scheme in DCQCN, HPCC, and *RoCC*. We set the buffer limit on the switches to 3× the PFC threshold. We record the total number of retransmissions, along with the total number of packets transmitted. We observe that both DCQCN and HPCC incur ~ 30% retransmissions, resulting in significantly increased FCTs (see Fig. 20 for details). In contrast, *RoCC* experiences minimal retransmissions and as a result, FCTs very close to the ones with PFC enabled and limited buffer.



(a) DCQCN. (b) HPCC.

Figure 19: Verification of the implementations of DCQCN and HPCC.

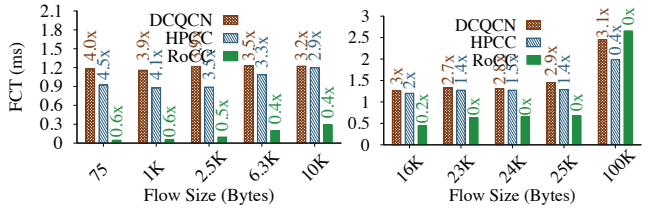


Figure 20: Average FCT of DCQCN, HPCC, and *RoCC* on a lossy network with *Go-back-N* loss recovery (*FB_Hadoop* traffic at 70% average load). The numbers in respective colors show the fold increase in FCT, w.r.t. the case when PFC is enabled with limited buffer (Fig. 16).