SOFTWARE-DEFINED BUFFER MANAGEMENT AND ROBUST CONGESTION CONTROL FOR MODERN DATACENTER NETWORKS

by

Danushka Menikkumbura

A Dissertation

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy

Department of Computer Science

West Lafayette, Indiana

May 2022

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Patrick Eugster, Co-Chair

Department of Computer Science

Dr. Sonia Fahmy, Co-Chair

Department of Computer Science

Dr. Muhammad Shahbaz

Department of Computer Science

Dr. Dongyan Xu

Department of Computer Science

Approved by:

Dr. Kihong Park

An inspiration to my son to do great things in life, and a tribute to my parents for everything they have done to make me who I am today

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to all those who helped me finish my PhD. Out of those there are some special individuals whom I would like to mention with a lot of love and respect.

First and foremost, I would like to express my heartfelt gratitude to two special people without whom my PhD journey would not have set out. I met Dr. Shahani Markus at a time I had given up on grad school, and without her motivation and encouragement my PhD would have been just a dream. I am forever in debt to Prof. Patrick Eugster, not only for expressing interest to have me as a student, but also, helping me lay a solid foundation to embark on my PhD journey.

I am profoundly thankful to Prof. Sonia Fahmy and Prof. Patrick Eugster for being the best PhD advisors one can ever have. Without their guidance, support, and mentorship, I would not have been able to successfully finish my PhD. I truly appreciate the amount of freedom they gave me to try new things. They have always been very friendly, supportive, and compassionate. Their remarkable knowledge and experience in the field helped me horn my research and literature skills, which will be key pillars of my future career. They have truly been my parents away from home who have always been there for me to share my ups and downs.

I very much appreciate Prof. Dongyan Xu and Prof. Muhammad Shahbaz for taking the time out of their busy schedules to serve on my exam committee and providing me with some invaluable feedback.

I am immensely grateful to Dr. Srinath Perera and Mr. Suresh Marru for writing me strong recommendations for grad school. I appreciate their helpful input in the grad school application process.

I am indebted to everyone at Purdue Computer Science department including the faculty and staff who assisted me complete my PhD qualifying exams. I would like to convey a special appreciation to Ms. Monica Shively and Ms. Lacey Siefers for all their support throughout my PhD.

With a heavy heart, I express my heartfelt gratitude to late Dr. Kirill Kogan. He was a great colleague who was extremely supportive and welcoming. I was fortunate to have a chance to collaborate with Dr. Kogan. I am thankful to Dr. Gustavo Petri, Dr. Yangtae Noh, Dr. Sergey Nikolenko, and Dr. Alexander Sirotkin for all their help and support during our collaboration.

I would like to respectfully express my sincere gratitude to Mr. Tom Edsall for giving me an opportunity to work with him. I was privileged and honored to be able to collaborate with Tom. I am thankful to Parvin Taheri and Erico Vanini for all their help and support while working on the project RoCC.

I am grateful to all the colleagues in my research group for their support, especially Amit Sheoran and Pavel Chuprikov for their insightful comments and feedback during our discussions. I am very much thankful to David Gengenbach and Marcel Blöcher for going out of their way to help me get access to their P4 testbed, without which, I would not have been able to complete some crucial experiments.

I am forever in debt to Dr. Sundar Iyer, Mr. Harsha Jagannati, and Dr. Ramana Kompella for helping me have a very productive internship at Cisco Systems. I appreciate all the fruitful discussions we had on using research skills in the industry. I owe them a special thank for introducing me to Mr. Tom Edsall, which created a lot of new opportunities for me.

I cannot thank my parents enough for everything they have done for me from the day I was born to this date. It brings tear to my eyes when I think of the amount of sacrifices my parents had to make to raise myself and my two siblings with lots of love and care. I am thankful to my brother Duleepa and sister Gayani for always being on my side. I would like to give special thanks to my in-laws for all their support.

Last but not least, I would like to give special thanks to my loving son Dihein and wife Dinushi for all their support and being my best friends all these years. Especially, I appreciate their tolerating all the hardships they had to go through while I was doing my PhD.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

PFC	Priority-based Flow Control
RDMA	Remote Direct Memory Access
FCT	Flow Completion Time
DPDK	Data Plane Development Kit
FPGA	Field-Programmable Gate Array
ICMP	Internet Control Message Protocol
ECN	Explicit Congestion Notification
INT	In-band Network Telemetry
ASIC	Application-Specific Integrated Circuit

ABSTRACT

Modern datacenter network applications continue to demand ultra low latencies and very high throughputs. At the same time, network infrastructure keeps achieving higher speeds and larger bandwidths. We still need better network management solutions to keep these two demand and supply fronts go hand-in-hand. There are key metrics that define network performance such as flow completion time (the lower the better), throughput (the higher the better), and end-to-end latency (the lower the better) that are mainly governed by how effectively network application get their fair share of network resources. We observe that buffer utilization on network switches gives a very accurate indication of network performance. Therefore, network buffer management is important in modern datacenter networks, and other network management solutions can be efficiently built around buffer utilization. This dissertation presents three solutions based on buffer use on network switches.

This dissertation consists of three main sections. The first section is on a specification language for buffer management in modern programmable switches. The second section is on a congestion control solution for Remote Direct Memory Access (RDMA) networks. The third section is on a solution to head-of-the-line blocking in modern datacenter networks.

1. INTRODUCTION

Modern datacenter applications demand very high throughput and ultra low latency that require datacenter networks to be very efficient and robust. At the same time network hardware keeps becoming faster (e.g. line-rate switches and 100Gb/s links) that makes efficient resource allocation within networks more challenging. Moreover, datacenter traffic is very bursty by nature and demands efficient buffer management and congestion control solutions to achieve small flow completion times for short-lived, heavy-tailed mice flows and persistent throughput for long-running elephant flows. This diversity in flow composition in datacenter networks also leads to issues like head-of-line blocking caused by buffer overrun due to congestion that in turn leads to other critical problems like routing deadlocks. The deadlocks in datacenter networks are catastrophic and can make a whole datacenter unusable. When that happens, the datacenter needs to be restarted, which is a very costly operation.

The focus of this dissertation is on improving efficiency and robustness of datacenter networks by finding solutions to the problems mentioned above. We try to address *three* problems in particular; (i) Buffer Management, (ii) Congestion Control, and (iii) Head-of-line Block Clearance. We briefly discuss what each of these problems is and why it is required to solve these problems in order to improve efficiency and robustness in datacenter networks.

1.1 Buffer Management

In-network buffering is an attribute of any packet-switched routing network. Network switches and routers use buffers to retain packets when there is congestion. Switch buffer management and active queue management (AQM) has evolved over time to meet different challenges including quality of service (QoS) and service-level agreements (SLA). With the advent of modern programmable line-rate switches, buffer management has become even more important. One major problem in switch buffer management is that it is very difficult for network administrators to configure and administer them with the limited set of tools available. Most of the existing tools are very low-level and error prone, hence lead to misconfiguration due to human errors. Modern high-speed switch architectures use *match-action* tables to execute rules. A match-action table entry has a condition check and a correspond-

ing action that executes a piece of logic if there is a match. To gain high performance and speed, these condition checks and actions should be simple and efficient. To gain maximum out of programmable switches, it is very important that buffer management rules are able to be modified dynamically. Therefore, high-level switch buffer architecture specification languages that are simple, expressive, efficient, and dynamic are a critical need, hence a growing area of research.

1.2 Congestion Control

Congestion control is essential in any packet-switched network to make sure proper functionality. Congestion control is required to provide fair share of network bandwidth across competing data flows and improve efficiency by reducing packets drops, hence retransmissions. With the advent of remote direct memory access (RDMA) networks, congestion control has become even more important. RDMA uses PFC for link-level flow control to make the network lossless. PFC can lead to problems like HoL blocking, congestion spreading, and routing deadlocks. Congestion control also plays an important role in datacenter networks where traffic is heterogeneous and demands a wide variety of quality of service. In datacenter networks, congestion control solutions are required to meet some important yet conflicting goals such as fast convergence, high stability, fairness, and minimal operation and implementation overheads.

1.3 Head-of-line Block Clearance

A routing deadlock occurs in a network when a loop forms in a resource dependency graph, where switch buffer space is usually the resource. When a node does not have sufficient buffer space to accept incoming traffic, it notifies the upstream node to temporarily shut the data flow until enough buffer space becomes available. This handshake between two adjacent nodes is handled by different protocols. The IEEE 802.3x standard defined Link-level Flow Control (LFC) for Ethernet networks and IEEE 802.1Qbb extends the same idea as part of Datacenter Bridging (DCB) standard to support fine-grained flow control based on traffic classes. Credit-based routing is another technique that uses the same idea where positive

credit denotes spare buffer space on a receiving node. When a deadlock occurs in a network it does not go away automatically and human intervention is required to resolve it. When a deadlock is formed, it can spread into other parts of the network bringing the whole network to a standstill. To resolve a deadlock, network nodes that are part of it need to be restarted and when there are multiple cascading deadlocks, it requires a whole datacenter network to be restarted and that is a very costly operation. Therefore, deadlock prevention/resolution in datacenter networks has got attention of the research community.

1.4 Thesis Statement

The thesis of this dissertation: *as modern datacenter network infrastructure continues to advance with programmable hardware and high-speed transports to achieve high-throughput and low-latency data transmission, there is a need for tools and network services to optimize network utilization. This includes: (a) a complete buffering architecture specification to program datacenter network switches more efficiently and effectively, (b) a new congestion control solution to meet throughput and latency demands while improving fair bandwidth allocation and network stability, and (c) an efficient solution to PFC-induced head-of-line (HoL) blocking, which is a critical problem in lossless datacenter networks. HoL-blocking increases latency and causes congestion spreading, leading to routing deadlocks.*

1.5 Contributions

This dissertation make three core contributions.

1. *OpenQueue*: A comprehensive specification language for defining complete buffering architectures and policies using simple operators.

2. *RoCC*: A switch-centric congestion control solution for RDMA networks, which is more stable, efficient, and fairer than the state of the art.

3. *Escape*: A solution to PFC-induced HoL blocking in datacenter networks.

2. SOFTWARE-DEFINED BUFFER MANAGEMENT

In this chapter, we present OpenQueue, a specification language for defining complete buffering architectures and policies. Buffering architectures and policies for their efficient management are core ingredients of a network architecture. However, despite strong incentives to experiment with and deploy new policies, opportunities for changing anything beyond minor elements are limited. We introduce a new specification language, OpenQueue, that allows to express virtual buffering architectures and management policies representing a wide variety of economic models. OpenQueue allows users to specify entire buffering architectures and policies conveniently through several comparators and simple functions. We show examples of buffer management policies in OpenQueue and empirically demonstrate its impact on performance in various settings.

2.1 Introduction

Buffering architectures define how input and output ports of a network element are connected [1]. Their design and management directly impact performance and cost of each network element. Traditional network management only allows to deploy a predefined set of buffer management policies with parameters that can adapt to specific network conditions. Incorporating new management policies requires complex control/data plane code and even hardware changes. Objectives beyond fairness [2] and additional traffic properties [3],[4],[5] lead to new challenges in the implementation and performance of switching architectures. Unfortunately, current developments in software-defined networking mostly eschew these challenges and concentrate on flexible and efficient representations of packet classifiers (e.g., OpenFlow [6]), which do not really capture buffer management. This calls for novel well-defined abstractions that enable buffer management policies to be deployed on real network elements at runtime. Design of such abstractions is hard, as they must satisfy possibly conflicting requirements: (1) `EXPRESSIVITY`: expressible policies should cover a large majority of existing and future deployment scenarios; (2) `SIMPLICITY`: policies for different objectives should be expressible concisely with a limited set of basic primitives and should not impose hardware choices; (3) `PERFORMANCE`: implementations of policies should

be efficient; (4) `DYNAMISM`: one should be able to specify new policies at runtime with no code changes or (re-)deployments.

2.2 OpenQueue Design

2.2.1 Overview

To specify an adequate language for software-defined buffer management, we need to identify primitive entities, their properties, and a logic to manipulate these primitives. The choice of primitives dictates SIMPLICITY and EXPRESSIVITY.

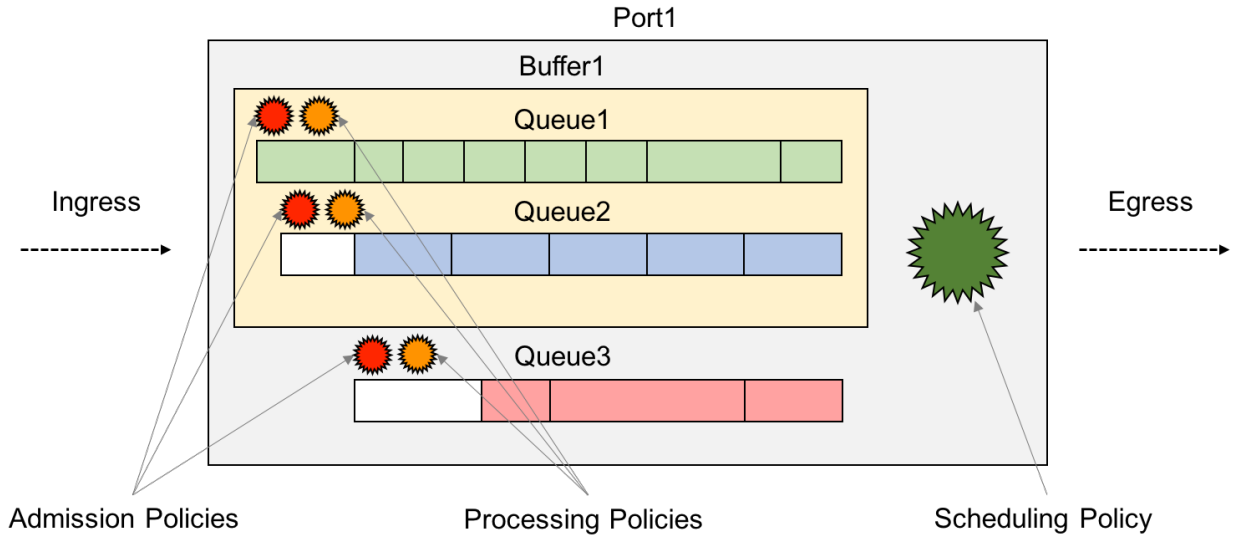


Figure 2.1. OpenQueue Schema

Fig. 2.1 depicts the main elements of OpenQueue schema. OpenQueue has two main types of object: *ports* and *queues* assigned to ports. Each queue has an *admission control policy*, deciding which packets are admitted or dropped [7],[8], and a *processing policy* defining its HoL packet; each port, a *scheduling policy*, selecting a queue whose HoL packet will be processed next [9],[10]. In some cases, (e.g. shared memory switches [11],[12],[13]), several queues share the same buffer space, and admission control can routinely query the state of several queues (e.g. the LQD policy under congestion drops packets from the longest queue [11]). Thus, OpenQueue deals with buffers and admission control policy to resolve congestion at the buffer level. Management policies for multi-level buffering architectures

can be implemented in a centralized or distributed manner, synchronously (e.g. finding a matching between input and output ports) [14],[15] or asynchronously, like packet scheduling in a buffered crossbar switch [16],[17],[15]. Specific implementations are beyond the scope of OpenQueue.

In summary, to define a buffering architecture and its management in OpenQueue, one needs to create instances of ports, queues, and buffers, and specify relations among them: admission control, processing, and scheduling policies.

2.2.2 Multiple Queues

A central primitive in our language is the *queue*. How complex should the queue abstraction and implementation be to achieve `EXPRESSIVITY`? In contrast to exploring a universal scheduling policy [18] that can satisfy multiple objectives instead of having a flexible interface to define new buffer management policies, we argue for separating admission, processing, and scheduling policies, and supporting multiple (as well as single) queues, through some novel fundamental results.

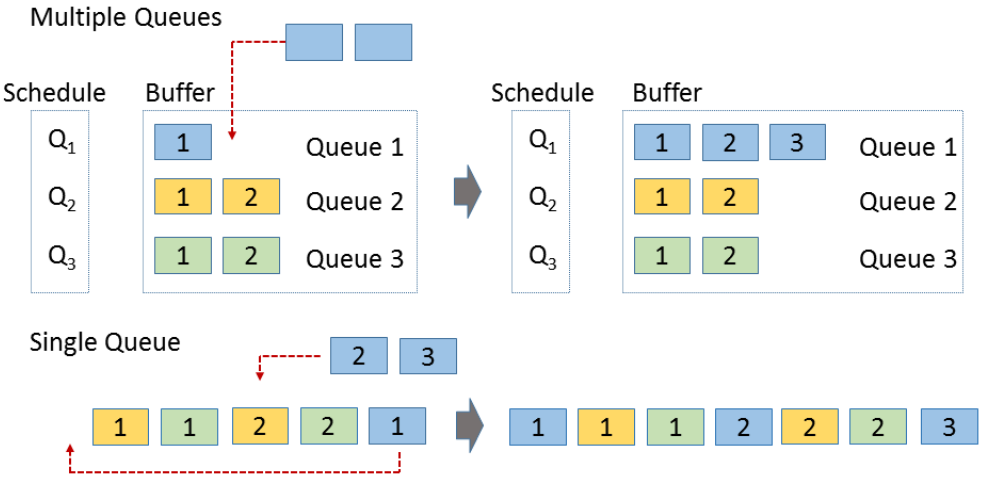


Figure 2.2. Operational time complexity: MQ vs. SQ.

Fig. 2.2 visually represents operational time complexity comparison between MQ and SQ. In the case of MA, the three incoming packets (in blue) can get inserted in to the

buffer without rearranging existing packets, whereas in the case of SQ it requires rearranging existing packets before inserting the new packets.

2.2.3 Priority Queue

Buffer management policies are generally concerned with *boundary conditions* (e.g., upon admission a packet belongs to the *lowest* priority class can be dropped). Hence, *priority queue* arises as a natural choice for implementing actions related to user-defined priorities. The priority criteria do not change at runtime (e.g., a queue's ordering cannot change from FIFO to LIFO). Thus, admission, processing, and scheduling policies in OpenQueue use priority queue data structures that uses a simple Boolean *comparator* to define its ordering.

2.2.4 Policy Framework

In this section we try to justify the way policies are structured in OpenQueue. OpenQueue provides three different points in the buffering architecture to attach policies. This in a way can be seen as decomposing a flat all-in-on policy into multiple sub policies based on three different key stages of the life cycle of a packet inside a switch buffer, (i) admission (accept the packet or not), (ii) processing (if accepted where to place it in the queue/buffer), and (iii) scheduling (packet is ready to exit or not). This level of policy decomposition improves `EXPRESSIVITY` of OpenQueue.

Impact of admission control: The modern network edge is required to perform tasks with heterogeneous complexity: deep packet inspection, firewalling, and intrusion detection. Hence, the way packets are processed may significantly affect desired objectives. For example, increasing per-packet processing time for some flows can trigger congestion even for traffic with relatively modest burstiness.

Consider throughput maximization in a single queue buffering architecture of size B, where each unit-sized and unit-valued packet is assigned the number of required processing cycles, ranging from 1 to k (see Fig. 2.3(a)). Defining a new admission control policy in OpenQueue requires only one comparator (admission order upon congestion) and one congestion condition (when an event of congestion occurs). The processing policy is defined

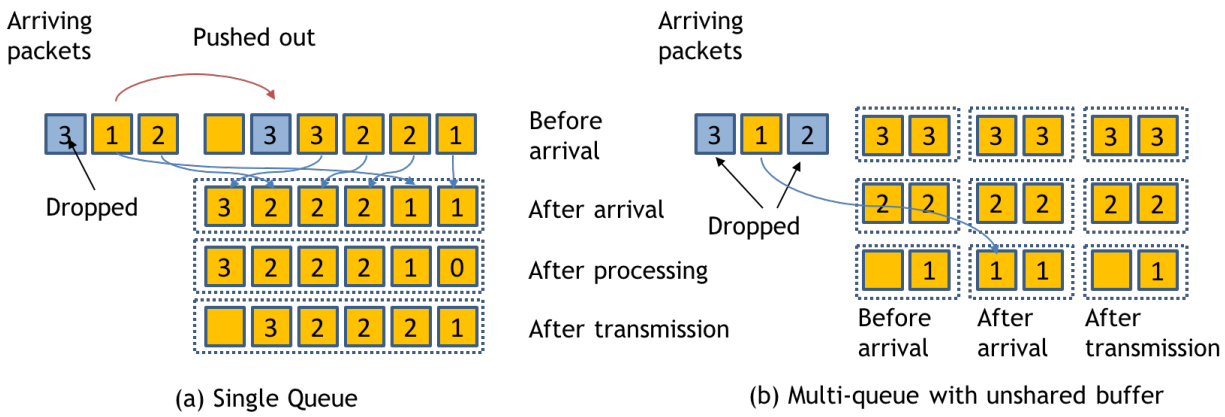


Figure 2.3. Left: single priority queue with buffer of size $B = 6$; right: multiple separated queues with three queues ($k = 3$) of size 2 each. Dashed lines enclose queues.

by one additional comparator (defining in which order packets are processed). Note that admission and processing comparators can be different. In this case, advanced processing and admission orders are not required as only packets with same processing requirements (remaining processing time) are admitted to the same queue.

Impact of scheduling: A common architecture for packets with heterogeneous processing requirements is to allocate a separate queue for packets with the same processing requirement; usually defined using traffic classes in practice, and use a scheduler to pick next packet from HoL of one of the queues. Common scheduling algorithms include Round-Robin (RR), Weighted-Round-Robin (WRR), Longest-Queue-First (LQF), Shorted-Queue-First (SQF), Shorted-Remaining-Processing-Time (SRPT), etc. Fig. 2.3(b) shows an example of using SRPT as scheduling policy to maximize throughput. The decision of scheduling next packet in order to maximize throughput could be non-trivial since it is unclear which characteristic (i.e., buffer occupancy, required processing, or a combination) is most relevant for throughput optimization.

2.3 OpenQueue Language

2.3.1 Rules

Comparators: The core data structure used in OpenQueue is *priority queue*. To express queues with different priorities, while abstracting actual implementations, data structures in OpenQueue (usually packets) are parameterized by a priority relation that determines their ordering in a queue. To that end, we introduce the notion of a *comparator*, a Boolean binary predicate (over different types). Since comparators are used internally by OpenQueue to implement the queues, the comparison operation has to be efficiently computable (ideally at the hardware level). This is especially important when it comes to modern programmable switches that use high-speed packet processing pipelines based on match-action tables. To achieve efficient computation with comparators, OpenQueue imposes certain syntactic restrictions on their definition. The syntax below captures the main restrictions.

x			formal variables
n			numeric constants
e	::=	$n \mid x.f \mid \mathrm{e} \oplus \mathrm{e}$	arithmetic expressions
b	::=	$\mathrm{e} \oslash \mathrm{e} \mid b \owedge b$	predicates
c	::=	$comp\_name(x, x) = b$	comparator def.

Comparator declarations require providing a name $comp\_name$, and take two arguments. The type of the arguments varies depending on the priority being defined. For instance for queues in OpenQueue, which hold packets, a packet comparator has to be defined. The fourth production of the grammar contains predicates, which are the Boolean expressions b defining the comparison function. Aside from the standard arithmetic expressions (we generically denote by $\oplus$ the standard arithmetic operators), the first-order Boolean operators (denoted with the symbol $\owedge$) and arithmetic relations (denoted with $\oslash$), we allow the inspection of the fields of the arguments using the standard dot notation $x.f$, where f is assumed to be a field of the parameter x. It is assumed here that field access operations require a small constant number of memory accesses (generally one). Importantly, no function or procedure calls are allowed in comparators.

Boundary conditions: Data structures manipulated by OpenQueue can behave differently depending on whether the network entity is operating in *normal state*, or in *congested state*. For example, when a queue or a buffer becomes saturated, the user could specify that certain packets should be dropped to achieve graceful degradation. We allow the user to specify conditions under which a data structure should be considered congested. Again, we use a restricted language to express these *boundary conditions*. Unlike comparators, the predicates of boundary conditions are *unary* since they consider a single entity at any time.

$$
\begin{array}{lll}
pf & ::= \texttt{weightAdm} \mid \texttt{weightSched} & \text{modifiables} \\
ac & ::= \texttt{drop}(P) \mid \texttt{modify}(pf := \mathrm{e}) \mid \texttt{mark} \mid \texttt{notify} \mid ac \cdot ac & \text{actions} \\
cl & ::= (b,\ ac) \mid (b,\ ac) \cdot cl & \text{condition cases} \\
cd & ::= \mathit{cond\_name}(x) = cl & \text{declarations}
\end{array}
$$

This syntax shares the definitions of predicates and declarations with comparators seen before. Importantly, boundary conditions can be a sequence of cases (each case separated with a dot above). This is represented by the *cl* meta-variable representing a list of pairs, whose first component contains a predicate and second component defines an *action* represented by the meta-variable *ac*.[1] For the time being, we focus on the $\texttt{drop}(P)$ action which indicates that packets have to be dropped from the queue with probability P if the matching predicate evaluates to true. Actions $\texttt{modify}(pf := \mathrm{e})$, `mark`, and `notify` will be discussed later. Moreover, we allow actions to be sequenced, although generally only one action is used in conditions. Condition cases enable the expression of different response scenarios according to different types of congestion. For example, under severe congestion a more aggressive drop policy can be put in place by increasing the probability of dropping a packet. It is best if the conditions are mutually exclusive; in the current version of OpenQueue only actions of the first matching condition (in lexicographic order) will be triggered.

In the sequel we present the different entities comprising OpenQueue in detail. For each entity we provide its properties; some are primitives of the domain (e.g., packet size),

[1] ↑The syntax presented here is simplified for presentation purposes.

and others have to be set by the programmer. For each property we indicate in comments whether it is `r` read-only or `rw` writable, and `cons` if it's value is fixed during execution, or `dyn` otherwise. For functions we provide the return type (e.g., `bool fun`), and we denote comparators indicating their input types (e.g., `Packet comp.`), and boundary conditions indicating the actions that they allow (e.g., `drop cond`).

2.3.2 Abstract Interfaces

1) Queue: Fig. 2.4 shows the declaration of Queue interface. Standard property `size` is defined by the user at declaration time, as well as `buffer`, the buffer that contains the queue and is shared among several queues in the shared memory case. The `currSize` property serves to query current size and changes dynamically as the queue is updated. Abstractly, a queue contains packets ordered according to user-defined priorities for admission control and processing.

```
Queue {
 Queue(size)                                  // constructor
 size                // declared size in bytes [r, cons]
 currSize                 // current queue size [r, dyn]
 // admission policy
 admPrio(p1, p2)                               // [bool fun]
 congestion()        // [{drop,mark,notify,modify} cond]
 admState                   // admission state [rw, dyn]
 // processing policy
 procPrio(p1, p2)         // proc comparat.[Packet comp]
 schedState          // per-queue sched. state [rw, dyn]
}
```

Figure 2.4. OpenQueue Queue interface

Admission policy: The first policy concerns the admission of packets into the queue. `admPrio(p1, p2)` is a packet comparator used in case of congestion to choose the packets to be dropped from the queue. As an alternative, instead of defining an admission policy we could simply drop the least valuable packets according to `procPrio` priority that we will describe shortly. However, separate priorities for admission and processing not only

give more `EXPRESSIVITY` but also improve `PERFORMANCE` . There are several properties related to the admission policy.

(a) The user-defined `congestion()` boundary condition that shows when a queue is virtually *congested*, and defines which/how packets should be dropped (here we only consider the `drop(P)` action). The single argument of boundary condition declarations is implicitly instantiated to the queue being defined, hence this is a queue boundary condition. We notice here that the deterministic drop action corresponds to an action `drop(1)` in the syntax presented before. Usually, `congestion()` is a set of different buffer occupancy levels and corresponding drop probabilities [7]. In the example below we show a possible congestion policy whereby packets are dropped with a probability of 0.5 if the current occupation of the queue is greater than 3/4 of the total size of the queue but lower than 9/10; they are dropped with a probability of 0.9 if the occupation exceeds 9/10 but is lower than 19/20; and they are always dropped otherwise.

```
congestion() =
   (currSize >= 0.95 * size, drop(1)).
      (currSize >= 0.90 * size, drop(0.9)).
         (currSize >= 0.75 * size, drop(0.5))
```

OpenQueue can *push out* already admitted packets. To use the same implementation for push-out and non-push-out cases, an admission control policy could always admit incoming packets. In case of congestion, admission control randomly drops least valuable packets until congestion disappears.

(b) The optional function `postAdmAct()` is a boundary condition like `congestion()`, except that it is restricted to `mark`, `notify`, and `modify(`pf := e`)`. These actions are intended to tell subsequent processing entities that the packet is subject to special conditions.

(c) The function `postAdmAct()` can be used to implement *explicit congestion notifications* [19] or *backpressure*; `postAdmAct()` can return actions such as `mark` or `notify`. When bandwidth is allocated not only with respect to packet attributes, queues maintain a `weightAdm` variable that can be updated dynamically after each scheduling, and the is what `modify(`pf := e`)` is for.

Processing policy: The processing policy defines the priorities of packets in the queue through `procPrio(p1, p2)`; a packet comparator defined as a function taking two abstract packets and returning *true* if `p1` has a higher processing priority than `p2`. We are only concerned with the highest processing priority packet at any point. This priority defines the most and least valuable packets in the queue. Hence, the only way to access packets in the queue ordered by `procPrio` is through the `getHOL()` primitive which returns the HOL (i.e., packet with highest processing priority as defined by `procPrio`); e.g. the user can set

```
procPrio(p1, p2) = (p1.arrival < p2.arrival)
```

to encode FIFO processing so calls to `getHOL()` return the longest standing packet.

Scheduling policy: This policy allows to specify static bandwidth allocations among queues of the same port during scheduling. In this case, the policy is defined in part in queue declaration, and in part in port declarations. The `weightSched` variable for each queue is updated by the `postSchedAct()` function defined in ports (see below).

2) Buffer: A buffer is an optional entity, declared only when several queues share buffer space (see Fig. 2.5). It manages a set of queues assigned to it at creation; `congestion()`, `postAdmAct()`, `size`, and `currSize` are similar to the respective queue attributes. Under congestion, an admission control policy on the buffer level finds a queue that requires to drop a packet and the queue's admission control policy determines which packet to drop. To order queues for admission, user specifies the `queuePrio` comparator. e.g. to implement LQD, one can use.

```
queuePrio(q1, q2) = (q1.currSize < q2.currSize)
```

```
Buffer {
 Buffer(size, q1, ..., qk)                // constructor
 size                // declared size in bytes [r, cons]
 currSize                 // current queue size [r, dyn]
 currQueue                // current queue      [rw,dyn]
 // admission policy
 admPrio(q1, q2)                           // [bool fun]
 congestion()       // [{drop,mark,notify,modify} cond]
 admState        // optional per-buffer state [rw, dyn]
}
```

Figure 2.5. OpenQueue Buffer interface

3) Port: The interface for ports is presented in Fig. 2.6. A port manages a set of queues assigned to it at its declaration. `schedPrio(q1,q2)` is a user-defined scheduling property that defines which HoL packet is scheduled next (this packet is accessed through `getBestQueue()`). For example, priority based on packet values with several levels of strict priorities is defined as

```
Port {
 Port(rate, q1, ..., qk)                  // constructor
 rate           // declared rate in % or kbps [r, cons]
 // scheduling policy
 schedPrio(q1, q2)                         // [bool fun]
 schedState()             // scheduling state [rw, dyn]
 postSchedAct()          // [{mark,notify,modify} cond]
 currQueue                    // current queue [rw, dyn]
}
```

Figure 2.6. OpenQueue Port interface

```
schedPrio(q1, q2) = (q1.getHOL().value > q2.getHOL().value)
```

Finally, `postSchedAct()` is similar to `postAdmAct()` and is used to define new services/hooks.

4) Packet: The notion of a packet is *primitive*, meaning that the user cannot modify or extend packets; packet fields can be used to implement policies. To be independent of

traffic types and to have a clear separation from the classification module (that can be expressed in a different language), every incoming packet is prepended with three mandatory parameters, *arrival time*, *size* in bytes, and *destination queue*, and four optional parameters, intrinsic *value* (with application-specific meaning), *processing requirement* in virtual cycles, *slack* (maximal offset in time from *arrival* to transmission), and *flow* (a traffic aggregation that the packet belongs to). We assume these properties are set by an external *classification unit* (e.g.,OpenFlow [6], if a virtual switch is defined with the finest possible resolution), except for *arrival* (set by OpenQueue when a packet is received) and *size*.

```
Packet {
 // set by external packet classifier
 policyId          // used on policy changes   [r, cons]
 queue                      // target queue id [r, cons]
 value                      // virtual value   [rw, dyn]
 processing                 // no of cycles    [r,  dyn]
 slack                      // offset in time  [r, cons]
 flow                       // flow id         [r, cons]
 // set on arrival
 arrival                    // arrival time    [r, cons]
 size                       // size in bytes   [r, cons]
}
```

Figure 2.7. OpenQueue Packet interface

Fig. 2.7 depicts the `Packet` data structure. Intrinsic *value* and *processing* requirements are used to define prioritization levels [20]. *slack* is a time bound used in management decisions of latency-sensitive applications; e.g., if buffer occupancy already exceeds the *slack* value of an incoming packet, the packet can be dropped during admission even if there is available buffer space. We posit that all decisions of buffer management policies (admission, processing, or scheduling) are based only on specified packet parameters and internal state variables of a buffering architecture (e.g., buffer occupancy).

2.4 Using OpenQueue

In this section we demonstrate the use of OpenQueue using selected examples. We try to show how the schema and language of OpenQueue together help achieve its main goals, `EXPRESSIVITY`, `SIMPLICITY`, and `DYNAMISM`.

2.4.1 Example Policy Definitions

```
// priorities for admission and processing
fifo(p1, p2) = (p1.arrival < p2.arrival)
rfifo(p1, p2) = (p1.arrival > p2.arrival)
srpt(p1, p2) = (p1.processing < p2.processing)
rsrpt(p1, p2) = (p1.processing > p2.processing)
// congestion conds. considered.
// trigger when occupancy exceeds size.
tailDrop(q) = q.currSize >= q.size, drop(1)
```

Figure 2.8. Example priorities and congestion conditions.

```
// LQF: HOL packet from Longest-Queue-First
lqf(q1, q2) = (q1.currSize > q2.currSize);
// SQF: HOL packet from Shortest-Queue-First
sqf(q1, q2) = (q1.currSize < q2.currSize);
// MAXQF: HOL packet from queue that admits max processing
maxqf(q1, q2) = (q1.schedState > q2.schedState);
// MINQF: HOL packet from queue that admits min processing
minqf(q1, q2) = (q1.schedState < q2.schedState);
// CRR: Round-Robin with per cycle resolution
crr(q1, q2) = (q1.schedState < q2.schedState);
```

Figure 2.9. Example scheduling policies.

Fig. 2.8 shows how to express three commonly used priorities and congestion condition using OpenQueue. Fig. 2.9 shows how to express five commonly used scheduling policies using OpenQueue.

In Fig. 2.3(b) we saw an architecture for packets with heterogeneous processing requirements in which we allocate a dedicated queue for packets with the same processing requirement. The OpenQueue code in Fig. 2.10 creates this buffering architecture, with **k** separate

```
// create k queues each of size B
q1 = Queue(B); ...; qk = Queue(B);
out = Port(100%, q1, ..., qk);
// fifo admission order
q1.admPrio = rfifo(); ...; qk.admPrio = rfifo();
// fifo processing order
q1.procPrio = fifo(); ...; qk.procPrio = fifo();
// congestion condition
q1.congestion = tailDrop(q1); ...; qk.congestion = tailDrop(qk);
```

Figure 2.10. Multiple separated FIFO queues with a single output port architecture.

queues of size `B`. In this case, advanced processing and admission orders are not required as only packets with same processing requirements are admitted to the same queue. Here, the decision of which packet to process in order to maximize throughput is non-trivial since it is unclear which characteristic (i.e., buffer occupancy, required processing, or a combination) is most relevant for throughput optimization. Fig. 2.9 presents five different scheduling policies we can use in a case like this.

Fig. 2.11 shows how to define a shared buffer architecture. Here the size of the buffer is B and the size of each queue is also B.

```
// create n queues of size B
q1 = Queue(B); ...; qn = Queue(B);
// create a shared buffer of size B and attach queues to it
b = Buffer(B, q1, ..., qn);
// create an output port per queue
out1 = Port(q1); ...; outn = Port(qn);
```

Figure 2.11. Shared buffer architecture.

2.4.2 Configure a Switching Fabric

Here, we show a sample multi-level buffering architecture that demonstrates the applicability of OpenQueue to specify management policies in switching fabrics. Unlike an input-queued or combined-input-output-queued switch that requires synchronous policies that usually compute matching between input and output ports, adding an additional buffer-

ing level at crosspoints allows to make this buffering architecture asynchronous. Here, we consider a full-fledged version with three buffering levels, where the first level implements virtual-output queues (see Fig. 2.12 and Fig. 2.13).

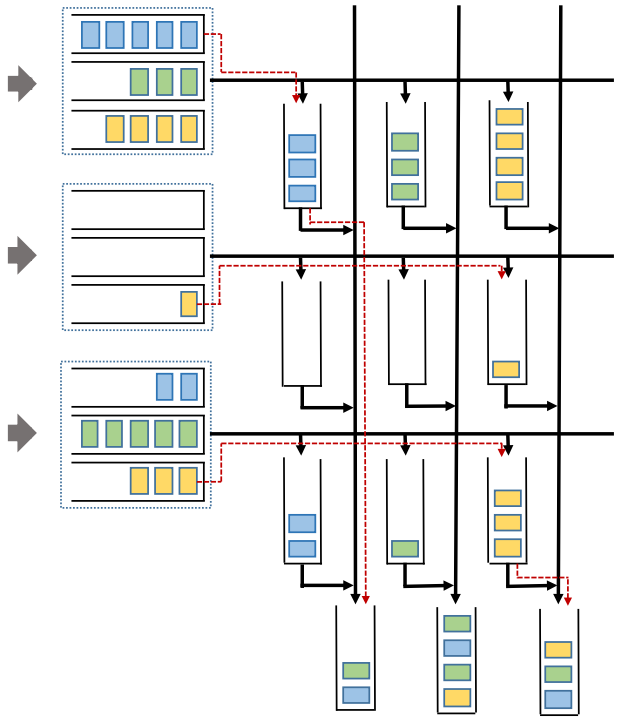


Figure 2.12. Buffered-crossbar switch with three hierarchical levels.

2.5 Feasibility of OpenQueue

A fundamental building block in OpenQueue is the *priority queue* data structure, where the order of elements is maintained based on a user-defined priority. An OpenQueue implementation can keep a single copy of a packet and use packet references to encode priorities (see Fig. 2.14). Therefore, the performance of OpenQueue on a given platform largely boils down to the efficiency of underlying priority queue implementation. While priority queue operations take $O(\log N)$ time in general, where N is a queue size, there are restricted versions (e.g., for predefined ranges of priorities) that support most operations in $O(1)$ and can be efficiently implemented even in hardware [21], [22], further increasing OpenQueue's appeal. To guarantee a constant number of insert/remove and lookup operations during admission

```
// create 9 virtual-output queues of size B
voq11 = Queue(B); ...; voq33 = Queue(B);
// attach voqs to input ports
in1 = Port(100%, voq11, voq12, voq13);
in2 = Port(100%, voq21, voq22, voq23);
in3 = Port(100%, voq31, voq32, voq33);
// create 9 crosspoint queues of size B
cq11 = Queue(B); ...; cq33 = Queue(B)
// crosspoints as ports
cp11 = Port(100%, cq11); ...; cp33 = Port(100%, cq33);
// create 3 output queues of size B
oq1 = Queue(B); oq3 = Queue(B);
// attach oqs to output ports
out1 = Port(100%, oq1); ...; out3 = Port(100%, oq3);
// setting queues:
// admission order to fifo
voq11.admPrio = fifo; ...; voq33.admPrio = fifo;
cq11.admPrio = fifo; ...; cq33.admPrio = fifo;
oq1.admPrio = fifo; ...; oq3.admPrio = fifo;
// processing order to fifo
voq11.proPrio = fifo; ...; voq33.proPrio = fifo;
cq11.proPrio = fifo; ...; cq33.proPrio = fifo;
oq1.proPrio = fifo; ...; oq3.proPrio = fifo;
// congestion condition
voq11.congestion = defCongestion(); ...;
cq11.congestion = defCongestion(); ...;
oq1.congestion = defCongestion(); ...;
// LQF: HOL packet from Longest-Queue-First
lqf(q1, q2) = (q1.currSize > q2.currSize);
in1.schedPrio = lqf; in2.schedPrio = lqf;
out1.schedPrio = lqf; out2.schedPrio = lqf;
```

Figure 2.13. OpenQueue configuration for the fabric in Fig. 2.12

or scheduling of a packet (i.e., to avoid rebuilding the priority queue), OpenQueue's user-defined expressions for priorities are immutable. The complexity of OpenQueue is hence reduced to translating user-defined settings to a target system that implements a virtual buffering architecture.

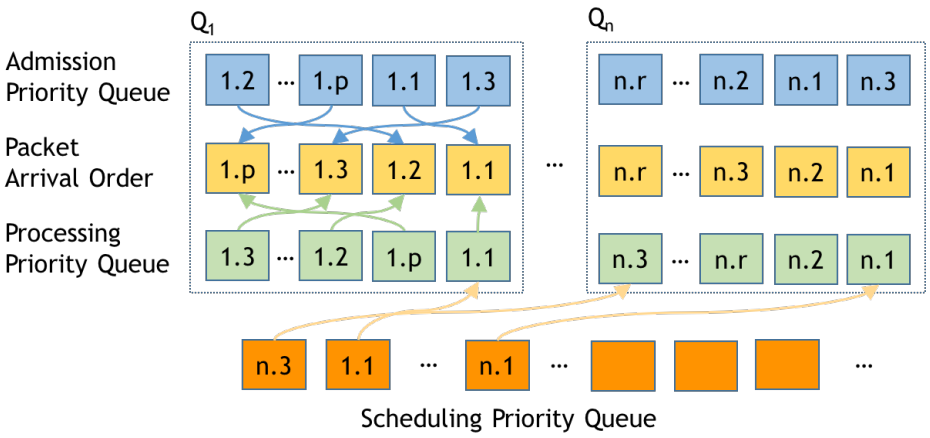


Figure 2.14. OpenQueue priority queues in Linux kernel.

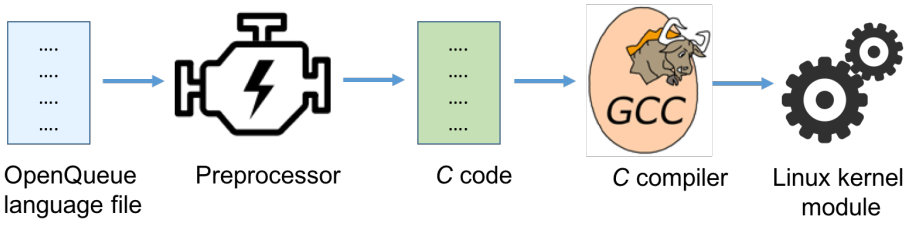


Figure 2.15. From OpenQueue language to Linux kernel module.

2.5.1 OpenQueue in the Linux Kernel

We have completed a proof-of-concept implementation of OpenQueue [23] in the Traffic Control (TC) layer of the Linux kernel. To that end, we have extended the `tc` Linux command to attach instances of our OpenQueue Queuing Discipline (as a qdisc[2]) to a network interface. Our qdisc is implemented as a Linux kernel module, which can be loaded into the kernel dynamically. A OpenQueue kernel module contains C language constructs correspond to OpenQueue policy elements. OpenQueue module name is given as a parameter to the `tc` command. For example, if the OpenQueue implementation has been compiled into a module named `my_openqueue` calling the command

```
tc qdisc add dev eth0 root my_openqueue
```

[2]↑qdisc is a part of Linux Traffic Control (TC) used to shape traffic of a network interface; qdisc uses `dequeue` to handle outgoing packets and `enqueue` to fetch incoming ones.

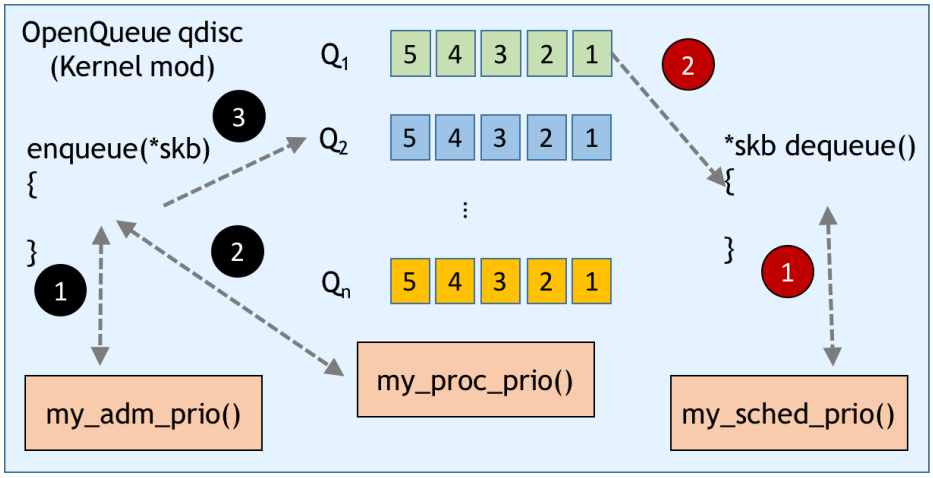


Figure 2.16. Use of function calls inside the kernel module during packet **enqueue** and **dequeue**.

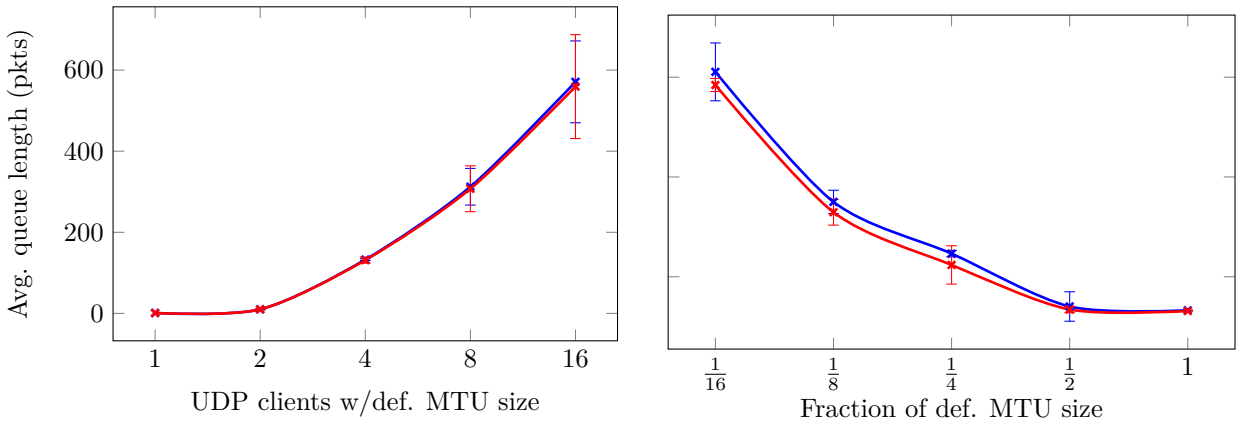


Figure 2.17. *Left:* average queue length as a function of number of clients generating UDP traffic with default MTU size. *Right:* fraction of default MTU size; blue: FIFO with prioritization; red: regular FIFO.

attaches our qdisc to `eth0` where the OpenQueue policies defined in the input OpenQueue file have been compiled into the module `my_openqueue`.

Fig. 2.15 illustrates how the loadable OpenQueue policy modules are generated. The input of our preprocessor is an OpenQueue file containing the desired architecture for the interface. Then, the preprocessor generates a corresponding C source code for that. Subsequently, we compile this file into a loadable kernel module to be loaded when using the `tc`

command dynamically (cf. the `insmod` command). This strategy allows to load new policies seamlessly, without disrupting currently executing policies on other interfaces.

The admission policy is evaluated inside the `enqueue` method. If a packet is admitted, the corresponding processing policy calculates its rank according to processing order. Similarly, the scheduling policy is evaluated inside the `dequeue` method to find the index of the queue that the next HOL packet is taken from. This interaction between the qdisc and predefined functions is depicted in Fig. 2.16. As a priority queue data structure we use B-Trees that keep pointers to packets (Fig. 2.14). The operational cost of packet insertions and deletions is $O(log\ N)$, where N is number of admitted packets.

2.5.2 OpenQueue Code Generation for Linux Kernel

Given the abstract nature of OpenQueue syntax and semantic, it is possible to generate high-level language code for any target runtime environment without hassle. We have a OpenQueue language parser and code generation toolset for Linux kernel. In our implementation OpenQueue policies are defined using syntax very similar to what we discussed in § 2.4. A sample policy file is given in Fig. 2.18. This policy file has three main sections. The *imports* section at the top specifies C header files that define function signatures. These functions include all possible C routines that can be used as function pointers for various queue/port operations that are defined dynamically. These functions are validated for their signature as each operation has its own specific format and also mundane sanity checks like argument types, duplicate function names, missing semicolons, etc. The next section defines queues and their attributes. The last section defines port and its attributes. These three sections collectively define a complete OpenQueue policy. Furthermore, queue/port operation attributes are not limited to precompiled C routines but also support a limited set of inline functions for improved usability.

The generated code can be compiled into a loadable Linux kernel module that can be attached to a network interface using Linux *tc* command. The implemented kernel module provides useful statistics about the qdisc runtime that can be queried using the *tc* command itself. List. 2.1 shows some statistics for the OpenQueue qdisc for the policy given in Fig. 2.18;

```
// Define OpenQueue policy for a port
// Import function definitions
import "include/routine/routines.h"

// Create queues
// Size 128
Queue q1 = Queue(128);
// Size 1024
Queue q2 = Queue(1024);

// Attributes of q1
q1.admPrio = my_adm_prio;
q1.congestion = my_congestion_condition;
q1.congAction = drop_tail;
q1.procPrio = my_pro_prio;

// Attributes of q2
// TOS field as admission priority
q2.admPrio = inline{Packet.TOS};
// Queue is congested if its length is 1024
q2.congestion = inline{Queue.length == 1024};
// Drop packets with 95% probability when the queue is congested
q2.congAction = drop_tail(0.95);
q2.procPrio = my_pro_prio;

// Create port
Port myPort = Port(q1, q2);

// Define port attributes
myPort.queueSelect = select_admission_queue;
myPort.schedPrio = my_schd_prio;
```

Figure 2.18. Sample policy file.

here 200 packets were sent through the qdisc where queue selector was set to pick a queue at random.

```
danushka@OpenQueue:~/OpenQueue$ tc qdisc show
qdisc openqueue 8002: dev eth0 root refcnt 2
Port: myPort
Queue: q1, Max: 128, Curr: 0, Dropped: 0, Total: 140
Queue: q2, Max: 1024, Curr: 0, Dropped: 0, Total: 60
```

2.5.3 Priority Queue and Performance

To explore the performance overhead introduced by several priority queues (implemented as B-trees) in OpenQueue, we used priorities based on arrival time to compare it with the base-line qdisc implementation that is implemented as a doubly linked list.

In our testbed we use a 3-node line topology to measure the performance overhead of our packet prioritization logic. The middle node runs Open vSwitch (OVS) with modified data plane (Linux kernel) and acts as a pass-through switch. We vary the number of parallel traffic generators on the first node and measure average queue length (i.e., number of packets in the default queue) on the third receiver node for two qdiscs: base-line FIFO and extended FIFO with prioritization in OpenQueue, reporting the average value of 50 runs with 95% confidence interval. Fig. 2.17(left) shows the average queue lengths for the two qdiscs; in both cases, average queue length increases with the number of UDP clients. In FIFO with 16 clients, the most congested case, regular FIFO has an average queue length 559.333 vs. 571 packets for FIFO with prioritization, which represents a mere 2% degradation. We also varied MTU sizes in the same 3-node line topology with 4 parallel UDP generators, which is enough to observe queue build-ups without dropping packets in the pass-though switch. We measured average queue lengths of the two qdiscs by varying MTU sizes from $\frac{1}{16}$ of the default MTU size to its default size (1500 bytes). Fig. 2.17(right) shows that for both qdiscs the average queue length decreases as MTU size increases; FIFO with prioritization incurs only 4% overhead: for MTU size of $\frac{1500}{16}$ bytes the result is 584.3 vs. 610.7 packets. This demonstrates that packet prioritization on top of FIFO incurs negligible performance overhead.

2.5.4 Evaluation of Operational Complexity in DPDK

We evaluate the operational complexity of OpenQueue using testbed experiments. We implement OpenQueue using DPDK [24] where the OpenQueue runtime is single-threaded and bound to a dedicated CPU core. During execution, the program polls traffic from a single port assigned to it at initialization. Ingress packets are written to an intermediate ring buffer based on forwarding decisions. Each port has an associated ring buffer, which acts as a thread-safe transmission channel between ingress and egress packet pipelines. The egressing packets are processed according to OpenQueue model where a packet is: (1) admitted based on the admission policy, (2) placed on a queue based on its processing policy, and (3) finally transmitted based on the scheduling policy. We use a setup on CloudLab [25] with 2 source nodes connected to a destination node via a switch node. Each node 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. The bandwidth of each interconnection is 10 Gbps. In this setup, the switch node is capable of working as a 10 GbE 3-port switch. We use DPDK version 18.11.2 on Ubuntu 18.04.1 LTS with *igb* as kernel driver. We use *iperf* v. 2.0.13 for our clients and servers. We have UDP clients on the source nodes sending traffic to servers on the destination node depending on the experiment as explained below. We set the load offered by each client (using -b command line option) high enough such that there is congestion, hence queue build up on the switch. On our switch, we record the number of packet drops and packet transmissions and derive average throughput attained by clients to assess the operational overheads incurred by OpenQueue. We use IPv4 Type of Service (ToS) field to mark traffic priority. We use the ToS values 63 and 1 for high-priority and low-priority traffic respectively. The two ToS values represent two distinct traffic priority levels for evaluation of operational complexity versus optimized objectives.

Experiment 1: We compare three algorithms: Algorithm A has FIFO admission and processing priorities and tail drop in case of congestion, Algorithm B also has FIFO admission and processing but drops low-priority packets on congestion, and Algorithm C has FIFO admission priority, SRPT processing priority w.r.t. ToS value, and drops low-priority packets

on congestion. Algorithm A is a native implementation without any priority queues (PQs), Algorithm B has one PQ, and (the optimal) Algorithm C has two PQs, for admission and processing. Results of this experiment are shown in Fig. 2.19a-b. Algorithm C spends the most operations to process a packet, Algorithm B is in the middle, and Algorithm A is the fastest. Thus, Algorithm C can process fewer packets than A and B and as a result, Algorithm A is best in terms of throughput (Fig. 2.19a). However, due to the smarter processing that more advanced algorithms provide the weighted throughput (total transmitted value) of Algorithms B and C outperforms Algorithm A; in terms of weighted throughput, $A < B < C$ (Fig. 2.19b). Here, weighted throughput is based on the traffic priority level (high vs. low) to show the advantage of specifically designed algorithms despite additional complexity.

Experiment 2: We assess the impact of using multiple queues attached to the same port. We use $k = 2, 4, 8$ queues equally sharing a total size of 512 with round-robin scheduling, comparing Algorithms A and C defined above; we measure weighted throughput as k changes. As a result (Fig. 2.19c-d), Algorithm A again outperforms Algorithm C in terms of throughput due to simpler processing but Algorithm C wins very convincingly in weighted throughput.

2.6 Related Work

Frenetic [26] and Pyretic [27], and Maple [28], among others, focus on service abstractions based on flexible *classifiers*, and do not address management of buffering architectures. Other approaches [29], [30] allow only for a *predefined set* of parameters for buffer management, which intrinsically limits expressivity. Another line of research abstracts representations of the southbound API (e.g., OpenFlow) in the data plane [31]–[33], while languages such as P4 [31] are very successful in representing packet classifiers, they are less suited to express buffer management policies. Our work was inspired by [34] that introduces a set of primitives to define admission control policies. Recently, Sivaraman *et al.* [22], [35] explored the expression of policies by one priority and one calendar queue, still leaving the language specification as future work. Mittal *et al.* describe an attempt to build a universal packet scheduling scheme [36]. In contrast to these approaches, OpenQueue considers the composi-

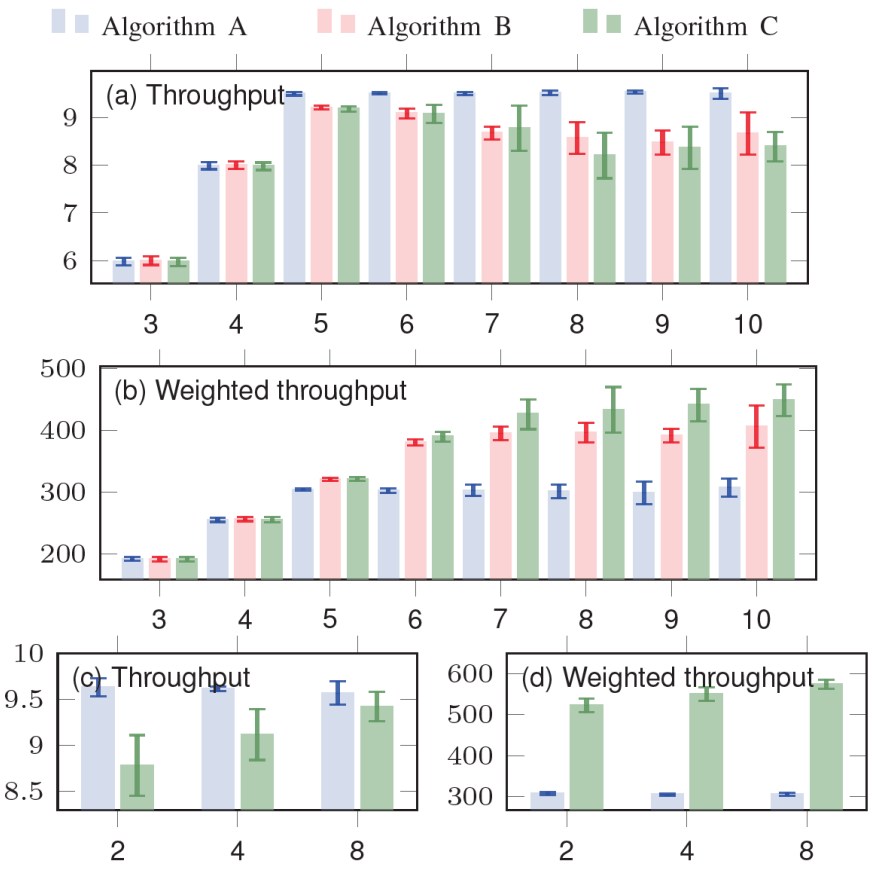


Figure 2.19. Average (weighted) throughput (shown on the Y-axis) as a function of input load in Gbps (X-axis) in Experiment 1 (a-b) and no. of queues in Experiment 2 (c-d).

tion of admission control, processing, and scheduling policies to optimize chosen objectives on user-defined buffering architectures. We formally show that while a single queue architecture can express management of multiple queues for every objective, a multiple queue architecture may be preferrable to improve operational complexity. We formally define the syntax of OpenQueue. New transports such as [2] require complex code changes on control and data planes of network elements to provision desired management policies. Once OpenQueue is supported on the network element, new policies can be added without code changes at runtime.

2.7 Chapter Summary

We propose a concise yet expressive language to define buffer management policies at runtime; provisioning new buffer management policies does not require control/data-plane code changes. We believe that OpenQueue can enable and accelerate innovation in the domain of buffering architectures and management, similar to programming abstractions that exploit OpenFlow for services with sophisticated classification modules. The conciseness of OpenQueue and ability to implement priority queue data structures at line-rate, make OpenQueue attractive for hardware implementations.

3. A CONTROL-THEORETIC APPROACH FOR CONGESTION CONTROL IN DATACENTER NETWORKS

In this chapter, 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× reduction in PFC frames generated under high 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* can be efficiently implemented in P4 and FPGA-based switch hardware.

3.1 Introduction

Congestion control in packet-switched networks has a clear goal: reduce FCTs 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 [37], [38], leverage latest developments in network hardware [39], and revisit previous work with a new perspective [40].

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 [41]–[45]. 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.

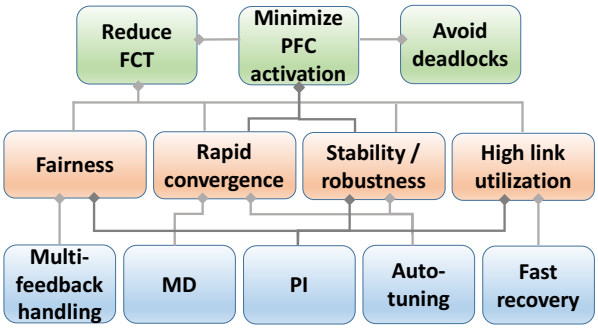


Figure 3.1. Relationship of the components (§ 3.3) of *RoCC* to its requirements (§ 3.2) and high-level goals (§ 3.1).

TCP becoming a bottleneck in datacenter networks [37] has made operators switch to transports based on 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) [46], which prevents packet drop by using back-pressure (at the traffic class level). Alas, PFC has been observed to cause problems such as HoL blocking, congestion spreading, and routing deadlocks [37]–[40], [47]. Less aggressive flow control mechanisms [48] 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 [39], and the many RDMA congestion control solutions developed over the recent past, e.g., [37]–[40], are indicators that congestion control for RDMA is a critical problem.

Fig. 3.1 summarizes high-level goals of congestion control we aim for and the technical requirements we pose to achieve these goals (detailed in § 3.2), and foreshadows the components of our solution and how these fulfill the requirements.

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 3.1 summarizes the most widely-known datacenter congestion control solutions. A very popular choice in production datacenter networks is DCQCN [37]. DCQCN is a source-driven congestion control approach for RoCEv2, which adapts the CP algorithm of 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 [49], [50]. TIMELY [38] 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 [49]. Several enhancements, e.g., DCQCN+PI [49], DCQCN+ [50], and patched TIMELY [49], have been proposed, but they (too) fail to meet important properties such as stability and fairness, which affect FCT.

The recently proposed HPCC [39] 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 (item *c)*) — commonly observed in modern datacenter networks [37]. 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 [37], network delay in [38], and INT in HPCC [39]) 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 [52] by leading switch manufacturers. Similarly, reservations that switch-driven congestion control can hinder line-rate packet

Table 3.1. Comparison of selected congestion control solutions (*solution-specific, CNP: congestion notification packet).

Solution	Switch action	Source action	Destination action
DCTCP [43]	Mark ECN	Adjust congestion window based on ECN	Echo ECN
QCN [51]	Compute and send F_b* to source	Compute rate based on F_b	None
DCQCN [37]	Mark ECN	Compute rate based on CNP	Send CNP to source
TIMELY [38]	None	Send RTT probes and compute rate based on RTT	Echo RTT probes
HPCC [39]	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

processing are countered by recent work [53]–[55] 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 proportional integral (PI) controller [56], (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. This paper extends our previous work [57] with emphasis on the hardware feasibility of the solution.

Our contributions can be summarized as follows: 1. After establishing important design requirements (§ 3.2), we present the design of *RoCC* (§ 3.3). 2. We evaluate *RoCC* via simulations and a DPDK implementation (§ 4.6) 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. 3. We explore the implementation feasibility of *RoCC* (§ 4.5) using P4 and FPGAs, two common technologies used to implement modern datacenter switching devices.

§ 3.6 summarizes related work, and § 3.7 summarizes the chapter.

3.2 Solution Requirements

We design *RoCC* to satisfy *four* key requirements for effective congestion control in RDMA datacenter networks (Fig. 3.1).

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 flow completion times (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 [39].

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, 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 § 3.3).

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 priority-based flow control (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.

3.3 *RoCC* Design

We now discuss the different components of *RoCC* and how they achieve our requirements and goals (Fig. 3.1). 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 [58] (proportional integral (PI) controller), QCN [51] (multi-bit feedback), PIE [59] (control parameter auto-tuning), and TCP (multiplicative decrease). Fig. 3.1 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).

3.3.1 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 3.2 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 § 3.3.2. F is the

Table 3.2. 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 § 3.3.3)
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*

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.

3.3.2 CP Algorithm

The CP periodically calculates the fair rate (FAIR) and sends it to certain sources using a special control message. Fig. 3.2 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.

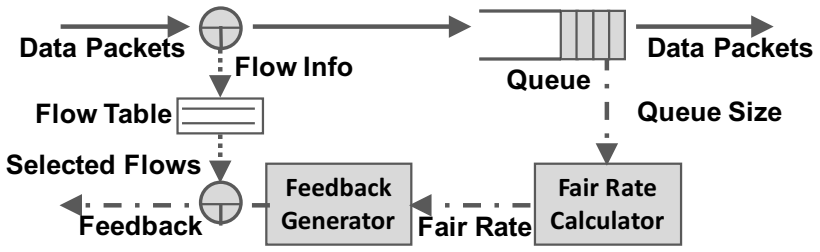


Figure 3.2. Overview of *RoCC* design at the CP.

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 § 3.3.3). 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., $> \frac{F_{\text{max}}}{8}$), the fair rate is set to F_{min} or $\frac{F}{2}$, respectively (Alg. 3 and Alg. 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 [58], PIE [59], and QCN [51]. 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 (Alg. 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 (Alg. 10) and lower bound (Alg. 12). After calculating the fair rate, Q_{old} is set to Q_{cur} (Alg. 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 (Alg. 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

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
```

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 **??**).

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

3.3.3 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 Internet control message protocol (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., explicit congestion notification (ECN) in datacenter QCN (DCQCN) [37], delay in TIMELY [38], and in-band network telemetry (INT) in high precision congestion control (HPCC) [39]).

3.3.4 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 [58], the first AQM mechanism that leveraged flow size and flow rate distributions to scale per-flow state.

4. ElephantTrap [60]: 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 [61]: 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

$$F = \frac{C_l - BW_{\text{mice}}}{N} \tag{3.1}$$

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, hence queue buildup.

3.3.5 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).

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

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 (Alg. 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 (Alg. 5). The algorithm also remembers CP_{rcvd} as most congested CP on the flow's path (Alg. 6).

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 (Alg. 8). *RoCC* stops fast recovery upon accepting a CNP (Alg. 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.

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()
```

3.3.6 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).

3.4 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 the CP algorithm on a custom application-specific integrated circuit (ASIC). We study P4, which is an emerging technology

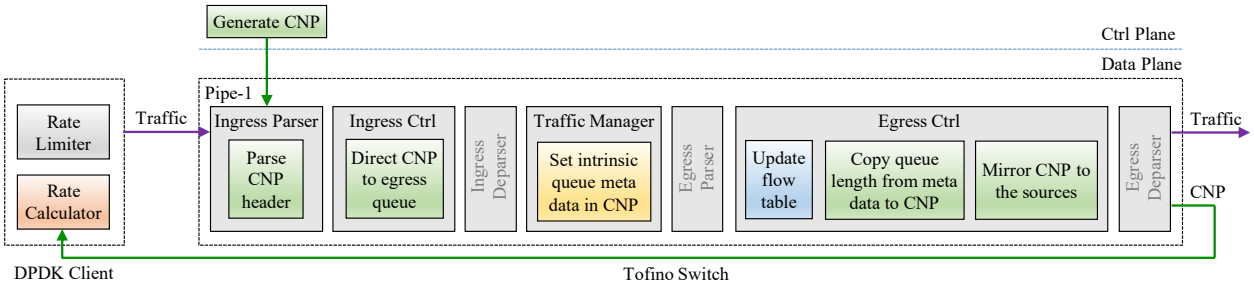


Figure 3.3. *RoCC* switch implementation in P4.

for implementing datacenter switches. Additionally, we study field-programmable gate array (FPGA) to assess the resource and latency demands of *RoCC* in switch hardware.

3.4.1 Basics

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.

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.

3.4.2 P4 Implementation

As data plane programmability becomes widespread, P4 [52] is becoming the de facto framework for programming switches, and major switch vendors already support P4.

Fig. 3.3 illustrates our *RoCC* implementation in P4. Our switch component (CP) is implemented in P4 using Intel Tofino that is based on Tofino native architecture (TNA). 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. Fig. 3.4 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 [55], 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 § 3.3.6) on CNP and "mirror" it to selected sources based on the flow table.

Our client (RP) is implemennted using data plane development kit (DPDK), and performs *two* main tasks: (I) intercept CNPs to compute fair rate, and (II) rate limit flows based on the computed rate. Pktgen [62] is an efficient traffic generation tool based on 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;
}
```

Figure 3.4. P4 header definition for CNP.

3.4.3 FPGA Implementation

We use Xilinx Vitis HLS 2020.2 [63] targeting Xilinx Virtex-7 XC7V2000T 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.

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.

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.

3.5 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 § 3.2 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 (§ 3.2).

For simulations, we use OMNeT++ [64] 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 [65], 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 [43]–[45].

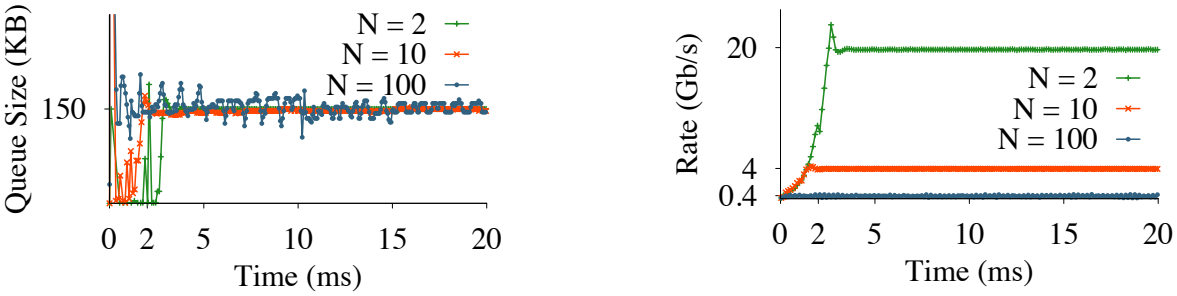
System parameters: All interconnections in our simulations are either 40 Gb/s or 100 Gb/s links. We chose PFC threshold values 500 KB and 800 KB for 40 Gb/s and 100 Gb/s links, respectively, based on [66]. NIC (RP) reaction delay for feedback messages is 15 μs. Δ^F is 10 Mb/s and Δ^Q is 600 B. T is set to 40 μs. F_{min} is 10, irrespective of link bandwidth. F_{max} is 4K and 10K 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. § 3.3.4 (1)).

3.5.1 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 remote direct memory access (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. 3.5 shows, the computed fair rate converges in $\sim$ 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. Results for $B = 100$ Gb/s are consistent with those for $B = 40$ Gb/s.



(a) Queue stability.

(b) Fair rate.

Figure 3.5. Fairness and stability of *RoCC* as load increases.

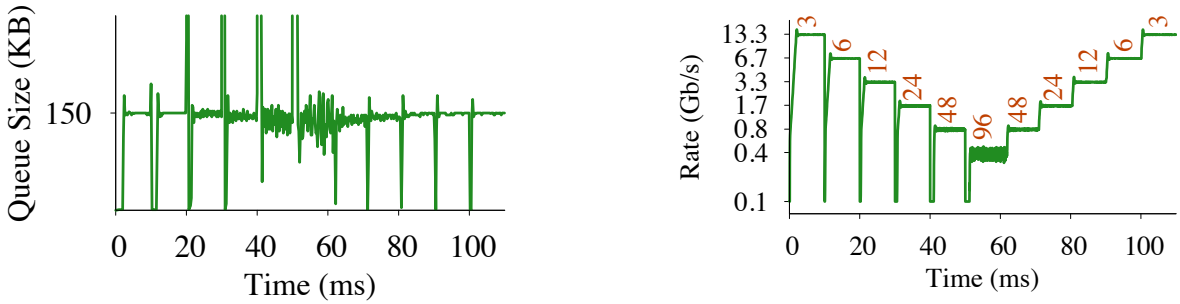
b) Convergence (CONV) and high link utilization (EFF) We exponentially increase (and reduce) the link load level by dynamically starting (and stopping) flows to investigate the system behavior when load fluctuates. We start with 3 flows (i.e., $N = 3$) and start new flows every 10 ms such that N doubles 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. 3.6 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 properties. We include QCN [51], DCQCN [37], DCQCN+PI [49], TIMELY [38], and HPCC [39] 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. 3.8a 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. 3.8b and Fig. 3.8c 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 $\sim$ 100 KB and $\sim$ 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 [49]). 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 case are consistent with its expected behavior. The stability of the rate attained by each flow closely follows that of link utilization.



(a) Queue stability.

(b) Fair rate.

Figure 3.6. Convergence of *RoCC*. The numbers in red are the flow counts during the intervals.

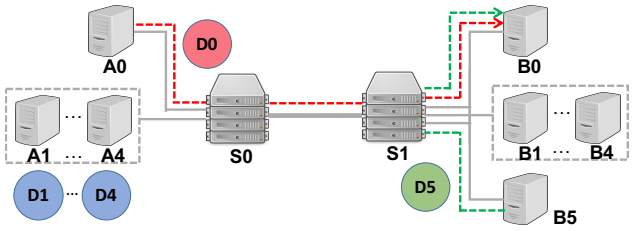
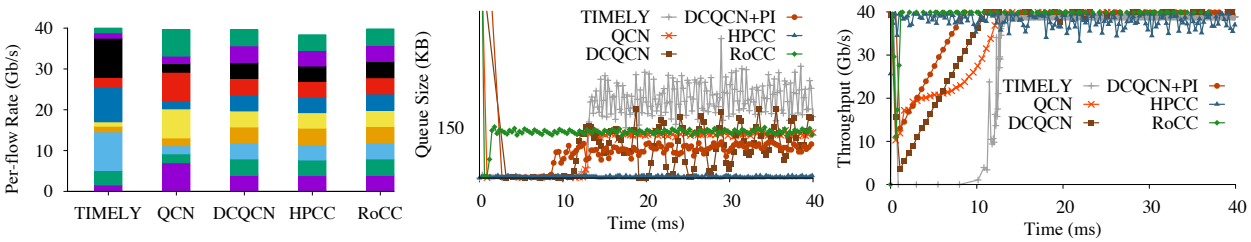


Figure 3.7. Multi-bottleneck topology.



(a) Fairness.

(b) Queue stability.

(c) Convergence.

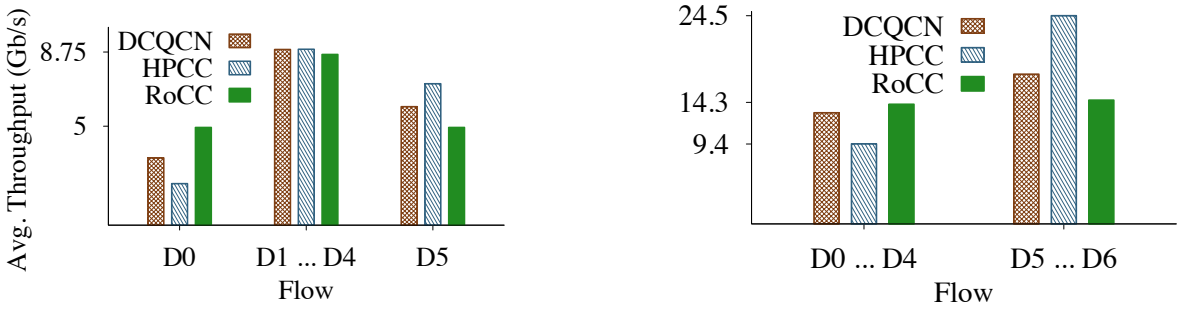
Figure 3.8. Comparing *RoCC* with TIMELY, QCN, DCQCN, and HPCC in terms of fairness, stability, and convergence.

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. 3.7, 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 [37] and HPCC [39] in terms of flow-level bandwidth allocation. As Fig. 3.9a 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).

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 [37] and HPCC [39] 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 × 5 and 100 Gb/s × 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.

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. 3.9b 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.



(a) Multi-bottleneck.

(b) Asymmetric topology.

Figure 3.9. Fairness of DCQCN, HPCC, and *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.

3.5.2 Evaluation with DPDK Implementation

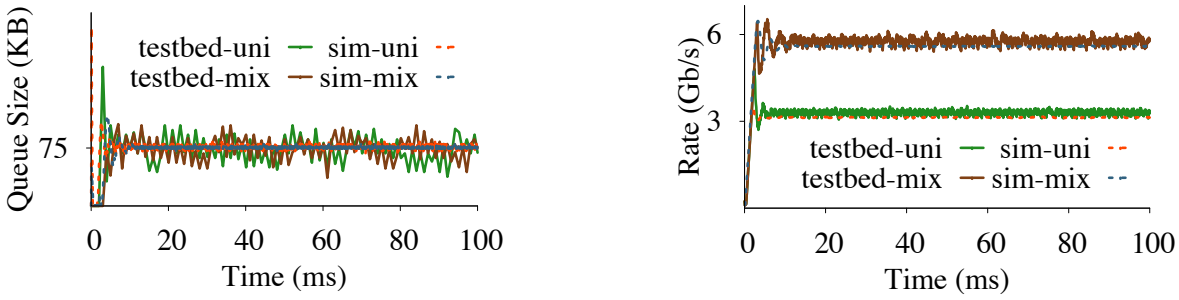
We implement *RoCC* using the popular DPDK [24] kernel bypass stack to validate our simulation results. The switch implementation uses three logical cores for handling data reception, packet switching, and data transmission and congestion control respectively. The source implementation uses two logical 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 [25] 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 [67] 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. 3.11a shows that the queue size stabilizes at 75 KB for both the testbed (testbed-uni) and simulation (sim-uni). In Fig. 3.11b, 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. 3.11a shows that Q_{cur} stabilizes at 75 KB for both the testbed (testbed-mix) and simulation (sim-mix) results. Fig. 3.11b shows that the flows attain the max-min fair value of 6 Gb/s in both testbed and simulation experiments.



(a) Queue stability.

(b) Fair rate.

Figure 3.10. Testbed results vs. simulation results.

3.5.3 Evaluation with P4 Implementation

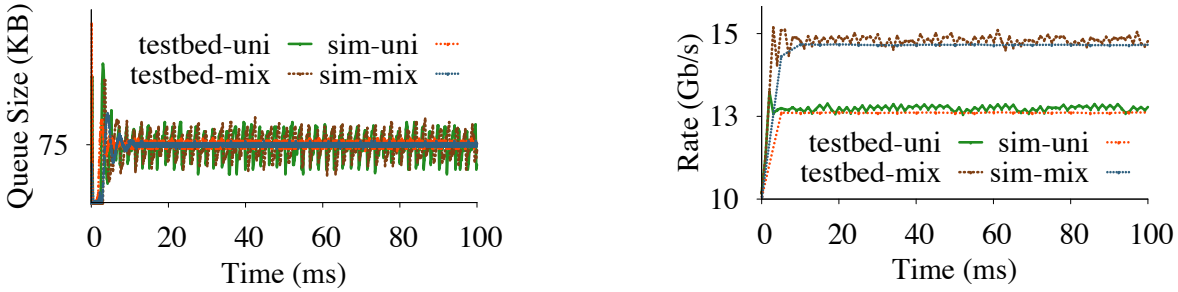
We deploy our P4 switch implementation on a Wedge 100BF-32X switch that has a programmable Intel Tofino ASIC. The switch is connected to a node that has a D1517 1.60GHz 8-core CPU and 8G RAM, using Chelsio T62100-SO-CR unified wire ethernet controllers. We run our traffic generators and receivers on 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 § 3.5.2).

In the first scenario, we configure each sender to transmit data at link speed (40 Gb/s). Fig. 3.11a shows that the queue size stabilizes at 75 KB for both the testbed (testbed-uni) and simulation (sim-uni). In Fig. 3.11b, 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. 3.11a shows that Q_{cur} stabilizes at 75 KB for both the testbed (testbed-mix) and simulation (sim-mix) results. Fig. 3.11b 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 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.



(a) Queue stability.

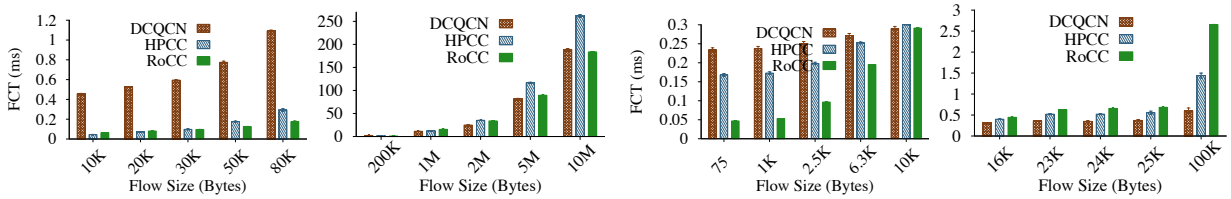
(b) Fair rate.

Figure 3.11. P4 results vs. simulation results.

3.5.4 Large-Scale Simulations

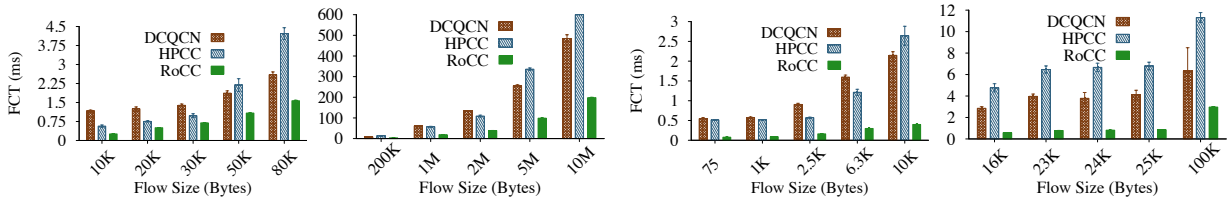
We use large-scale simulations to evaluate *RoCC* and compare it with DCQCN [37] and HPCC [39], in terms of (1) FCT and (2) PFC activation. We use a two-level fat-tree [68] topology with 3 core switches and 3 edge switches. Each edge switch is connected to each core switch using 2 100 Gb/s links (i.e., 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 [43], [44] (*WebSearch* traffic) and latency-sensitive small flows [44], [45] (*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) *WebSearch* traffic. (b) *FB_Hadoop* traffic.

Figure 3.12. Average FCT of DCQCN, HPCC, and *RoCC* (70% average load).



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

Figure 3.13. 99^{th} percentile FCT of DCQCN, HPCC, and *RoCC* (70% average load).

FCT: Fig. 3.12 and Fig. 3.13 respectively show the average 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 (bins) based on the flow size distributions of *WebSearch* traffic and *FB_Hadoop* traffic. Based on the 99^{th} percentile FCT, *RoCC* clearly outperforms 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 [39] 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 $\sim$ 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 mentioned above, and the bottleneck link utilization is close to link bandwidth. Table 3.3 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

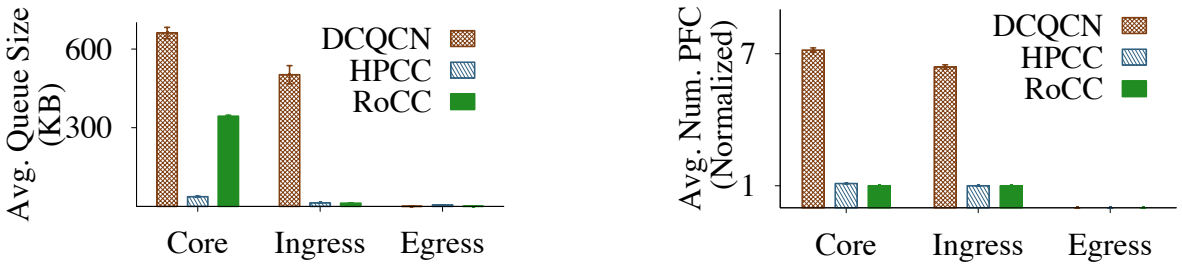
that flows of the same size exhibit low variance in their FCT (in other words, almost equal average, 90^{th} percentile, and 99^{th} percentile FCT).

Table 3.3. 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 $\sim$ 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

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. 3.14a 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. 3.14b 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. 3.14a), where DCQCN has deep queues, whereas HPCC has shallow queues. In contrast, *RoCC* maintains



(a) Average queue size.

(b) PFC activation.

Figure 3.14. Queue size and PFC activation of DCQCN, HPCC, and *RoCC* with *WebSearch* traffic (70% average load).

a stable queue at the CP. The PFC activation for *FB_Hadoop* traffic is consistent with that for *WebSearch* traffic.

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 without activating PFC, yet not dropping packets. We also examine the impact on FCT in this situation. Fig. 3.15 shows the FCTs of the solutions at 70% load. The FCT of DCQCN increases up to a factor of $\sim$ 13 and that of HPCC increases up to a factor of $\sim$ 7. In contrast, the FCTs of *RoCC* remains close to the FCTs when PFC is enabled with limited buffer (see Fig. 3.12). The average buffer usage of *RoCC* is around the reference value (300 KB), whereas DCQCN and HPCC on average use $\sim 80\times$ and $\sim 20\times$ 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 $\sim 4\times$ and $\sim 3\times$ lower than those of DCQCN and HPCC, respectively. For *FB_Hadoop* traffic, they are up to $\sim 7\times$ and $\sim 8\times$ lower than those of DCQCN and

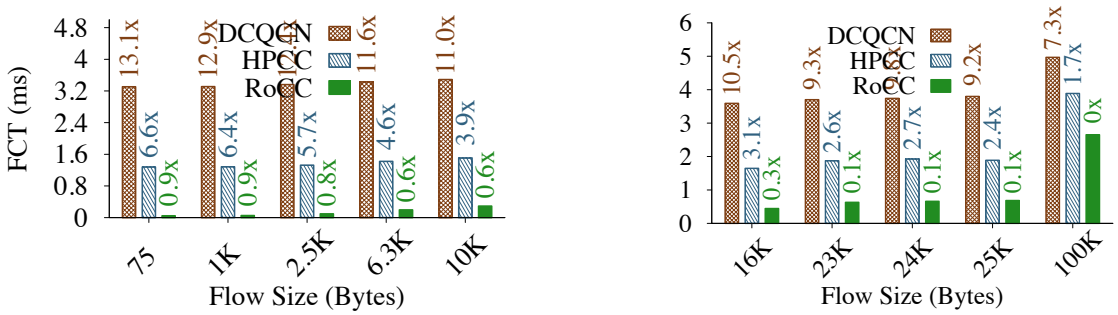


Figure 3.15. Average FCT of DCQCN, HPCC, and *RoCC* with PFC disabled and unlimited buffer (*FB_Hadoop* traffic at 70% average load). The numbers in respective colors show the fold increase in FCT, w.r.t. the case when PFC is enabled with limited buffer (Fig. 3.13).

HPCC, respectively., and (ii) *RoCC* is more stable (STBL) than DCQCN and HPCC, and as a result *RoCC* causes up to $\sim 7\times$ reduction in PFC activation, compared to DCQCN.

3.6 Related Work

Although congestion control research started as far back as the 1980s, several new proposals for the Internet (e.g., [69]–[73]) and datacenters (e.g., [37]–[40], [43], [50], [74], [75]) have emerged over the past few years. Table 3.1 and §3.1 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 [37] is one such widely deployed solution. TIMELY [38] is another production-grade solution that uses RTT as congestion signal. We have shown that they do not meet convergence and fairness goals. HPCC [39] 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 [51] 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 [76] 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 [77] 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 [78] 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.

3.7 Chapter Summary

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.

4. A SOLUTION TO PFC-INDUCED HEAD-OF-THE-LINE BLOCKING IN DATACENTER NETWORKS

In this chapter, we present Escape, a solution to head-of-the-line blocking in datacenter networks. Head-of-(the-)line (HoL) blocking in datacenter networks increases latency and causes congestion spreading, leading to routing deadlocks that can render parts or all of a network unusable. However, to the best of our knowledge, the relationship between HoL blocking and flow control in datacenter networks has not been investigated.

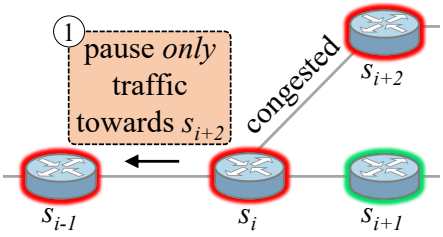
We dissect the conditions for datacenter HoL blocking to occur, and pinpoint a previously ignored cause: *in-order frame processing*. Based on this insight, we propose Escape, a protocol that allows HoL-blocked data frames to move forward *without rerouting* until they reach an unblocked port on their path. Simulations and evaluation with a DPDK-based prototype demonstrate that by clearing HoL blocking, Escape prevents routing deadlocks and reduces overall flow completion times. Escape reduces flow completion times of short flows passing through a single bottleneck by ∼20%. We also show how Escape can be efficiently implemented in hardware.

4.1 Introduction

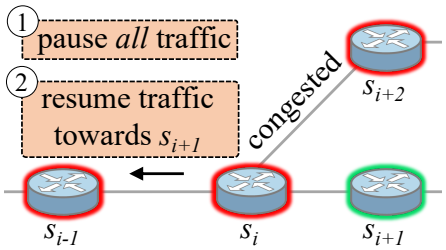
Head-of-(the-)line (HoL) blocking degrades network performance. When a frame at the front of a switch queue (head) stalls due to insufficient space in the downstream queue, it *blocks* the upstream traffic (line). The complexity of datacenter network traffic [79], [80] and its interaction with flow control create a type of HoL blocking, which we refer to as PFC-induced HoL blocking, where PFC [81] is priority-based flow control (we use "HoL blocking" to refer to PFC-induced HoL blocking in the remainder of this paper). Switching fabrics employ frame queuing at the ingress (*input-queued*), egress (*output-queued*), or both, and datacenter switch fabrics commonly use output-queued architectures, hence datacenter networks mostly suffer from HoL blocking at the egress. HoL blocking increases FCT and creates routing deadlocks [47], [48], [82]–[85].

Cause of datacenter HoL blocking. The coarse-grained XOFF/XON operation of PFC flow control is the main cause of datacenter HoL blocking. At the same time, this flow control protocol plays an important role in quickly reacting to potential buffer overrun and reducing frame losses which increase FCT. Specifically, modern datacenter networks commonly rely on kernel bypass transports based on RDMA (e.g., RoCEv2 [86]), which are loss-sensitive. Such transports require PFC to substantially reduce frame loss due to congestion.

Avoiding HoL blocking. It is possible that congestion is localized and traffic experiences little or no congestion on switches downstream of a congested switch. Therefore, effective flow control should consider *downstream congestion status*, not just *local congestion status*. Fig. 4.1a illustrates ideal flow control that *only* pauses the traffic that creates congestion. This solution is difficult to realize in practice due to implementation overhead, i.e., limited resources on switch ASICs for maintaining current congestion information for *all* data flows. We observe that we can approximate this solution by pausing all traffic, and then selectively allowing certain traffic to make progress based on the downstream congestion status. This solution is depicted in Fig. 4.1b.



(a) *Only* pause traffic that creates congestion (Ideal flow control).



(b) Pause *all* traffic (PFC), then *only* resume traffic that does not create congestion (Escape).

Figure 4.1. Flow control without HoL blocking.

State of the art. To the best of our knowledge, no existing work specifically addresses the relation between flow control and HoL blocking. While a main goal of congestion control is to eliminate PFC activation, state-of-the-art congestion control solutions [87]–[89] only *reduce* activations, primarily due to reaction delays. Routing deadlock solutions [83], [85] also mitigate HoL blocking to some extent. Such solutions are either *proactive* or *reactive*. Most proactive solutions use deadlock-free routing to eliminate cyclic buffer dependencies,

which over-constrains routing since a deadlock will not occur as long as congestion is not severe enough to trigger PFC. Reactive solutions detect deadlocks and break them, e.g., by dropping frames or adapting routing. Such solutions are typically inefficient (e.g., requiring consensus to select a node to react, possibly inducing livelocks). Recent work [48] replaces the PFC XOFF/XON mode of operation, but introduces overhead. An alternative approach is to prioritize traffic based on its class [90], [91]. The number of traffic classes is limited, however, and flow-level HoL blocking is still possible.

Solution overview. When a switch queue experiences HoL blocking on a frame destined to a PFC-activated egress port, the switch may have ports that are *not* PFC-activated, hence open to subsequent frames in the *blocked* upstream queues. Our solution, Escape, allows these blocked frames to leave the queues and reach a downstream switch with open egress ports. Escape uses dedicated queues for escaping frames to facilitate their movement. A key benefit of Escape is that it *only* activates when there is HoL blocking, and does not interfere with network operation under normal conditions (e.g., no frame tagging or control message passing required in the absence of HoL blocking).

Contributions and roadmap. We summarize work related to HoL blocking (§ 4.7), and present our design goals and key ideas (§ 4.2). We present Escape, a solution to HoL blocking in datacenter networks (§ 4.3). We argue that Escape preserves the properties of HoL blocking and deadlock clearance, no frame drop, and in-order frame delivery (§ 4.4), and describe its implementation in FPGA and DPDK (§ 4.5). Simulations and DPDK-based experiments quantitatively show that Escape clears routing deadlocks, increases link utilization (∼90%), and reduces average FCT (§ 4.6). For instance, Escape reduces the average FCT of “innocent flows" (not contributing to congestion) traversing a single bottleneck by ∼20%.

4.2 Design Rationale

We present the goals and insights that guided our design.

4.2.1 Design Goals

To our knowledge, no existing solution (cf. § 4.7) simultaneously meets the following goals:

1. HoL block clearance (CLR): HoL blocking should be cleared by eliminating a *necessary* condition (§ 4.2.2).

2. Zero frame drop (DRP): While frame dropping is a simple way to clear HoL blocking, ultra-low latency demands in datacenter networks make it undesirable, since frame drop implies retransmission and possible livelocks [92]. Dropping frames also defeats the purpose of using PFC.

3. In-order frame delivery (ORD): Rerouting blocked frames has also been proposed. This can, however, cause out-of-order frame delivery.

4. Efficiency (EFC): Normal network operation should not be affected, and the solution should be efficient.

5. Ease of configuration (CFG): Solutions with complex parameters are difficult to configure and are less robust.

4.2.2 Dissecting HoL Blocking

The fundamental cause of HoL blocking is *resource contention*, where buffer space on a routing node is the resource being exclusively used. Fig. 4.2 Ⓐ shows the set of *necessary* conditions for HoL blocking, where eliminating *any* of these clears the blocking. Three of the conditions are common to any resource contention problem and have been studied in system deadlocks [93]. 1. *Mutual exclusion*: No two frames can share the same buffer space. 2. *Hold and wait*: A frame requires sufficient buffer space on a switch (intra-switch) or on the downstream switch (inter-switch). 3. *No preemption*: A frame cannot relinquish its buffer space for another frame lest it be lost. If a frame moves to another buffer to avoid loss (e.g., structured buffers [47]), out-of-order delivery may occur [92]. *In-order frame processing*, a specific *fourth* condition together with the three conditions above, causes HoL blocking.

HoL blocking is *directional*, i.e., the data frame that causes HoL blocking always has a downstream target queue. When a sequence of HoL blocks satisfies a *fifth* condition, *circular wait*, it forms a routing deadlock (cf. Fig. 4.2 Ⓑ).

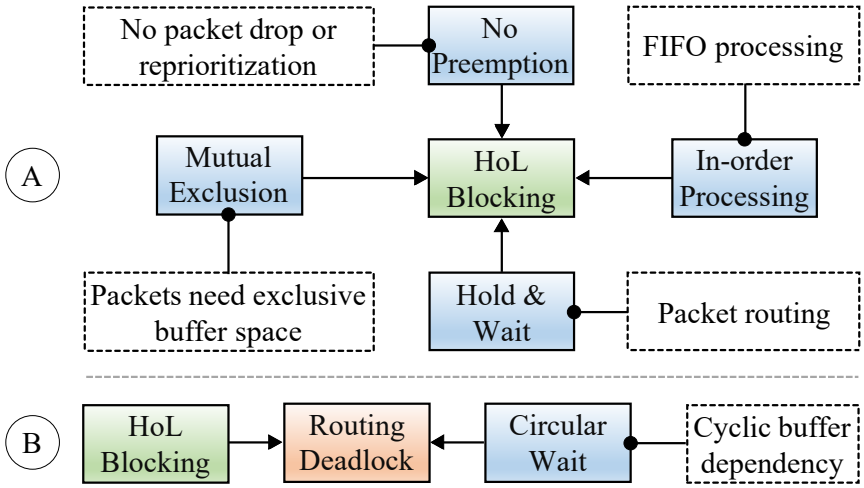


Figure 4.2. Ⓐ the *four* necessary conditions (blue boxes) and corresponding network requirements (dashed boxes) for HoL blocking that leads to Ⓑ routing deadlock formation when a *fifth* condition is met.

4.2.3 Key Insight

Escape is based on the simple idea that *HoL blocking can be cleared by allowing blocked frames to leave the queue and move forward, if they have open (i.e., not PFC-activated) switch egress ports downstream.*

Fig. 4.3 shows a typical HoL blocking scenario in a datacenter network where h, s, p, and f denote a network host, switch, port on the switch, and traffic flow, respectively. f_1, flowing from h_1 to h_3, has a high bandwidth demand and creates persistent congestion along its path, causing PFC activation. f_2, flowing from h_2 to h_4, has low bandwidth demand and creates little or no congestion; nevertheless it is HoL blocked at s_1. In order to unblock f_2, s_1 should be made aware that f_2 is not blocked downstream (p_3 on s_2 is open), hence s_1 can extract the frames of f_2 from the blocked queue, and let them *escape*, clearing HoL blocking.

This highlights *two* key observations about HoL blocking in datacenter networks that guide our solution.

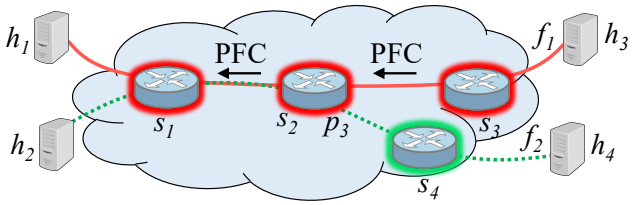


Figure 4.3. Frames of a HoL-blocked flow can be pushed out of the queue and allow them move forward, if the flow is not blocked downstream.

1. When a switch has one or more PFC-activated ports causing HoL blocking, it can have other ports that are not PFC-activated, hence *open* to some of the HoL-blocked traffic upstream.

2. By propagating upstream the information about the flows that traverse these open ports, the upstream switches can *extract* the blocked frames of the flows, and let them flow downstream, clearing HoL blocking.

4.3 Escape

Fig. 4.4 depicts how Escape clears HoL blocking. On detecting HoL blocking on switch s_a, Escape ① sends upstream a token τ carrying the identifier of a blocked flow that traverses an open port p_2 on s_a. Upon receiving τ at switch s_b, ② Escape extracts the first (blocked) frame d of that flow from queue q_2, tags the frame, and ③ lets it move forward through *dedicated queues*, unblocking the flow.

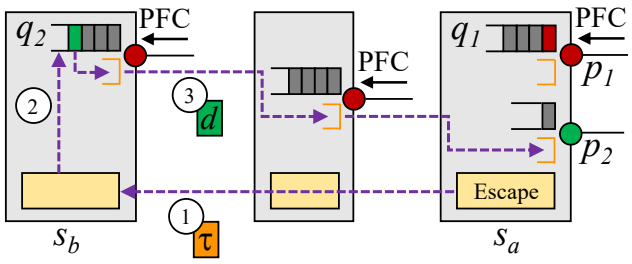


Figure 4.4. Escape overview: A flow that is HoL-blocked at switch s_b, is cleared as frame d of the flow in queue q_2, *escapes*.

4.3.1 Overview

Escape comprises the following components:

1. *Token manager* performs two key functions: (a) Emit tokens for potential HoL-blocked flows. A token can unblock *one* frame of a flow. (b) Intercept tokens received from a downstream switch to unblock frames matching the flow identifier in the token. If the token manager cannot find a matching frame, it forwards the token to the immediate upstream switch on the path of the flow.

2. *Token pool* is a fixed number of tokens per switch egress port. This number is a parameter that is equal to the capacity of $\vec{q}$ (see 5 below) of the egress port. Minimum pool capacity is *one*. Increasing the token pool capacity increases the number of frames that can simultaneously escape. [EFC]

3. *Escape token frame (ETF)* is a control frame used by Escape to disseminate tokens. ETF includes a flow identifier (fid), and the number of switches ($hops$) it has traversed from its origin.

4. *Escape frame header (hops)* flags unblocked frames. Identifying these frames is important for (a) reclaiming the token pool entries at the switches, as unblocked frames traverse them, and (b) queuing them (see § 4.3.3.2). In RoCEv2, the most common transport used in datacenter networks, the Escape header fits between the UDP header and Base Transport Header.

5. *Escape queue ($\vec{q}$)* is a per-port queue dedicated to unblocked frames. Modern switches have up to 8 queues per port, which may not all be used. [DRP] [ORD]

6. *Port registry* tracks if a port currently requested (ingress) or has activated (egress) PFC. Additionally, it maintains the current token pool size of each egress port.

7. *Flow table* includes ingress ports of flows traversing an egress port. A flow in this table is uniquely identified using the flow's 5-tuple[1]. Escape uses the flow table to identify flows

[1]↑(i) source IP, (ii) destination IP, (iii) source port, (iv) destination port, and (v) protocol.

that can be unblocked. The table does not include *all* flows traversing a port; Escape's default flow table (§ 4.5.2) maintains the last N flows that traversed the port where N is its token pool size.

8. *Queue supporting "extraction"* enables finding frames matching a given flow identifier, and extracting them.

Table 4.1. Algorithm symbol definitions.

Symbol	Definition
f	Active data flow
d	Data frame of f
F	Currently active data flows on switch
τ	Escape token frame (ETF)
p^I	Ingress port
p^E	Egress port
$T(p^E)$	Current token pool size of p^E
P	All ports on switch
Δ^{esc}	Period at which main Escape task should run

4.3.2 Algorithm

Table 4.1 lists the symbols used in the algorithm. There are *three key events*, shown in Alg. 3, that affect the algorithm: e_1, a timer task that triggers frame escape operations (TRIGGERESCAPE, line 1); e_2, an egress port receiving a token (RECEIVE(τ), line 15); e_3, an egress port transmitting a data frame (TRANSMIT(d), line 29). The algorithm uses the PFC meta information in the port registry (§ 4.3.1.6).

We now explain the algorithm in detail.

Triggering Escape (e_1). Routine TRIGGERESCAPE (line 1) executes on a switch every Δ^{esc}, which is a system parameter. This parameter does not affect the qualitative algorithm properties. It is set to ensure a subsequent HoL clear operation on the same port is attempted roughly after a message on the immediate upstream switch has escaped, to minimize unnecessary HoL clear attempts. A good choice for Δ^{esc} is 2 × link delay. Escape

Algorithm 3 Escape algorithm

```
1: task triggerEscape every Δ^esc do                                   ▷ event e1
2:     P^I ← {p^I | p^I ∈ P ∧ p^I is PFC-requested}
3:     for each p^I ∈ P^I do
4:         clearHoLBlock(p^I)
5: procedure clearHoLBlock(p^I)
6:     for each f ∈ F do
7:         p^I' ← ingress port of f
8:         p^E ← egress port of f
9:         if p^I' = p^I ∧ p^E not PFC-activated ∧ T(p^E) ≠ 0 then
10:            τ ← create new token
11:            τ.fid ← id of f
12:            τ.hops ← 1
13:            send τ from p^I to upstream switch
14:            T(p^E) ← T(p^E) − 1
15: upon receive(τ) on p^E do                                          ▷ event e2
16:    if T(p^E) = 0 then
17:        return
18:    f ← flow corresponds to τ.fid
19:    d ← first frame of f in p^E's queue
20:    if d exists then
21:        d.hops ← τ.hops                                             ▷ set Escape header
22:        transmit d through p^E
23:        return
24:    p^I ← ingress port of f
25:    if p^I is PFC-requested then
26:        τ.hops ← τ.hops + 1
27:        forward τ from p^I to upstream switch
28:        T(p^E) ← T(p^E) − 1
29: upon transmit(d) on p^E do                                         ▷ event e3
30:    if d.hops ⩾ 1 then                                              ▷ escaping frame
31:        T(p^E) ← T(p^E) + 1
32:        d.hops ← d.hops − 1
```

finds all the ingress ports that have currently requested PFC (line 2), and attempts to clear potential HoL blocking caused by each of those ports using the clearHoLBlock(p^I) routine (line 5). The routine traverses the flow table (line 6) and attempts to unblock a flow f, if (i) it is potentially blocked at p^I, (ii) it is destined to an open egress port p^E, and (iii) the token pool of p^E is not exhausted (line 9). To trigger the unblock operation for f, Escape

first creates a token τ (line 10), sets its fid to f's identifier (line 11), and *hops* to 1 (line 12). Escape sends τ through p^I (line 13), and decrements p^E's token pool size (line 14).

Receiving token (e_2). Routine RECEIVE(τ) (line 15) handles the receipt of a token τ at p^E. If the token pool of p^E is exhausted, Escape ignores τ (line 16). If not, Escape sequentially scans p^E's queue to find a data frame belonging to flow f, specified in τ (line 19). If there is a matching data frame, it is extracted from the queue, flagged as an escaping frame (line 21) and transmitted on p^E, terminating the unblock operation (line 22). If Escape fails to find a frame of f in the queue, it extends the frame search onto the immediate upstream switch through f's ingress port p^I, if it has currently requested PFC (line 25). To extend the frame search, τ is forwarded through p^I (line 27) after incrementing *hops* of τ (line 26), and decrementing p^E's token pool size (line 28).

Transmitting data frame (e_3). When an escaping frame d (line 30) is transmitted from an egress port p^E, Escape reclaims the corresponding token of p^E by incrementing its token pool size (line 31). Additionally, Escape decrements $d.hops$ (line 32). As d traverses switch ports that are HoL-blocked, $d.hops > 1$, and when it reaches the switch from which the corresponding token originated, $d.hops = 1$. This is consistent with the fact that the value of *hops* of a token at its origin is 1 (line 12). Beyond this point, $d.hops = 0$, and d is not considered an escaping frame.

4.3.3 Design Details

1. *Token-based buffer allocation* [DRP]: Escape reserves space in $\vec{q}$ (§ 4.3.1.5) of an egress port p^E for *every* escaping frame d, using *three* simple rules: (i) drop a token at p^E if its token pool is exhausted so $\vec{q}$ is fully *reserved* by previous tokens; (ii) decrement the token pool size of p^E, reserving a spot in $\vec{q}$, as a *token* traverses p^E so d is guaranteed to have space in $\vec{q}$; (iii) increment the token pool size of p^E as d leaves it to reclaim space in $\vec{q}$.

2. *Queuing* [DRP] [ORD]: Escaping frames are queued on the switch based on their *hops* header (§ 4.3.1.4). Frame d is classified to use $\vec{q}$ of an egress port, if *hops* of d is *not zero*. d uses the *dedicated queue* (bypassing the blocked queues) on each switch as d

reaches the switch from which the token that caused d to escape originated. Escape does not classify frames, and the rules for classifying escaping frames are added as part of deploying Escape on a switch.

3. *Scheduling* [ORD]: Frames only escape while regular traffic is paused. Prioritizing escaping frames prevents out-of-order delivery in case paused traffic is resumed when an escaping frame reaches a previously blocked switch.

4. *Selecting flows* [EFC]: Escape's main goal is to clear HoL blocking with minimal overhead, hence by default, it sequentially scans the flow table (line 6) for selecting flows. Escape can use other flow selection schemes (e.g., weight-based) to meet quality of service demands.

5. *Reclaiming unused tokens:* The flow table includes active flows and a token is reclaimed as an escaping frame traverses the egress port (line 28). However, if a flow in the table has just finished, the tokens generated for that flow may not be reclaimed. These tokens can be reclaimed using a timer event with a large interval (e.g., maximum delay across the network) by saving the token pool size in each interval, and incrementing it during the next interval if it has not changed.

6. *Token message overhead* [EFC]: The token message overhead depends on *two* factors: (i) Average convergence time of congestion control – higher convergence times imply higher token message overhead. (ii) Difference between PFC XOFF and XON thresholds – when it is large, many frames need to escape until the queue reaches its XON threshold. Congestion control typically has bounded convergence times and PFC thresholds are set according to strict guidelines [66], [94] enforcing minimal gaps between the two thresholds. Thus, tokens incur bounded overhead.

4.4 Verification

In this section, we formalize the properties that ensure Escape's correctness and argue that Escape preserves them.

4.4.1 Properties

Definition 4.4.1 (HoL block freedom)**.** *Network traffic is free from HoL blocking* $\iff$ *the following holds for any two distinct flows* f_i *and* f_j *traversing any node* n *where at time* t *the frames of* f_i block *those of* f_j *in a queue on* n*:* $\lim_{t\to\infty} C_\mathrm{i} = 0$ *and* $\lim_{t\to\infty} C_\mathrm{j} > 0$ *where* C_i *and* C_j *are the throughputs of* f_i *and* f_j *through* n*, respectively.*

Definition 4.4.2 (Losslessness)**.** *Frame delivery is lossless* $\iff$ *the following holds for any set of data frames* D^t *transmitted by the source and the set of data frames* D^r *received by the destination:* $D^t = D^r$*.*

Definition 4.4.3 (In-order frame delivery)**.** *Frame delivery is in-order* $\iff$ *the following holds for any sequence of data frames* D^t *transmitted by the source and the sequence of data frames* D^r *received by the destination where* D^t*:* $D^r \sqsubseteq D^t$ *(*D^r *is a subsequence of* D^t*).*

4.4.2 HoL Block Clearance [clr]

Theorem 4.4.1. *Escape clears HoL blocking.*

Proof. Let f be a data flow traversing a sequence of nodes $N = [n_1, \ldots, n_m]$. f is HoL-blocked at time t at node $n_k \in N$. According to 4.4.1, if f unblocks at n_k, throughput C attained by f at n_k should be s.t. $\lim_{t\to\infty} C > 0$. Consider the two key steps in Escape for unblocking f:

1. *Reserve buffer:* As Escape on n_k selects f as a flow that can unblock (line 9), it emits tokens for f (line 13). As token τ moves up the path of f and reaches $\forall\ n_u \in N$ where $1 \leqslant u < k$, Escape has reserved space for a frame of f, in the dedicated queue on every node from n_u to n_k (line 28). Therefore, a frame has an unblocked path from n_u to n_k.

2. *Extract frame:* Upon receiving τ, Escape on n_u extracts d_i of f (line 19), and transmits it (line 22). d_i is flagged (line 21), enabling it to use the unblocked path to n_k.

Steps 1 and 2 together guarantee that the blocked frames of f can leave the queues and move downstream through an unblocked path, and as a result, throughput C attained by f at n_k is s.t. $\lim_{t\to\infty} C > 0$. Thus Escape clears HoL blocking.

□

4.4.3 Zero Frame Drop [drp]

Theorem 4.4.2. *Escape never causes frame drops.*

Proof. Let f be a data flow traversing the sequence of nodes $N = [n_1, \ldots, n_m]$. f is HoL-blocked at time t at node $n_k \in N$. Let D^t and D^r be the sets of data frames of f transmitted by the source and received by the destination, respectively. According to 4.4.2, if there is no frame drop then $D^t = D^r$. The only way to violate this condition is that $\exists\ n_u \in N$ s.t. $D^r_u \neq D^t_u$ where D^r_u and D^t_u, are the sets of frames received and transmitted by n_u, respectively. Then $\exists\ d \in D^r_u$ s.t. $\vec{q}_u$ of n_u's egress port that d traverses is *full*.

Consider the following two cases that collectively cover the transmission of an escaping frame d of f.

1. Token-based buffer allocation in Escape (§ 4.3.3.1) ascertains that as token τ that n_k emits traverses the sequence of nodes $N' = [n_k, \ldots, n_l] \sqsubseteq N$ (lines 13 and 27), and causes d on n_l to escape, it has reserved space in $\forall\ \vec{q}_x$ on node $n_x \in N'$ (lines 14 and 28). I.e., τ *never* reaches n_l if $\exists\ \vec{q}_x$ of $n_x \in N'$ that Escape fails to reserve space on (lines 9 and 16). Thus $\nexists\ n_x \in N'$ s.t. $\vec{q}_x$ is full, as d reaches n_x.

2. Beyond n_k, d is not treated as an escaping frame (line 30). In the case that n_k's egress port p^E_k (for which τ is generated) gets blocked (i.e., PFC-activated) by the time d reaches p^E_k, n_k's downstream node has sufficient buffer space to absorb d since PFC requires a sufficiently large headroom buffer.

Based on cases 1 and 2, $\forall\ d \in D^r_u$, $\vec{q}_u$ of n_u's egress port that d traverses is not full. Therefore, $\forall\ n_u \in N\ D^r_u = D^t_u$, hence $D^t = D^r$. Thus Escape never causes frame drops. □

4.4.4 In Order Frame Delivery [ord]

Theorem 4.4.3. *Escape never causes out-of-order frame delivery.*

Proof. Let f be a data flow traversing the sequence of nodes $N = [n_1, \ldots, n_m]$, HoL-blocked *only* at time t at node $n_k \in N$. Let $D^t = [d_1, \ldots, d_u]$ be the sequence of data frames of f, transmitted by the source, and $D^r = [d_v, \ldots, d_w]$ the sequence of data frames received by the destination.

Without loss of generality, consider the following scenario. Let d_x and d_y be two data frames of f that escape at times t_x and t_y, respectively, where $t < t_x < t_y$, and are flowing downstream (i.e., in $\vec{q}$ or on the wire). Assume f unblocks at time t_z (at n_k) due to PFC deactivation where $t_x < t_y < t_z$. i.e., PFC deactivation occurs while escaping frames are being transmitted. As a result, there are *three* distinct frame sequences: $D_1 = [d_1, \ldots, d_{x-1}]$, $D_2 = [d_x, d_y]$, and $D_3 = [d_{y+1}, \ldots, d_u]$ s.t. $D_1 \sqcup D_2 \sqcup D_3 = D^t$. Here, D_1 is not affected by HoL block at n_k, and undergoes normal frame transmission, hence in-order delivery at the destination. Therefore, D_1 is received by the destination as D'_1 where $D'_1 \sqsubseteq D_1$.

Consider the *four* cases that can invert frame ordering of D_2 and D_3:

1. *Frame extraction: All* frames of f traverse the same sequence of nodes and same pair of ingress-egress ports within a node (i.e., unique deterministic path). A token in Escape always traces back this path (lines 7 and 24). A queue is scanned from head to tail when searching for frames (line 19). Therefore, $x < y$. In other words, d_x is extracted from the queue, before d_y.

2. *Queuing and scheduling:* Extracted frames awaiting transmission use a dedicated queue (§ 4.3.3.2), and as a result, they preserve their ordering (same as in 1) on the wire. Therefore, d_x precedes d_y on the wire. $\forall d_{\mathrm{i}} \in D_2, d_{\mathrm{j}} \in D_3$ awaiting transmission at the *same* egress port are scheduled for transmission at times t_{i}, t_{j}, respectively, where $t < t_{\mathrm{i}} < t_{\mathrm{j}}$. Therefore, D_2 precedes D_3 when they are queued and scheduled for transmission.

3. *Unblocking an intermediate port:* Regular traffic cannot resume (with PFC deactivation) until after $\vec{q}$ (§ 4.3.1.5) of an egress port is *empty* (§ 4.3.3.3). Prioritizing extracted

frames over regular frames assures the two types cannot mix, thus preventing frame order inversion in this case.

4. *Blocking the downstream egress port for which a token is generated:* If the open egress port p_k^E on n_k for which a token was originated activates PFC before an escaping frame leaves p_k^E, from 2 (above), escaping frames use $\vec{q}$ (while regular traffic is blocked) and preserve their ordering.

Based on these four cases, D_2 precedes D_3 during transmission and they are received by the destination as D_2' and D_3', respectively, where $D_2' \sqsubseteq D_2$ and $D_3' \sqsubseteq D_3$. Therefore, the destination receives D_1', D_2', and D_3' in that order, hence $D_1' \sqcup D_2' \sqcup D_3' = D^r \sqsubseteq D^t$. Thus based on 4.4.3, Escape does not cause out-of-order frame delivery. □

4.4.5 Termination of Algorithm

Theorem 4.4.4. *Escape HoL unblocking always terminates.*

Proof. An Escape HoL unblock operation begins as a token is generated (and transmitted upstream) by the switch (line 10), and it terminates when the token stops being transmitted. The following *three* cases collectively assert that a token always eventually ceases to exist:

1. The token pool of an egress port is exhausted when the token reaches the port, and as a result, the token is prevented from moving upstream along the path of the flow (line 17).

2. The token causes a frame to escape, and consequently, the token stops being transmitted (line 23). A token corresponds to *one* HoL-blocked frame that can escape.

3. The token reaches the source of the flow, and stops being transmitted, as a result of not finding a matching frame on any of the switches that it traversed.

Therefore, we can ascertain that an Escape HoL unblock operation always terminates. □

4.5 Implementation

We first consider Escape's implementation independently of any platform, then in FPGA [95] and DPDK [24].

4.5.1 Escape Components

Flow table. Each entry in this table has a flow's (1) identifier (5-tuple), (2) ingress port, and (3) egress port. Datacenter switches maintain similar flow state information, e.g., ETRAP [96], and *flowlet table* [97]. Shpiner et al. uses a similar flow table [83] that stores ingress-egress port pairs of flows traversing a switch.

Port registry. A bitmap is sufficient to maintain the ingress port numbers that requested PFC. Similarly, the egress port numbers that activated PFC can be maintained. Additionally, a vector (index is port number) is sufficient to maintain the current token pool size of each egress port.

Escape token frame. We propose ETF, a MAC frame with opcode 0x0102 as token frame. ETF includes the 5-tuple (13 bytes) of a data flow, and the number of hops (2 bytes) the frame travels from its origin.

Feasibility of extracting frames from queue. Escape requires the ability to "extract" frames from arbitrary locations in a queue. *Push-In-Extract-Out (PIEO)* [98] is a queue primitive that supports dequeuing frames from arbitrary locations in the queue. Feasibility of PIEO has been verified using a hardware implementation [98].

4.5.2 FPGA Implementation

The feasibility of Escape depends on its implementability in hardware, specifically with minimal (1) impact on the data path performance (i.e., latency), and (2) on-chip resource demand. We identify *queue supporting "extraction"* (§ 4.3.1.8) and *flow table* (§ 4.3.1.7) as the components required by Escape that can potentially increase data path latency and be resource-intensive. Therefore, we use Xilinx Vitis HLS 2020.2 [63] targeting Xilinx Virtex-7 XC7V2000T FPGA device to design and perform a high-level synthesis of: (1) a queue that supports frame extraction, and (2) a flow table that supports flow id update and lookup.

Queue. Our queue implementation uses a block RAM (BRAM) vector to store data frames. Each element of the vector is 2048 bytes (assuming MTU is 1500). The 5-tuple of each frame is kept in a registry vector enabling efficient lookup. Additionally, the queue im-

plementation uses two key structures: (1) *bitmap* to maintain the vector (frame and header) slot availability, and (2) *shift register* to maintain the FIFO ordering of the frames. Fig. 4.5 depicts the use of the bitmap and shift register in queue operations. This design maintains frame ordering after dequeue and extract, *without copying actual frames and header*. As a consequence, our queue achieves the same initiation interval (II) irrespective of size and throughput close to line rate. We synthesize a queue that has a capacity of 1024 frames (i.e.∼1.5 MB). Typically, per-port buffer capacity in modern datacenter switches is smaller than this value [48], [66]. Based on the Vitis synthesis report, the queue supports a maximum clock frequency of 226.71 MHz. Enqueue and dequeue, the two operations that affect data path performance, have a latency of 4.41 ns with initiation interval of 1, with 226.7 Mpps throughput. Assuming MTU of 1500, this is equivalent to 2.72 Tbps. The extract operation (required by Escape) has a latency of 13.23 ns with initiation interval 3, with 75.7 Mpps throughput. The queue only requires 4% FF and 10% LUT demonstrating its feasibility on FPGA.

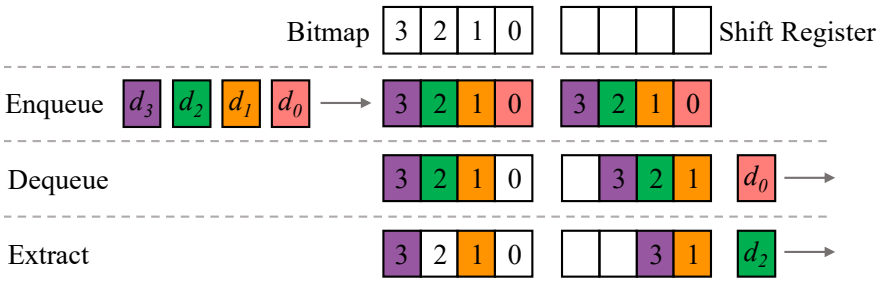


Figure 4.5. Overview of FPGA queue implementation: The bitmap maintains the vector indices of frames, and the shift register is used for FIFO ordering.

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.

4.5.3 DPDK Implementation

We have integrated Escape into DPDK [24]. Our implementation uses a *run-to-completion* model in which Escape's runtime is single-threaded and bound to a dedicated CPU core. The program polls a port assigned to it at initialization to read data frames, makes forwarding decisions for a received data frame, updates flow table, and adds the frame to its intended egress ring buffer. It polls its own egress ring buffer to transmit frames. The program uses function calls to update the flow table, forward frames, and transmit them. The program has a port table that uses DPDK atomic operations supporting read and write by different Escape runtime instances, which minimizes performance overhead on the data path. We implement PFC in the receiver module.

DPDK extension. The default ring buffer in DPDK (`struct rte_ring`) does not support random access, needed for frame extraction. We propose and implement API extensions (cf. [99]) supporting frame extraction. We define the function type, `comp_obj` that compares two objects and returns `1` if they are equal; `0` otherwise. In our implementation, we use a comparator of type `comp_obj` to verify if a data frame (`struct rte_mbuf *`) matches a given flow identifier (`char *`). The new function `rte_ring_sc_jump` (25 lines of code) takes a pointer to a ring buffer, a comparator function (`func`), and a reference value to use with the comparator (`obj2`) as inputs and returns a pointer to the first object matching the comparison based on `func` and `obj2` (flow identifier in our case). The function removes the pointer from the ring and updates its internal pointers (i.e., consumer tail and head tail). In the absence of a match, `NULL` is returned.

4.6 Evaluation

We evaluate Escape as follows. (A) *Micro-benchmarks* show that Escape meets its design goals. (B) *Testbed experiments* confirm that Escape behavior with our DPDK implementa-

tion is in line with our simulations. (C) *Comparisons with the state of the art* show that Escape has a lower FCTs as a result of clearing HoL blocking, even in cases when it does not lead to a deadlock (D). We use OMNeT++ [100] to compare Escape and several state-of-the-art solutions, which we implement since they do not have publicly available OMNeT++ implementations. We configure the solutions based on details given in the respective papers [48], [83]. We use traffic workloads derived from two commonly-used traffic traces [44], [89]. Unless otherwise specified, the average link load is set to 70%. We use output-queued switches with a byte counter for each ingress port to trigger PFC (q_i^I denotes byte counter of port i). We configure PFC based on [66].

4.6.1 Micro-Benchmarks

Three-node topology. We use the deadlock-prone topology in Fig. 4.6 to show the effectiveness of Escape in clearing routing deadlocks. Switches s_1, s_2, and s_3 are connected in a ring and end hosts h_1, h_2, and h_3 are connected to the respective switches. All interconnects use 40 Gbps links. A data sender and receiver on each end host form three loop-free data flows. All senders start transmitting data simultaneously. We observe q_1^I, q_2^I, and q_3^I on s_1, s_2, and s_3 receiving traffic through l_3, l_1, and l_2, respectively, over time to determine deadlock formation (all ports simultaneously XOFF) in the absence of Escape, and deadlock clearance in its presence (ports fluctuate between XOFF and XON).

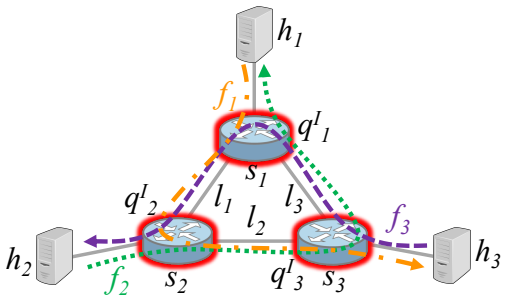
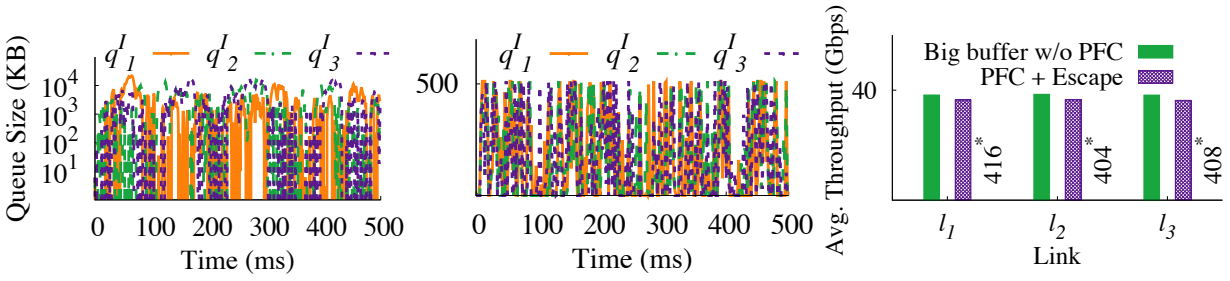


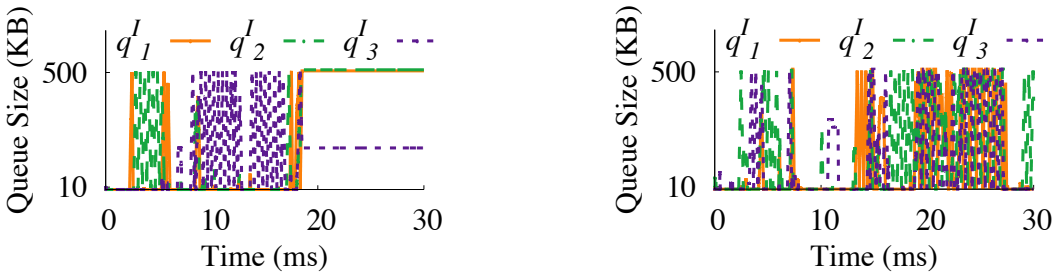
Figure 4.6. Simple three-node topology forming deadlock $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_1$.



(a) Large buffer. Without PFC. (b) PFC + Escape. (c) Throughput with Escape.
(*Number of PFC activations)

Figure 4.7. Efficiency of Escape on the three-node topology in Fig. 4.6.

Without Escape, q_1^I, q_2^I, and q_3^I do not reach the XON threshold after ∼18 ms (Fig. 4.8a) due to deadlock formation. With Escape, q_1^I, q_2^I, and q_3^I fluctuate between XOFF and XON thresholds (Fig. 4.8b), confirming absence of deadlock [CLR].



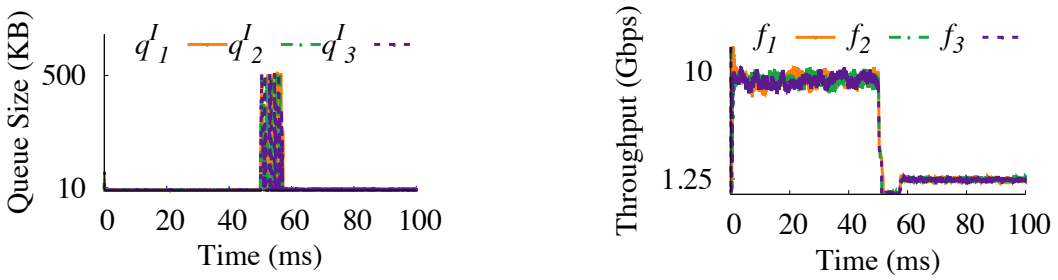
(a) Without Escape. (b) With Escape.

Figure 4.8. Deadlock clearance with Fig. 4.6 topology.

We compare average link utilization with Escape to ideal link utilization, which can be attained with unlimited buffer space and thus no PFC activation. We conduct the experiment for 500 ms, first with PFC disabled and then with PFC and Escape enabled. Fig. 4.7a and Fig. 4.7b show the variation of q_1^I, q_2^I, and q_3^I without PFC (ideal) and with PFC and Escape, respectively. As we use large buffers, the queue size can grow (note the y axis scale) without activating PFC. In the case of PFC and Escape, the buffer usage does not exceed the PFC threshold as a result of PFC activation [EFC]. As shown in Fig. 4.7c, links l_1, l_2, and l_3 experience several PFC activations with Escape, while links l_1, l_2, and l_3 achieve high link utilization (91%), close to the ideal value (96%) [EFC].

Effect of congestion control. We investigate whether the topology in Fig. 4.6 is deadlock-prone when congestion control is present. We implement and configure the state-of-the-art HPCC algorithm according to the details given in [89]. We initially use two competing flows at each source. To study the impact of *slow reaction* of congestion control to deadlock formation, we start 14 new data flows from each source node after 50 ms, creating congestion on the switches. We find that q_1^I, q_2^I, and q_3^I simultaneously reach their XOFF thresholds almost immediately causing PFC activation and deadlock.

With Escape, there is no deadlock. As seen in Fig. 4.9a, while incast congestion builds (at 50 ms), q_1^I, q_2^I, and q_3^I fluctuate between their XOFF and XON thresholds as a result of Escape activation. Meanwhile, congestion control slowly takes over and queues drain, avoiding deadlock. Fig. 4.9b shows how the throughput attained by three flows f_1, f_2, and f_3 originating at nodes h_1, h_2, and h_3, respectively, varies over time. Before congestion occurs, each flow attains its fair share link bandwidth of ∼10 Gbps (4 competing flows on each core link). With congestion, the network becomes unstable; Escape together with congestion control stabilizes it. Further, flow throughput stabilizes at ∼1.25 Gbps (32 competing flows on each core link) indicating system convergence with the aid of congestion control.



(a) Queue size.

(b) Throughput convergence.

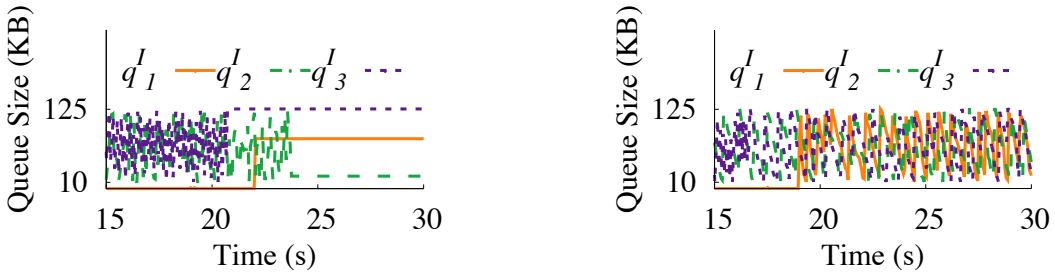
Figure 4.9. Deadlock clearance in Fig. 4.6 with Escape and HPCC.

4.6.2 DPDK-based Prototype Evaluation

We evaluate Escape on CloudLab [25] using our DPDK-based implementation on the topology of Fig. 4.6. Each node is a Dell Poweredge R430 machine with 2 2.4 GHz 64-bit 8-Core Xeon processors, 64 GB DDR4 RAM, and 2 Intel X710 10 GbE NICs. Each node s_1,

s_2, and s_3 is capable of working as a 10 GbE 3-port switch. We use DPDK version 18.11.2 on Ubuntu 18.04.1 LTS with `igb_uio` as kernel driver. We use `iperf` v. 2.0.13 as traffic generator. We set PFC XOFF and XON thresholds to 125 KB and 10 KB, respectively.

Fig. 4.10a shows that, with Escape disabled, q_1^I, q_2^I, and q_3^I do not reach the XON threshold after ∼24 s as a result of deadlock formation. We can recreate this result and deadlock every time we run this scenario. In contrast, with Escape enabled, q_1^I, q_2^I, and q_3^I fluctuate between XOFF and XON thresholds (Fig. 4.10b), implying absence of deadlock.



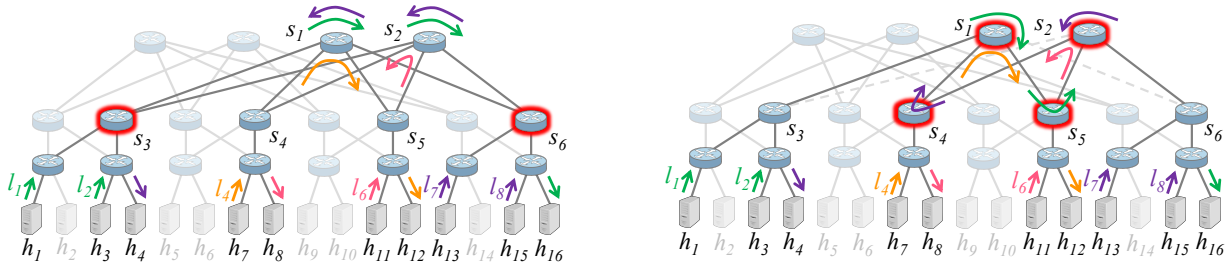
(a) Without Escape. (b) With Escape.

Figure 4.10. Deadlock clearance on the testbed topology.

4.6.3 Datacenter Simulations

We evaluate Escape on a scenario mimicking a modern datacenter topology, load, and traffic. We use HPCC [89] for congestion control. We use the three-level fat-tree topology depicted in Fig. 4.11a with 8 edge switches, 8 aggregation switches, and 4 core switches. It has 8 source nodes (h_1, h_3, h_5, ..., h_{15}) and 8 destination nodes (h_2, h_4, h_6, ..., h_{16}) connected to the edge switches. Each interconnection is a 40 Gbps link. Each source node starts 10 concurrent data flows. The flows from h_1 and h_3 are destined to h_{16} and create incast congestion on s_6. Similarly, the flows from h_{13} and h_{15} are destined to h_4, creating incast congestion on s_3. The flows from h_7 and h_{11} are destined to h_{12} and h_8, respectively, and create little or no congestion. Similarly, the flows from h_5 and h_9 are destined to h_{10} and h_6, respectively, creating little or no congestion. As the second step of this experiment, we disable the links $s_1 \rightarrow s_6$ and $s_2 \rightarrow s_3$, causing the flows destined to h_{16} and h_6 to take "bouncy" paths. As shown in Fig. 4.11b, the flows from h_1 and h_3 take the new route $s_3 \rightarrow s_1 \rightarrow s_5 \rightarrow s_2 \rightarrow s_6 \rightarrow h_{16}$

and the flows from h_{13} and h_{15} the route $s_6 \rightarrow s_2 \rightarrow s_4 \rightarrow s_1 \rightarrow s_3 \rightarrow h_3$. We run the two steps with Escape disabled and then with it enabled. We record link utilization l_{i} at each source node h_{i} to observe network stability. When Escape is enabled, we additionally record the size of ingress queue q^I_{j} of node s_{j} on the deadlock-prone path $s_1 \rightarrow s_5 \rightarrow s_2 \rightarrow s_4 \rightarrow s_1$.



(a) Before link failure.

(b) After link failure.

Figure 4.11. Three-level fat-tree topology used in the datacenter simulations.

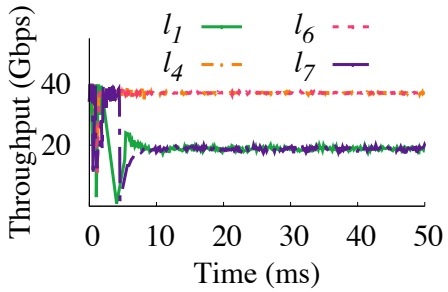
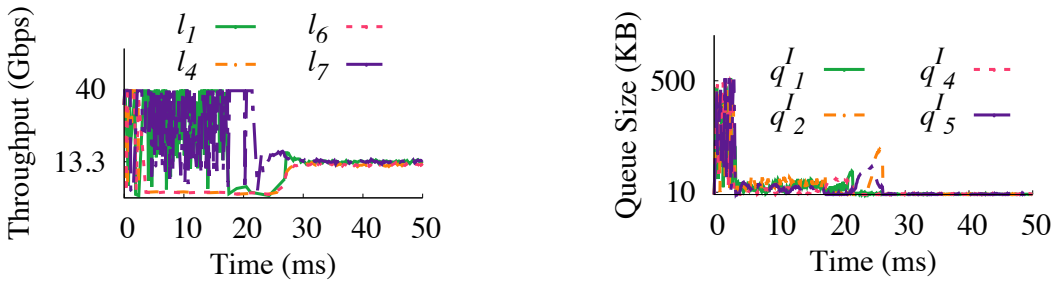


Figure 4.12. System stability (link utilization) before link failure (HPCC + Escape).

Fig. 4.12 shows the link utilization before link failure. The flows from h_1 and h_3 go through s_6, equally sharing the bottleneck link bandwidth, and as a result, the utilization of l_1 (and l_2) stabilizes at ∼20 Gbps. Further, the throughput attained by each of the 10 competing flows on the link, stabilizes at ∼2 Gbps. Similarly, the flows from h_{13} and h_{15} go through the bottleneck link at s_3, and the utilization of l_7 (and l_8) stabilizes at ∼20 Gbps. The flows from h_7 and h_{11} do not create congestion, hence l_4 and l_6 are fully utilized. Without Escape, switches s_1, s_2, s_4, and s_5 instantly run into a deadlock in step two (see Fig. 4.11b), causing the flows going through these switches to halt. With Escape, the link failure does not cause a deadlock and each sender attains its fair share link bandwidth of ∼13.3 Gbps, as Fig. 4.13a shows.

To understand the impact of Escape on system stability under normal conditions and when there is a potential deadlock, we observed ingress queue depths at congestion points and instantaneous bandwidths attained by the completing senders. Fig. 4.12 shows that throughput convergence of competing senders (h_1, h_3 and h_{13}, h_{15}) is as expected in a network with HPCC. The system converges in ∼5 ms. As a result of the link failure, the nodes on the deadlock-prone loop become the new congestion hot spots resulting in a different set of competing senders (h_1, h_3, h_7, and h_{11}, h_{13}, h_{15}). The system converges and each competing sender attains its fair share in ∼20 ms (Fig. 4.13a). At the same time, the queues drain (Fig. 4.13b) indicating system stability. We can conclude that Escape clears deadlocks caused by HoL blocking in practical fat-tree topologies, while preserving network stability.



(a) Link utilization.

(b) Queue size.

Figure 4.13. System stability after link failure (HPCC + Escape).

4.6.4 Escape vs. State of the Art

We compare Escape with state of the art, in terms of their effectiveness and efficiency as solutions to routing deadlocks. We carefully select our reference solutions considering *three* widely explored deadlock resolution models:

(1) Detect deadlock and drop frames to resolve it: Dropping frames is believed to be a viable *reactive* solution to deadlock. We use the token-passing technique in Loop Breaker (LB) [83] to efficiently detect a deadlock.

(2) Detect deadlock and facilitate traffic progress: Deadlock Breaker (DLB) [83] (another *reactive* solution) uses token-passing to detect a deadlock, and allocates extra buffer space along the path of the deadlock to facilitate traffic progress.

(3) Replace PFC with rate-based flow control: GFC [48] is a *proactive* solution that replaces PFC to prevent routing deadlocks.

We use the topology in Fig. 4.14 with the three reference solutions and Escape to investigate their: (i) ability to resolve a deadlock [CLR], and (ii) tail latencies [EFC]. Each source node, h_2, h_4, h_6, and h_8, has 10 traffic generators sending data to destination nodes, h_7, h_5, h_3, and h_1, respectively.

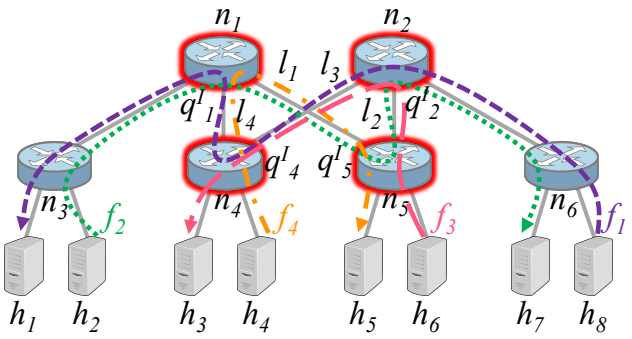
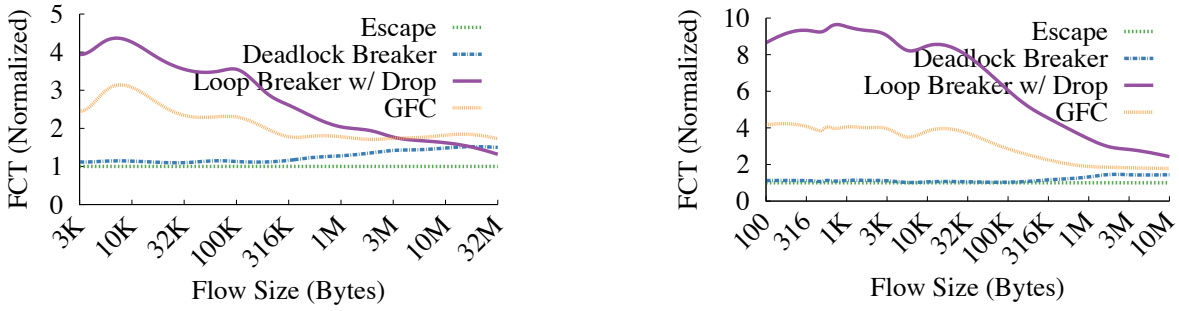


Figure 4.14. Two-level fat-tree topology that forms the deadlock, $n_1 \rightarrow n_5 \rightarrow n_2 \rightarrow n_4 \rightarrow n_1$.

We observe that LB, DLB, and Escape resolve deadlocks, while GFC prevents them. However, LB, DLB, and GFC incur higher FCT tail latencies than Escape (Fig. 4.15a and Fig. 4.15b). Short flows, which constitute the majority of traffic in datacenters, are more susceptible to increased tail latency. One critical observation is that dropping frames to resolve deadlocks (LB) incurs the highest tail latency, which is up to $\sim$10$\times$ that of Escape. Tail latency of GFC is up to $\sim 2-4\times$ that of Escape, which highlights that PFC (XOFF/XON) with Escape is better than GFC (rate-based). The tail latencies of DLB are at least $\sim$10% higher than those of Escape. Of the three reference solutions we use, DLB is more effective in resolving deadlocks, reinforcing Escape's premise: *to clear blocked traffic, allocate buffers to facilitate traffic progress.* However, DLB is not able to clear a HoL block when it does not lead to a deadlock.

4.6.5 FCT Reduction in Absence of Deadlocks

We study FCTs using the topology in Fig. 4.16. Switches $s_2 \ldots s_5$ connected to switch s_1, hosts h_1 and h_2 to s_2, and h_3 and h_4 to s_3 by 100 Gbps links. Hosts h_5 and h_6 are



(a) *WebSearch* traffic.

(b) *FB_Hadoop* traffic.

Figure 4.15. 99^{th}-percentile FCT in the fat-tree topology in Fig. 4.14.

connected to s_4 and s_5, respectively, by 40 Gbps links. Hosts $h_1 \ldots h_4$ are data sources and h_5, h_6 destinations. h_1 and h_3 transmit flows f_1 and f_3, respectively to h_6, and h_2 and h_4 flows f_2 and f_4, respectively, to h_5. We set the average load of heavy flows (f_1 and f_3) to 80% and of innocent (victim) flows [101] (f_2 and f_4) to 40%.

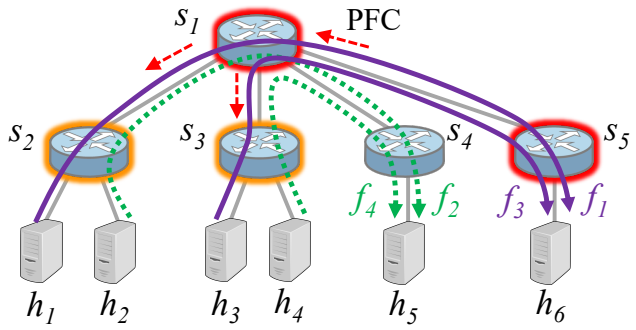
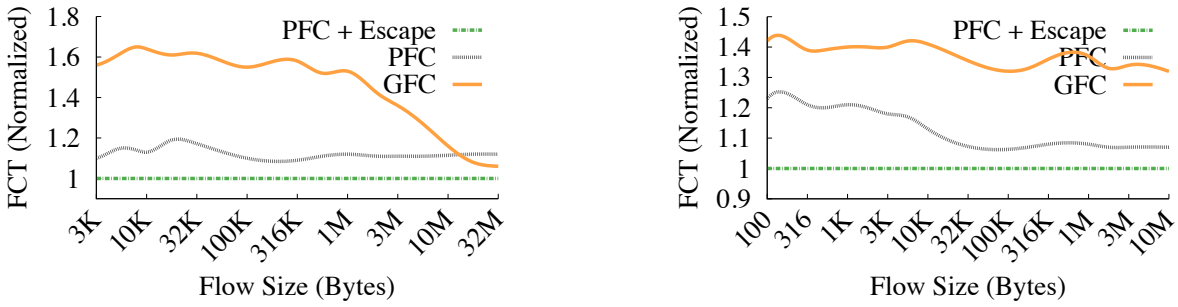


Figure 4.16. Two-level fat-tree topology with incast congestion on $s_1 \rightarrow s_5$ and little or no congestion on $s_1 \rightarrow s_4$.

This scenario is *not* deadlock-prone, but f_2 and f_4 can suffer from HoL blocking on s_1, as other flows create congestion. This is a common scenario in datacenter networks [87], [89]. We investigate if Escape reduces the FCTs of f_2 and f_4 as it clears HoL-blocked frames. We enable PFC and record average FCT of innocent flows with and without Escape. We repeat the experiment with GFC (PFC and Escape disabled).

As Fig. 4.17a shows, average FCT for flows smaller than 1 MB in *WebSearch* traffic is at least ∼10% higher without Escape, and at least ∼50% higher with GFC. With *FB_Hadoop*

traffic (Fig. 4.17b), all flow sizes experience at least ∼10% higher FCTs without Escape, and at least ∼30% higher with GFC. Thus, innocent flows obstructed by congestion achieve reductions in average FCT with Escape. Datacenter network traffic mostly comprises short flows (⩽10 KB) [79], [80] and based on these results, innocent flows on a single bottleneck experience ∼20% increase in average FCT *without* Escape. Replacing PFC and Escape with GFC increases average FCT by $\sim 40 - 60\%$.



(a) *WebSearch* traffic.

(b) *FB_Hadoop* traffic.

Figure 4.17. FCT of innocent flows with incast congestion.

4.7 Related Work

Congestion control ideally suppresses PFC activation and HoL blocking. However, state-of-the-art datacenter congestion control mechanisms (e.g., DCQCN [87], TIMELY [88], HPCC [89]) do not completely eliminate PFC due to their long reaction times stemming from the congestion signal propagation delays (i.e., ECN [87], RTT [88], and INT [89]).

Deadlock solutions address HoL blocking to some extent by facilitating traffic movement. Deadlock handling can be *proactive* or *reactive*. Proactive deadlock solutions mostly employ adaptive routing [102]–[107] to eliminate cyclic buffer dependencies, which requires significant changes to existing routing protocols. GFC [48] modifies PFC XOFF/XON behavior, but incurs implementation and operational overhead. Preventing buffer overrun due to traffic bursts (the rationale for PFC XOFF) may not be possible with GFC, and configuring GFC (e.g., its mapping function) requires considerable effort. Reactive deadlock solutions detect and resolve deadlocks, but incur significant implementation and performance overhead. A switch identifies ports prone to deadlock via HoL blocking and uses a probing technique (e.g.,

token passing) to ascertain that the switch is indeed on a deadlock. Switches on a deadlock then use a distributed algorithm (e.g., consensus-based) to break the deadlock. Loop Breaker [83] uses a token to chose a master to resolve the deadlock and frame rerouting to break it, assuming each flow that needs rerouting has an alternative path to its destination. Even if this assumption holds, rerouting can cause out-of-order frame delivery causing additional overhead for frame reordering and/or retransmission. Deadlock Breaker [83] uses the same deadlock detection technique as Loop Breaker, and facilitates traffic progress using a token passing mechanism for extra buffer allocation along the deadlock loop. The buffer allocation management can be complex, especially when a port is on multiple deadlocks simultaneously. Algorithm termination also introduces complexity. Disha [108] uses a set-aside buffer to route HoL-blocked frames, but a frame would need exclusive access to this buffer along the path of the frame when multiple nodes on the deadlock start the recovery process concurrently.

Flow prioritization mitigates HoL blocking. PFC itself was proposed as a solution to HoL blocking by operating on 8 distinct priority levels, as opposed to pausing all traffic on a link (as in PAUSE). By selectively pausing traffic according to class, PFC can mitigate HoL blocking at the class level but not at the flow level. QJump [90] uses coarse-grained priority levels to resolve interference for latency-sensitive traffic by prioritizing it over throughput-sensitive traffic. pFabric [91] prioritizes flows based on remaining flow size, and drops frames to reduce FCTs of short flows. These approaches cannot prevent flow-level HoL blocking.

Virtual output queue (VOQ) architectures [109] were proposed as a solution to HoL blocking at *ingress* ports. However, VOQs do not prevent HoL blocking at egress ports.

Frame detouring is an approach for minimizing loss without using PFC. Detour-induced buffer sharing (DIBS) [110] randomly detours frames received by a switch to its neighbors when there is no buffer space to store the frames. DIBS can cause out-of-order frame delivery and livelocks. It can also increase FCT of small flows, which constitute the majority of flows in a datacenter network.

4.8 Chapter Summary

We propose a new flow control protocol for modern datacenter networks that allows HoL-blocked frames at congestion points to leave queues and move downstream without getting rerouted. Escape eliminates routing deadlocks and reduces FCTs, especially for short flows which constitute the majority of flows in datacenters. We believe that Escape can be easily extended to propagate useful meta-data to aid other critical network services, e.g., congestion control and load balancing. The key components required by Escape can be efficiently implemented in hardware.

5. CONCLUSIONS AND FUTURE WORK

With the advent of modern line-rate switch architectures, the need for advanced yet simple specification languages for defining buffer architectures has become more important than ever before. In this dissertation, we introduce a new specification language, OpenQueue, that allows users to specify entire buffering architectures and policies conveniently through several comparators and simple functions, which makes OpenQueue ideal for modern programmable line-rate switch architectures that use match-action tables.

We also introduced a new congestion control solution for RDMA networks in this dissertation. *RoCC* can outperform well established similar solutions that are used in production right now. *RoCC* guarantees fairness when allocating bandwidth among competing flows. It is able to maintain stability, converge fast, and attain maximum link utilization. These requirements are of paramount importance in effective congestion control in modern datacenter networks.

We have also developed a solution for datacenter networks that deals with routing deadlocks.

5.1 OpenQueue Implementation in Hardware

"Over the next decade, I believe that networks are going to become end-to-end programmable with programmable pieces at every level" - keynote by Nick McKeown at Netdev 2020

As network programmability becomes widespread, it is becoming customary to have programmable network components at different levels.

Network programming frameworks such as P4 [111] enables significant amount of programming support in the data plane. However, to the best of our knowledge, buffer management and packet scheduling still lacks programming support, nevertheless they play a critical role in network performance. For example, the traffic manager in P4, which handles packet buffering and scheduling, is not programmable, hence fails to harness the full potential of P4.

We believe OpenQueue is capable of providing comprehensive language support for buffer management. However, the feasibility of OpenQueue in different hardware platforms has not been adequately investigated. We plan to implement OpenQueue using hardware technologies such as FPGA and ASIC. It is important to assess the feasibility of OpenQueue in terms of hardware resource demand and data plane latency overhead.

5.2 Predictive Congestion Control

Based on our experience on congestion control solutions in datacenter networks, there are *two* key challenges they have to deal with.

1. A congestion control solution usually comes with a set of parameters that directly affects its behavior. Tuning these parameters is challenging as they usually affect some conflicting goals (e.g. fast convergence vs. high stability). Given that datacenter traffic is very dynamic, it is difficult to tune a congestion control solution to meet its intended goals despite it claims to achieve its goals in theory.

2. Most of the existing congestion control solutions (including *RoCC*) are *reactive*, hence they by nature take some time to take measures to mitigate congestion. For example, DCQCN [37] suffers from excessive reaction delays due to queuing of marked packets, and both DCQCN [37] and TIMELY [38] can be over corrective when congestion is not persistent. Reaction delay impacts stability and other important properties of any reactive congestion control solution.

This is where the concept of *Predictive Congestion Control* comes into the picture. Predictive congestion control has been studied to some extent in other network domains [112]–[114] but to little or no extent in datacenter network domain. We plan to look into *two* possible approaches:

1. Use *learnability* of network traffic patterns to predict network flows and use that knowledge for proactive congestion control.

2. Use switch queue size as an indication of network congestion level. It is possible to use some technique to estimate queue length for next prediction interval by using

queue lengths of past intervals [115] and use that knowledge to do congestion control proactively.

5.3 Extensibility of Escape

We plan to extend the work on Escape in *two* directions:

I. Hardware implementation: We have investigated the hardware feasibility of the key components of Escape. However, in order to assess the hardware feasibility of Escape, it needs to be fully implemented using a widely used hardware technology such as FPGA or ASIC. We plan to implement Escape in FPGA.

II. Escape to support other network services: Escape uses "backward" notification to propagate congestion information upstream. Critical network services such as congestion control and load balancing rely on network congestion information provided by mechanisms such as ECN [37] and INT [39]. These mechanism use "forward" notification, hence fail to convey congestion information to the hosts in a timely manner for quickly mitigating network congestion. The delay in congestion notification is a critical challenge that congestion control faces. Therefore, it is interesting to investigate the possibility of integrating Escape control frame (ECF) with congestion control and load balancing to improve their effectiveness.

5.4 Large-scale Evaluation

Evaluating our solutions in a real-life setting is critical to assess their deployment readiness. So far, we have performed some limited evaluations of *RoCC* and Escape using public testbeds and some limited testbed facilities we possess. However, we have identified the following key evaluation scenarios to further evaluate *RoCC* and Escape.

I RoCC: Use the P4-based implementation of *RoCC* in a setup resembling a real datacenter network (e.g., two-level leaf-spine topology) to evaluate *RoCC* in terms of its high-level goals: (1) reduce tail latency, and (2) minimize PFC activation.

II Escape: Use the full hardware implementation of Escape (see § 5.3-I) to evaluate the FCT of Escape using a testbed that resembles a real datacenter network setup. We plan to

use two scenarios here: (1) a deadlock-prone setup, and (2) a setup that is not deadlock-prone and innocent flows suffer from HoL block.

III RoCC + Escape: It is interesting to investigate the behavior of a real datacenter network setup when both *RoCC* and Escape coexist. We plan to use a two-level leaf-spine topology to observe tail-latency and PFC activations in this scenario, with two widely-used traffic distributions, *WebSearch* traffic and *FB_Hadoop* traffic.

REFERENCES

[1] M. Goldwasser, "A survey of buffer management policies for packet switches," *SIGACT News*, vol. 41, no. 1, pp. 100–128, 2010. DOI: 10.1145/1753171.1753195.

[2] M. Alizadeh, S. Yang, M. Sharif, *et al.*, "Pfabric: Minimal near-optimal datacenter transport," in *ACM Computer Communication Review (CCR)*, 2013, pp. 435–446. DOI: 10.1145/2534169.2486031.

[3] K. Kogan, A. López-Ortiz, S. I. Nikolenko, G. Scalosub, and M. Segal, "Balancing work and size with bounded buffers," in *International Conference on Communication Systems and Networks (COMSNETS)*, 2014, pp. 1–8. DOI: 10.1109/COMSNETS.2014.6734878.

[4] P. Chuprikov, S. I. Nikolenko, and K. Kogan, "Priority queueing with multiple packet characteristics," in *IEEE International Conference on Computer Communications (INFOCOM)*, 2015, pp. 1418–1426. DOI: 10.1109/INFOCOM.2015.7218519.

[5] A. Davydow, P. Chuprikov, S. I. Nikolenko, and K. Kogan, "Throughput optimization with latency constraints," in *IEEE International Conference on Computer Communications (INFOCOM)*, 2017, pp. 1–9. DOI: 10.1109/INFOCOM.2017.8057015.

[6] N. McKeown and et al., *OpenFlow switch specification*, http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf, 2011.

[7] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *IEEE/ACM Transactions on Networking (ToN)*, vol. 1, no. 4, pp. 397–413, 1993. DOI: 10.1109/90.251892.

[8] K. M. Nichols and V. Jacobson, "Controlling queue delay," *Communications of the ACM*, vol. 55, no. 7, pp. 42–50, 2012. DOI: 10.1145/2209249.2209264.

[9] A. J. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queueing algorithm," in *ACM Computer Communication Review (CCR)*, 1989, pp. 1–12. DOI: 10.1145/75247.75248.

[10] P. McKenney, "Stochastic fairness queueing," in *IEEE International Conference on Computer Communications (INFOCOM)*, 1990, pp. 733–740.

[11] W. Aiello, A. Kesselman, and Y. Mansour, "Competitive buffer management for shared-memory switches," *ACM Transactions on Algorithms*, vol. 5, no. 1, 2008. DOI: 10.1145/1435375.1435378.

[12] P. Eugster, A. Kesselman, K. Kogan, S. I. Nikolenko, and A. Sirotkin, "Essential traffic parameters for shared memory switch performance," in *Structural Information and Communication Complexity (SIROCCO)*, 2015, pp. 61–75. DOI: 10.1007/978-3-319-25258-2_5.

[13] P. T. Eugster, K. Kogan, S. I. Nikolenko, and A. Sirotkin, "Shared memory buffer management for heterogeneous packet processing," in *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2014, pp. 471–480. DOI: 10.1109/ICDCS.2014.55.

[14] A. Mekkittikul and N. McKeown, "A practical scheduling algorithm to achieve 100% throughput in input-queued switches," in *IEEE International Conference on Computer Communications (INFOCOM)*, 1998, pp. 792–799. DOI: 10.1109/INFCOM.1998.665102.

[15] A. Kesselman, K. Kogan, and M. Segal, "Improved competitive performance bounds for CIOQ switches," *Algorithmica*, vol. 63, no. 1-2, pp. 411–424, 2012. DOI: 10.1007/s00453-011-9539-9.

[16] S. Chuang, S. Iyer, and N. McKeown, "Practical algorithms for performance guarantees in buffered crossbars," in *IEEE International Conference on Computer Communications (INFOCOM)*, 2005, pp. 981–991. DOI: 10.1109/INFCOM.2005.1498327.

[17] A. Kesselman, K. Kogan, and M. Segal, "Packet mode and QoS algorithms for buffered crossbar switches with FIFO queuing," *Distributed Computing*, vol. 23, no. 3, pp. 163–175, 2010. DOI: 10.1007/s00446-010-0114-4.

[18] R. Mittal, R. Agarwal, S. Ratnasamy, and S. Shenker, "Universal packet scheduling," in *USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2016, pp. 501–521.

[19] S. Bauer, R. Beverly, and A. W. Berger, "Measuring the state of ECN readiness in servers, clients, and routers," in *ACM Internet Measurement Conference (IMC)*, 2011, pp. 171–180. DOI: 10.1145/2068816.2068833.

[20] I. Keslassy, K. Kogan, G. Scalosub, and M. Segal, "Providing performance guarantees in multipass network processors," *IEEE/ACM Transactions on Networking (ToN)*, vol. 20, no. 6, pp. 1895–1909, 2012. DOI: 10.1109/TNET.2012.2186979.

[21] A. Ioannou and M. Katevenis, "Pipelined heap (priority queue) management for advanced scheduling in high-speed networks," *IEEE/ACM Transactions on Networking (ToN)*, vol. 15, no. 2, pp. 450–461, 2007. DOI: 10.1109/TNET.2007.892882.

[22] A. Sivaraman, S. Subramanian, A. Agrawal, *et al.*, "Towards programmable packet scheduling," in *ACM Workshop on Hot Topics in Networks (HotNets)*, 2015, 23:1–23:7. DOI: 10.1145/2834050.2834106.

[23] *Openqueue on github*, https://github.com/openqueuenew/icnp.

[24] *Intel DPDK*, http://dpdk.org/, 2019.

[25] D. Duplyakin, R. Ricci, A. Maricq, *et al.*, "The design and operation of CloudLab," in *USENIX Annual Technical Conference (ATC)*, 2019.

[26] N. Foster and et al., "Frenetic: A network programming language," in *ACM SIGPLAN Notices*, 2011, pp. 279–291. DOI: 10.1145/2034574.2034812.

[27] C. Monsanto and et al., "Composing software defined networks," in *USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2013, pp. 1–13.

[28] A. Voellmy and et al., "Maple: Simplifying SDN programming using algorithmic policies," in *ACM Computer Communication Review (CCR)*, 2013, pp. 87–98. DOI: 10.1145/2534169.2486030.

[29] R. Soulé and et al., "Merlin: A language for provisioning network resources," in *International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2014, pp. 213–226. DOI: 10.1145/2674005.2674989.

[30] A. Ferguson and et al., "Participatory networking: An API for application control of sdns," in *ACM Computer Communication Review (CCR)*, 2013, pp. 327–338. DOI: 10.1145/2534169.2486003.

[31] P. Bosshart, D. Daly, G. Gibb, *et al.*, "P4: programming protocol-independent packet processors," *ACM Computer Communication Review (CCR)*, vol. 44, no. 3, pp. 87–95, 2014. DOI: 10.1145/2656877.2656890.

[32] H. Song, "Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane," in *ACM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, 2013, pp. 127–132. DOI: 10.1145/2491185.2491190.

[33] C. Kozanitis and et all., "Leaping multiple headers in a single bound: Wire-speed parsing using the kangaroo system," in *IEEE International Conference on Computer Communications (INFOCOM)*, 2010, pp. 830–838. DOI: 10.1109/INFCOM.2010.5462139.

[34] A. Sivaraman, K. Winstein, S. Subramanian, and H. Balakrishnan, "No silver bullet: Extending SDN to the data plane," in *ACM Workshop on Hot Topics in Networks (HotNets)*, 2013, 19:1–19:7. DOI: 10.1145/2535771.2535796.

[35] A. Sivaraman, S. Subramanian, A. Agrawal, *et al.*, "Programmable packet scheduling at line rate," in *ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*, 2016, pp. 44–57. DOI: 10.1145/2934872.2934899.

[36] R. Mittal, R. Agarwal, S. Ratnasamy, and S. Shenker, "Universal packet scheduling," in *USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2016, pp. 501–521.

[37] Y. Zhu, H. Eran, D. Firestone, *et al.*, "Congestion control for large-scale rdma deployments," *ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*, vol. 45, no. 4, 2015. DOI: 10.1145/2829988.2787484.

[38] R. Mittal, V. T. Lam, N. Dukkipati, *et al.*, "Timely: Rtt-based congestion control for the datacenter," *ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*, vol. 45, no. 4, 2015. DOI: 10.1145/2785956.2787510.

[39] Y. Li, R. Miao, H. H. Liu, *et al.*, "HPCC: High precision congestion control," in *ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*, Beijing, China: ACM, 2019, ISBN: 978-1-4503-5956-6. DOI: 10.1145/3341302.3342085.

[40] R. Mittal, A. Shpiner, A. Panda, *et al.*, "Revisiting Network Support for RDMA," in *ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*, vol. 18, 2018. DOI: 10.1145/3230543.3230557.

[41] T. Benson, A. Akella, and D. A. Maltz, "Network Traffic Characteristics of Data Centers in the Wild," in *Internet Measurement Conference (IMC)*, 2010. DOI: 10.1145/1879141.1879175.

[42] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The Nature of Datacenter Traffic: Measurements & Analysis," in *Internet Measurement Conference (IMC)*, 2009. DOI: 10.1145/1644893.1644918.

[43] M. Alizadeh, A. Greenberg, D. A. Maltz, *et al.*, "Data Center TCP (DCTCP)," in *ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*, 2010. DOI: 10.1145/1851182.1851192.

[44] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout, "Homa: A receiver-driven low-latency transport protocol using network priorities," in *ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*, 2018. DOI: 10.1145/3230543.3230564.

[45] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the Social Network's (Datacenter) Network," in *ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*, 2015. DOI: 10.1145/2785956.2787472.

[46] I. D. Standard, *802.1Qbb - Priority-based Flow Control*, 2008.

[47] S. Hu, Y. Zhu, P. Cheng, *et al.*, "Deadlocks in datacenter networks: Why do they form, and how to avoid them," in *ACM Workshop on Hot Topics in Networks (HotNets)*, 2016. DOI: 10.1145/3005745.3005760.

[48] K. Qian, W. Cheng, T. Zhang, and F. Ren, "Gentle flow control: Avoiding deadlock in lossless networks," in *ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*, 2019. DOI: 10.1145/3341302.3342065.

[49] Y. Zhu, M. Ghobadi, V. Misra, and J. Padhye, "ECN or Delay: Lessons Learnt from Analysis of DCQCN and TIMELY," in *International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2016. DOI: 10.1145/2999572.2999593.

[50] Y. Gao, Y. Yang, T. Chen, J. Zheng, B. Mao, and G. Chen, "Taming Large-scale Incast Congestion in RDMA over Ethernet Networks," in *IEEE International Conference on Network Protocols (ICNP)*, 2018. DOI: 10.1109/ICNP.2018.00021.

[51] M. Alizadeh, B. Atikoglu, A. Kabbani, *et al.*, "Data Center Transport Mechanisms: Congestion Control Theory and IEEE Standardization," in *Annual Allerton Conference on Communication, Control, and Computing*, 2008.

[52] P. Bosshart, D. Daly, G. Gibb, *et al.*, "P4: Programming protocol-independent packet processors," *ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*, vol. 44, no. 3, 2014. DOI: 10.1145/2656877.2656890.

[53] S. Ibanez, G. Antichi, G. Brebner, and N. McKeown, "Event-driven packet processing," in *ACM Workshop on Hot Topics in Networks (HotNets)*, ser. HotNets '19, Princeton, NJ, USA: Association for Computing Machinery, 2019. DOI: 10.1145/3365609.3365848.

[54] R. Joshi, B. Leong, and M. C. Chan, "Timertasks: Towards time-driven execution in programmable dataplanes," in *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos*, 2019.

[55] J. Sonchack, A. J. Aviv, E. Keller, and J. M. Smith, "Turboflow: Information rich flow record generation on commodity switches," in *European Conference on Computer Systems (EuroSys)*, 2018. DOI: 10.1145/3190508.3190558.

[56] G. Franklin, D. Powell, and A. Emami-Naeini, *Feedback Control of Dynamic Systems*. 1995.

[57] P. Taheri, D. Menikkumbura, E. Vanini, S. Fahmy, P. Eugster, and T. Edsall, "Rocc: Robust congestion control for rdma," in *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, 2020, pp. 17–30.

[58] R. Pan, L. Breslau, B. Prabhakar, and S. Shenker, "Approximate fairness through differential dropping," *ACM Computer Communication Review (CCR)*, vol. 33, 2 2003. DOI: 10.1145/956981.956985.

[59] R. Pan, P. Natarajan, C. Piglione, *et al.*, "Pie: A lightweight control scheme to address the bufferbloat problem," in *IEEE International Conference on High Performance Switching and Routing (HPSR)*, IEEE, 2013. DOI: 10.1109/HPSR.2013.6602305.

[60] Y. Lu, M. Wang, B. Prabhakar, and F. Bonomi, "Elephanttrap: A low cost device for identifying large flows," in *IEEE Symposium on High-Performance Interconnects (HOTI)*, 2007. DOI: 10.1109/HOTI.2007.13.

[61] J. Ros-Giralt, A. Commike, S. Maji, and M. Veeraraghavan, "High speed elephant flow detection under partial information," in *IEEE International Symposium on Networks, Computers and Communications (ISNCC)*, IEEE, 2018. DOI: 10.1109/ISNCC.2018.8530979.

[62] *Pktgen*, https://pktgen-dpdk.readthedocs.io/en/latest/index.html, 2021.

[63] *Xilinx Vitis HLS*, https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/vitis_hls_process.html, 2020.

[64] *OMNeT++ Discrete Event Simulator*, https://omnetpp.org/, 2018.

[65] E. Vanini, R. Pan, M. Alizadeh, P. Taheri, and T. Edsall, "Let it flow: Resilient asymmetric load balancing with flowlet switching.," in *USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2017.

[66] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz, "Detail: Reducing the flow completion time tail in datacenter networks," in *ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*, 2012. DOI: 10.1145/2342356.2342390.

[67] *iPerf*, http://iperf.fr/, 2019.

[68] C. E. Leiserson, "Fat-trees: Universal networks for hardware-efficient supercomputing," *IEEE Transactions on Computers*, vol. 100, no. 10, 1985.

[69] V. Arun and H. Balakrishnan, "Copa: Practical Delay-Based Congestion Control for the Internet," in *USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2018, ISBN: 978-1-931971-43-0. DOI: 10.1145/3232755.3232783.

[70] P. Goyal, M. Alizadeh, and H. Balakrishnan, "Rethinking Congestion Control for Cellular Networks," in *ACM Workshop on Hot Topics in Networks (HotNets)*, 2017. DOI: 10.1145/3152434.3152437.

[71] K. Winstein and H. Balakrishnan, "TCP ex Machina: Computer-Generated Congestion Control," in *ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*, 2013, ISBN: 9781450320566. DOI: 10.1145/2486001.2486020.

[72] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "BBR: Congestion-Based Congestion Control," *ACM Queue*, vol. 14, no. 5, 2016, ISSN: 1542-7730. DOI: 10.1145/3012426.3022184.

[73] M. Dong, Q. Li, M. Schapira, D. Zarchy, and P. Brighten Godfrey, "PCC: Re-architecting Congestion Control for Consistent High Performance," in *USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2015. DOI: 10.5555/2789770.2789798.

[74] 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 *USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2012, ISBN: 978-931971-92-8.

[75] S. Huang, D. Dong, and W. Bai, "Congestion Control in High-speed Lossless Data Center Networks: A Survey," *Future Generation Computer Systems*, vol. 89, 2018. DOI: 10.1016/j.future.2018.06.036.

[76] D. Katabi, M. Handley, and C. Rohrs, "Congestion control for high bandwidth-delay product networks," in *ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*, 2002. DOI: 10.1145/633025.633035.

[77] C.-H. Tai, J. Zhu, and N. Dukkipati, "Making large scale deployment of rcp practical for real networks," in *IEEE International Conference on Computer Communications (INFOCOM)*, IEEE, 2008. DOI: 10.1109/INFOCOM.2008.285.

[78] J. Zhang, F. Ren, R. Shu, and P. Cheng, "Tfc: Token flow control in data center networks," in *European Conference on Computer Systems (EuroSys)*, 2016. DOI: 10.1145/2901318.2901336.

[79] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *ACM Internet Measurement Conference (IMC)*, 2010, pp. 267–280. DOI: 10.1145/1879141.1879175.

[80] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," *ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*, vol. 45, pp. 123–137, 2015. DOI: 10.1145/2785956.2787472.

[81] *802.1Qbb – Priority-based Flow Control*, https://1.ieee802.org/dcb/802-1qbb/, 2019.

[82] B. Stephens, A. L. Cox, A. Singla, J. Carter, C. Dixon, and W. Felter, "Practical DCB for improved data center networks," in *IEEE International Conference on Computer Communications (INFOCOM)*, 2014, pp. 1824–1832. DOI: 10.1109/INFOCOM.2014.6848121.

[83] A. Shpiner, E. Zahavi, V. Zdornov, T. Anker, and M. Kadosh, "Unlocking credit loop deadlocks," in *ACM Workshop on Hot Topics in Networks (HotNets)*, 2016, pp. 85–91. DOI: 10.1145/3005745.3005768.

[84] S. N. Avci, Z. Li, and F. Liu, "Congestion aware priority flow control in data center networks," in *IFIP Networking*, 2016, pp. 126–134. DOI: 10.1109/IFIPNetworking.2016.7497228.

[85] S. Hu, Y. Zhu, P. Cheng, *et al.*, "Tagger: Practical PFC deadlock prevention in data center networks," in *International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2017, pp. 451–463. DOI: 10.1145/3143361.3143382.

[86] *RoCE v2 Considerations*, https://community.mellanox.com/s/article/roce-v2-considerations, 2019.

[87] Y. Zhu, H. Eran, D. Firestone, *et al.*, "Congestion control for large-scale RDMA deployments," *ACM Computer Communication Review (CCR)*, vol. 45, pp. 523–536, 2015. DOI: 10.1145/2829988.2787484.

[88] R. Mittal, V. T. Lam, N. Dukkipati, *et al.*, "TIMELY: RTT-based congestion control for the datacenter," *ACM Computer Communication Review (CCR)*, vol. 45, no. 4, pp. 537–550, 2015. DOI: 10.1145/2829988.2787510.

[89] Y. Li, R. Miao, H. H. Liu, *et al.*, "HPCC: High precision congestion control," in *ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*, 2019, pp. 44–58. DOI: 10.1145/3341302.3342085.

[90] M. P. Grosvenor, M. Schwarzkopf, I. Gog, *et al.*, "Queues don't matter when you can JUMP them!" In *USENIX Conference on Networked Systems Design and Implementation (NSDI)*, Oakland, CA, May 2015, pp. 1–14.

[91] M. Alizadeh, S. Yang, M. Sharif, *et al.*, "Pfabric: Minimal near-optimal datacenter transport," *ACM Computer Communication Review (CCR)*, vol. 43, pp. 435–446, 2013. DOI: 10.1145/2534169.2486031.

[92] M. Karol, S. J. Golestani, and D. Lee, "Prevention of deadlocks and livelocks in lossless backpressured packet networks," *IEEE/ACM Transactions on Networking (ToN)*, vol. 11, pp. 923–934, 2003. DOI: 10.1109/TNET.2003.820434.

[93] J. Coffman Edward G. and A. Elphick Michael J. nad Shoshani, "System Deadlocks," *ACM Computing Surveys*, vol. 3, no. 2, pp. 67–78, 1971. DOI: 10.1145/356586.356588.

[94] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," *ACM Computer Communication Review (CCR)*, vol. 38, pp. 63–74, 2008. DOI: 10.1145/1402946.1402967.

[95] *FPGA*, https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html, 2021.

[96] *Intelligent Buffer Management on Cisco Nexus 9000 Series Switches*, 2017.

[97] M. Alizadeh, T. Edsall, S. Dharmapurikar, *et al.*, "CONGA: Distributed congestion-aware load balancing for datacenters," *ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*, vol. 44, pp. 503–514, 2014. DOI: 10.1145/2619239.2626316.

[98] V. Shrivastav, "Fast, scalable, and programmable packet scheduler in hardware," in *ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*, 2019, pp. 367–379. DOI: 10.1145/3341302.3342090.

[99] *Escape repository*, https://github.com/escape-repo, 2021.

[100] *OMNeT++ Discrete Event Simulator*, https://omnetpp.org/, 2018.

[101] C. Guo, H. Wu, Z. Deng, *et al.*, "RDMA over commodity ethernet at scale," in *ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*, 2016, pp. 202–215. DOI: 10.1145/2934872.2934908.

[102] W. J. Dally and H. Aoki, "Deadlock-free adaptive routing in multicomputer networks using virtual channels," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, pp. 466–475, 1993. DOI: 10.1109/71.219761.

[103] J. Wu and L. Sheng, "Deadlock-free routing in irregular networks using prefix routing," in *Proc. of the ISCA 12th International Conference on Parallel and Distributed Computing Systems*, 1999, pp. 424–430.

[104] J. C. Sancho, A. Robles, J. Flich, P. Lopez, and J. Duato, "Effective methodology for deadlock-free minimal routing in InfiniBand networks," in *International Conference on Parallel Processing (ICPP)*, 2002, pp. 409–418. DOI: 10.1109/ICPP.2002.1040897.

[105] J. Domke, T. Hoefler, and W. E. Nagel, "Deadlock-free oblivious routing for arbitrary topologies," in *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2011, pp. 616–627. DOI: 10.1109/IPDPS.2011.65.

[106] J. Duato, "A new theory of deadlock-free adaptive routing in wormhole networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 12, pp. 1320–1331, 1993. DOI: 10.1109/71.250114.

[107] J. Flich, T. Skeie, A. Mejia, *et al.*, "A survey and evaluation of topology-agnostic deterministic routing algorithms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 3, pp. 405–425, 2011. DOI: 10.1109/TPDS.2011.190.

[108] K. Anjan and T. M. Pinkston, "DISHA: A deadlock recovery scheme for fully adaptive routing," in *International Parallel Processing Symposium (IPPS)*, 1995. DOI: 10.1109/IPPS.1995.395983.

[109] Y. Tamir and G. L. Frazier, "High-performance multi-queue buffers for vlsi communications switches," *ACM SIGARCH Computer Architecture News*, vol. 16, no. 2, pp. 343–354, 1988. DOI: 10.1145/633625.52439.

[110] K. Zarifis, R. Miao, M. Calder, E. Katz-Bassett, M. Yu, and J. Padhye, "Dibs: Just-in-time congestion mitigation for data centers," in *European Conference on Computer Systems (EuroSys)*, 2014, pp. 1–14. DOI: 10.1145/2592798.2592806.

[111] M. Budiu and C. Dodd, "The p416 programming language," *ACM SIGOPS Operating Systems Review*, vol. 51, no. 1, 2017, ISSN: 0163-5980. DOI: 10.1145/3139645.3139648.

[112] G. Ramamurthy and B. Sengupta, "A predictive hop-by-hop congestion control policy for high speed networks," in *IEEE International Conference on Computer Communications (INFOCOM)*, IEEE, 1993, pp. 1033–1041. DOI: 10.1109/INFCOM.1993.253262.

[113] S. Jagannathan and J. Talluri, "Predictive congestion control of atm networks: Multiple sources/single buffer scenario," *Automatica*, vol. 38, no. 5, pp. 815–820, 2002. DOI: 10.1016/S0005-1098(01)00259-X.

[114] M. Zawodniok and S. Jagannathan, "Predictive congestion control protocol for wireless sensor networks," *IEEE Transactions on Wireless Communications*, vol. 6, no. 11, pp. 3955–3963, 2007. DOI: 10.1109/TWC.2007.051035.

[115] P. G. Kulkarni, S. I. McClean, G. P. Parr, and M. M. Black, "Proactive predictive queue management for improved qos in ip networks," in *International Conference on Networking, International Conference on Systems and International Conference on Mobile Communications and Learning Technologies (ICNICONSMCL)*, IEEE, 2006, pp. 7–7. DOI: 10.1109/ICNICONSMCL.2006.175.